Tarena

# 嵌入式教学课程

## Linux设备驱动

Oliver

# 知识点

- 嵌入式Linux开发基础
- Linux内核编程
- Linux字符设备驱动
- Linux设备驱动模型
- Linux块设备驱动
- Linux网络设备驱动
- 驱动实例（触摸屏和LCD驱动）

# 特别声明

- 1. 本文档涉及的PC Linux环境为Fedora 6
- 2. 本文档涉及的ARM板为Real 6410开发板(基于三星S3C6410芯片)
- 3. 本文档涉及的Linux内核版本为ARM板上工作的内核Linux 2.6.28.6

# 嵌入式Linux开发环境

- 嵌入式Linux开发硬件环境
- 嵌入式Linux开发软件环境

# 嵌入式Linux开发硬件环境

- 目标板(目标机)
  - ARM开发板
- PC机(开发机)
  - 运行开发工具的平台

# 嵌入式Linux开发软件环境

- 开发机操作系统
  - PC Linux平台
- 目标机编译器
  - 交叉编译工具链

# 嵌入式Linux系统构成

- 嵌入式Linux系统硬件构成
- 嵌入式Linux系统软件构成

# 嵌入式Linux系统硬件构成

- 处理器
  - S3C6410(ARM11)
- 存储设备
  - Nor/Nand闪存
- 内存设备
  - RAM

# 嵌入式Linux系统软件构成

- 启动代码
  - u-boot(版本： 1.1.6)
- Linux内核
  - Linux Kernel(版本： 2.6.28.6)
- 文件系统
  - Busybox(版本： 1.13)
  - QT(版本： 4.5.2)

# Nand Flash地址分配

| Nand Flash地址范围 | 大小 | 内容 |
|---|---|---|
| 0x0000 0000 – 0x0003 FFFF | 256KB | Bootloader(u-boot) |
| 0x0004 0000 – 0x003F FFFF | 3.75MB | Kernel(Linux) |
| 0x0040 0000 – 0x007F FFFF | 4MB | Cramfs文件系统 |
| 0x0080 0000 – 0x0FFF FFFF | 248MB | 真实文件系统(ubifs) |

# TFTP服务器配置

- TFTP服务器和客户端软件包安装
  - rpm -qa | grep tftp

```
[root@Fedora6 ~]# rpm -qa | grep tftp
tftp-server-0.42-3.1
tftp-0.42-3.1
```

# TFTP服务器配置

- TFTP服务配置修改
  - vim /etc/xinetd.d/tftp

```
service tftp
{
        socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -s /opt/tftpboot -c
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

# TFTP服务器配置

- TFTP服务启动
  - service xinetd restart
  - /etc/init.d/xinetd restart

  - chkconfig tftp [on/off]
  - setup
  - 图形界面：System -> Administration -> Server Settings -> Services

# TFTP服务器配置

- TFTP服务测试
  - 登录本机IP或回环地址验证tftp服务配置
  - tftp 127.0.0.1

# 擦除整个Nand Flash内容

- 设置目标板启动选择拨码开关为SD卡启动模式
- 从SD卡启动目标板进入u-boot
- 擦除整个Nand Flash内容
  - nand erase 0

# 烧写u-boot

- 准备u-boot文件
  - 拷贝光盘中u-boot.bin到TFTP服务器目录
- 下载u-boot文件到目标板内存
  - tftpboot c0008000 u-boot.bin
- 擦除Nand Flash中待写入u-boot区块
  - nand erase 0 40000
- 写入u-boot到Nand Flash中
  - nand write c0008000 0 40000

# 从Nand Flash启动u-boot

- 设置目标板启动选择拨码开关为从Nand Flash启动模式
- 从Nand Flash启动目标板进入u-boot

# 烧写Linux内核

- 准备Linux内核文件
  - 拷贝光盘中Linux内核文件zImage到TFTP服务器目录
- 下载zImage文件到目标板内存
  - tftpboot c0008000 zImage
- 擦除Nand Flash中待写入zImage区块
  - nand erase 40000 300000
- 烧写zImage到Nand Flash中
  - nand write c0008000 40000 300000

# 配置u-boot加载Linux内核

- 从Nand Flash启动目标板进入u-boot
- 设置bootcmd环境变量
  - setenv bootcmd "nand read c0008000 40000 300000; bootm c0008000"
- 保存环境变量
  - saveenv
- 重启动目标板
  - reset

# 烧写Cramfs文件系统

- 准备Cramfs文件系统文件
  - 拷贝光盘中cramfs文件系统文件root_mkfs.cramfs到TFTP服务器目录
- 下载root_mkfs.cramfs文件到目标板内存
  - tftpboot c0008000 root_mkfs.cramfs
- 擦除Nand Flash中待写入root_mkfs.cramfs区块
  - nand erase 400000 400000
- 烧写root_mkfs.cramfs到Nand Flash中
  - nand write c0008000 400000 400000

# 配置Linux内核加载cramfs

- 从Nand Flash启动目标板进入u-boot
- 设置bootargs环境变量
  - setenv bootargs noinitrd root=/dev/mtdblock0 console=ttySAC0 init=/linuxrc
- 保存环境变量
  - saveenv
- 重启动目标板
  - reset

# 烧写真实文件系统(ubifs)

- 准备ubifs文件系统文件
  - 拷贝光盘中真实文件系统文件qtopia.tar.gz到SD卡
- 烧写真实文件系统文件qtopia.tar.gz内容到Nand Flash中
  - 在u-boot菜单中选择[q] Burn qtopia image((put qtopia.tar.gz in SD/Udisk/tftp first))

# 配置Linux内核加载真实文件系统(ubifs)

- 在新目标板中此页步骤可跳过(已被自动完成)
- 从Nand Flash启动目标板进入u-boot
- 设置bootargs环境变量
  - setenv bootargs noinitrd root=ubi0:rootfs rootfstype=ubifs ubi.mtd=1 console=ttySAC0 init=/linuxrc video=fb:WX4300F ppp=none
- 保存环境变量
  - saveenv
- 重启动目标板
  - reset

# 嵌入式Linux软件开发

- 编写C语言Hello World程序
- 编译ARM Linux平台运行的Hello World程序

# 嵌入式Linux软件开发

- 启动目标板系统ftp服务和telnet服务
  - inetd
  - ftpd
  - telnetd
- 上传ARM Linux版本Hello World到目标板运行

# PC机Linux系统FTP服务器配置

- FTP服务器软件包安装
  - rpm -qa | grep ftp

```
[root@Fedora6 ~]# rpm -qa | grep ftp
vsftpd-2.0.5-8
lftp-3.5.1-2.fc6
tftp-server-0.42-3.1
gftp-2.0.18-3.2.2
tftp-0.42-3.1
ftp-0.17-33.fc6
```

# PC机Linux系统FTP服务器配置

- 默认FTP服务器目录
  - /var/ftp
- 添加FTP服务用户组和用户
  - groupadd ftp
  - useradd [ftpusername]
  - passwd [ftpusername]

# PC机Linux系统FTP服务器配置

- FTP服务启动
  - service vsftpd restart
  - /etc/init.d/vsftpd restart
  - chkconfig --level 5 vsftpd [on/off]
  - setup
  - 图形界面：System -> Administration -> Server Settings -> Services

# PC机Linux系统FTP服务器配置

- 允许root用户登录FTP服务配置
  - userdel –r ftp
  - groupadd ftp
  - useradd –d /opt/ftp –g ftp –s /sbin/nologin [ftpusername]
  - chmod 755 /opt/ftp
  - chown –R root.root /opt/ftp
  - vim /etc/vsftpd/ftpusers
  - vim /etc/vsftpd/user_list

# PC机Linux系统FTP服务器配置

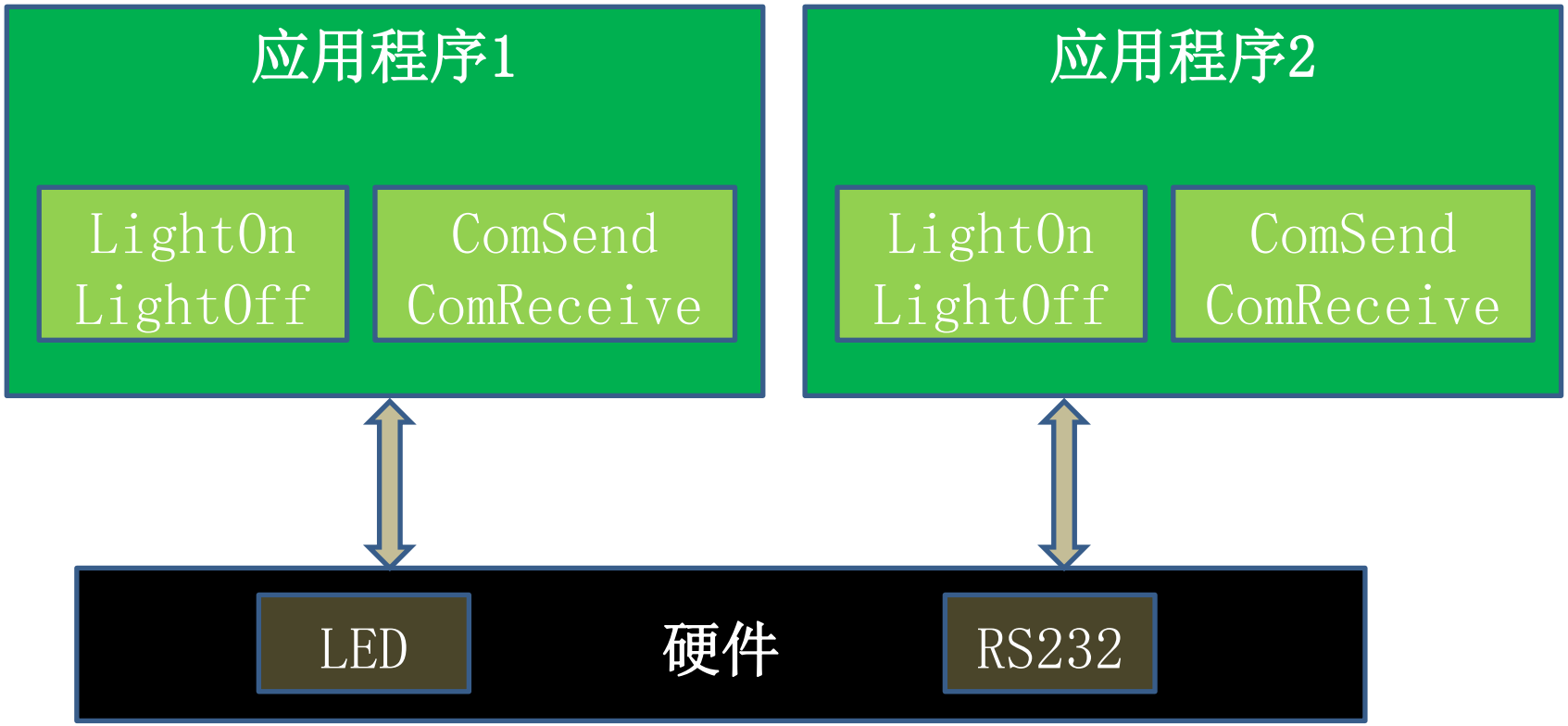
- FTP服务测试
  - 登录本机IP或回环地址验证ftp服务配置
  - ftp 127.0.0.1

# 认识设备驱动

- 什么是设备驱动？
  - 使硬件正常工作的软件

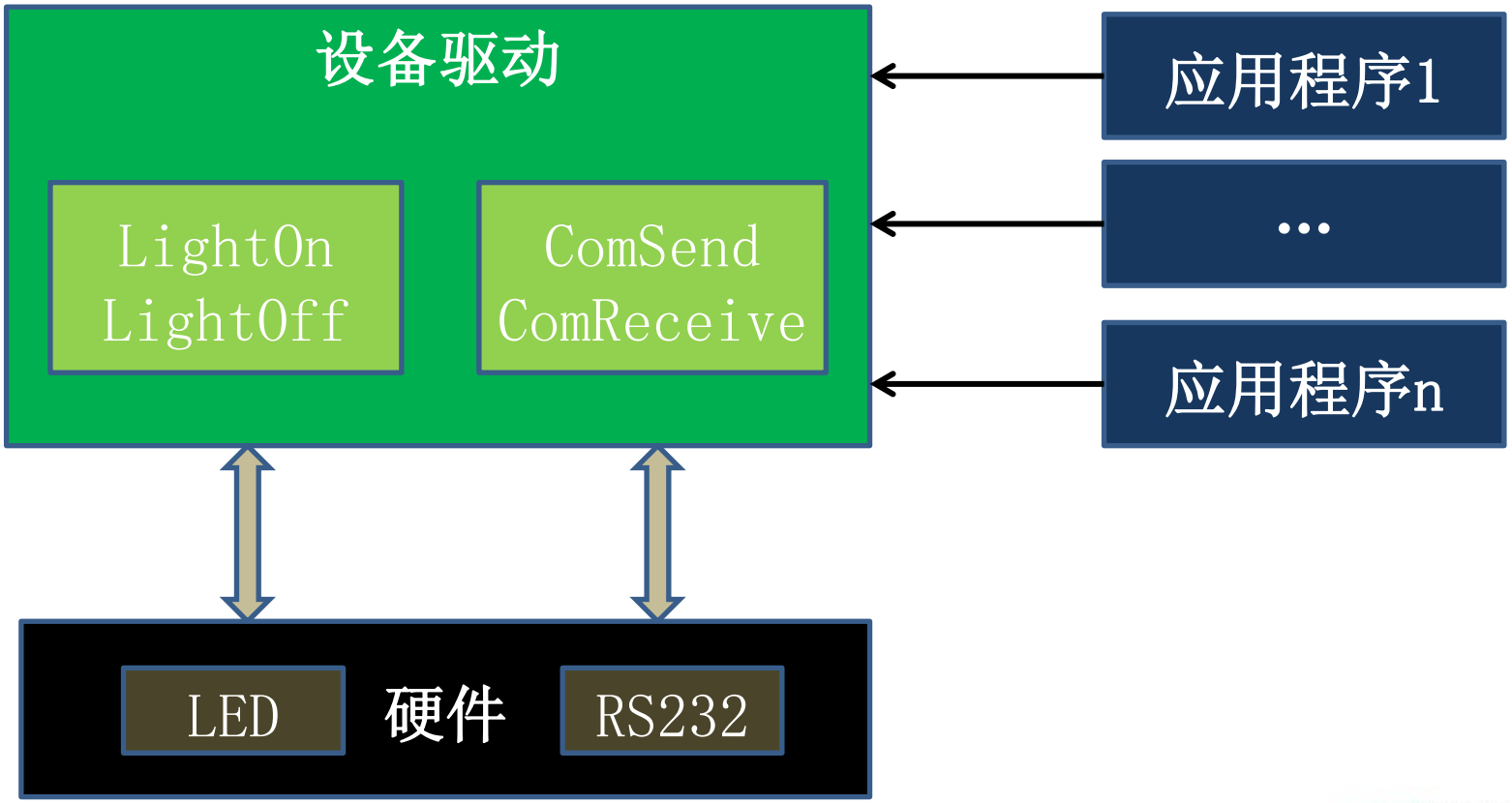
# 认识设备驱动

- 设备驱动模型1



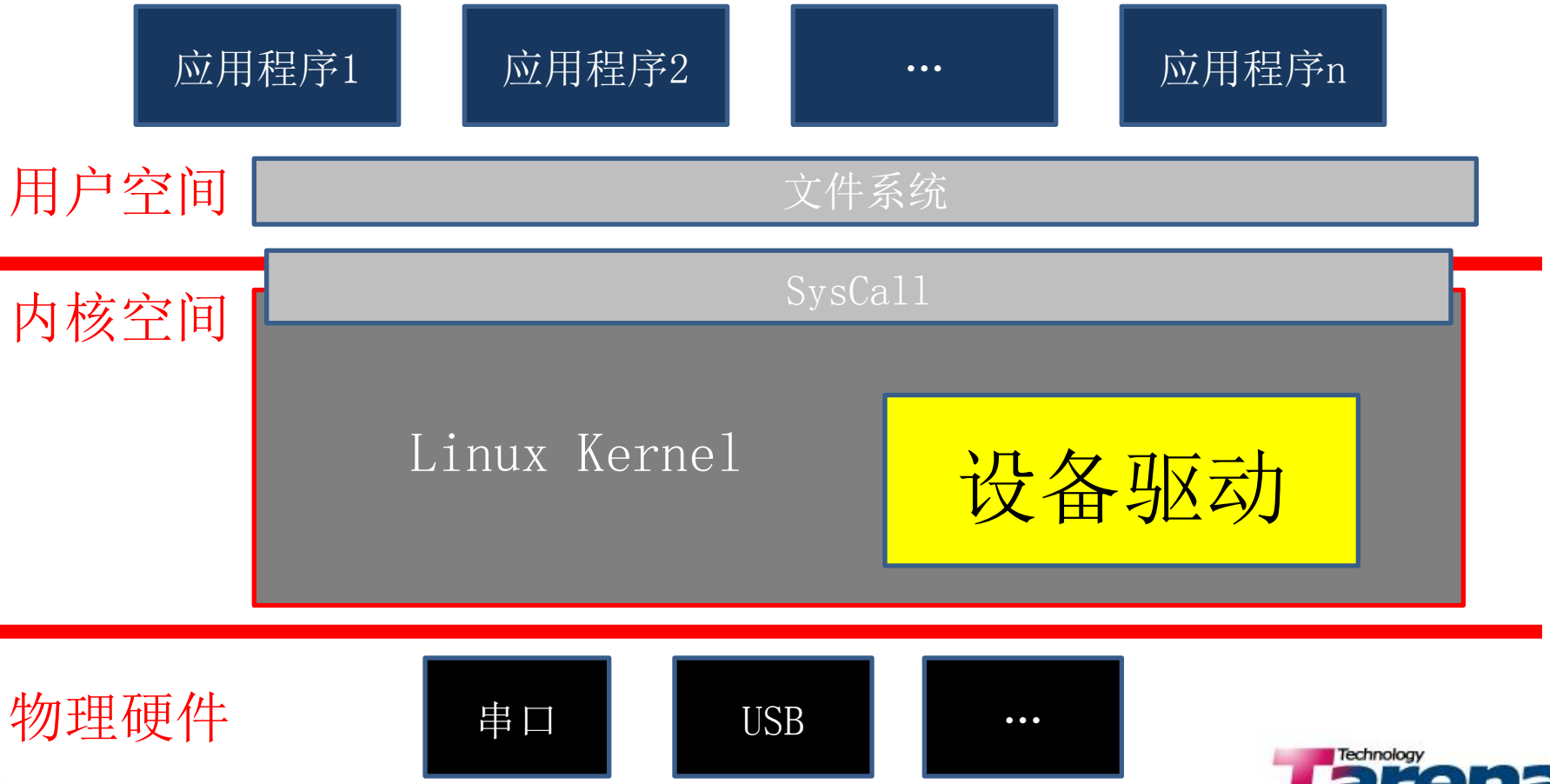
# 认识设备驱动

- 设备驱动模型2



# 认识设备驱动

- Linux系统设备驱动模型



# 内核编程语言工具链

- 交叉编译工具链(GNU Toolchain)
  - 语言编译工具(GNU C/C++等)
  - 二进制代码工具(链接等工具)
  - 标准LIBC代码库(glibc)

# 内核编程语言

- GNU C与ANSI C异同
  - 1. 命名习惯
  - 2. 零长度数组
  - 3. case语句
  - 4. 语句表达式
  - 5. typeof关键字
  - 6. 可变参数的宏
  - 7. 标号处理

# 内核编程语言

- GNU C与ANSI C异同
  - 8. 当前文件名称、函数名称和行号
  - 9. 特殊属性声明
  - 10. 内建函数
  - 11. do {} while(0);
  - 12. goto语句

# 内核编程语言

- GNU C与ANSI C异同
  - 1. 命名习惯
    - ANSI C
      - 宏名：全部用大写字母
      - 变量名：第一个单词全小写，后面所有单词首字母大写
      - 函数名：驼峰命名规则
    - GNU C
      - 宏名：全部用大写字母
      - 变量名：全部小写，各单词之间用下划线(_)连接
      - 函数名：全部小写，各单词之间用下划线(_)连接

# 内核编程语言

- GNU C与ANSI C异同
  - 2. 零长度数组
    - GNU C支持零长度数组
    - 常用于定义变长对象的头结构体中

# 内核编程语言

- GNU C与ANSI C异同
  - 3. case语句
    - GNU C支持类似case x...y的语法
    - 在[x,y]范围内的数都满足该case语句

# 内核编程语言

- GNU C与ANSI C异同
  - 4. 语句表达式
    - GNU C将包含在{}中的符合语句看作一个表达式，称为语句表达式
    - 语句表达式可以出现在任意允许表达式的地方
    - 常见于宏定义中使用局部变量避免参数被修改

# 内核编程语言

- GNU C与ANSI C异同
  - 5. typeof关键字
    - 获取变量的数据类型
    - 常用于定义变量时变量数据类型不确定的情况

# 内核编程语言

- GNU C与ANSI C异同
  - 6. 可变参数的宏
    - ANSI C
      - 支持可变参数的函数
    - GNU C
      - 支持可变参数的函数
      - 支持可变参数的宏

# 内核编程语言

- GNU C与ANSI C异同
  - 7. 标号处理
    - ANSI C
      - 要求数组和结构体的初始化值必须以固定的顺序出现
    - GNU C
      - 允许用”[INDEX]=”或”[FIRST...LAST]=”的形式为数组初始化值指定范围
      - 允许用”.member=VALURE,”的形式为结构体初始化值
      - 在结构体初始化中允许成员顺序不按照结构体定义的顺序出现

# 内核编程语言

- GNU C与ANSI C异同
  - 8. 当前文件名称、函数名称和行号
    - GNU C中包含几个预定义标识符
      - __FILE__表示当前源代码文件名称
      - __FUNCTION__表示当前源代码函数名称
      - __LINE__表示当前源代码行号

# 内核编程语言

- GNU C与ANSI C异同
  - 9. 特殊属性声明
    - GNU C允许声明函数、变量和类型的特殊属性，用于需要手动进行代码优化和定制代码检查方法等特殊地方
    - 使用方法：__attribute__((ATTRIBUTE))
    - 常见属性：
      - noreturn: 函数从不返回
      - format: 函数使用printf/scanf风格的可变参数
      - section: 将代码或数据链接到指定区段
      - aligned: 指定变量、结构体、联合体的数据边界对齐方式
      - packed: 指定变量或结构体成员使用最小边界对齐，或指定枚举、结构体、联合体使用最小内存
      - unused: 指定不用的函数或变量，避免编译警告

# 内核编程语言

- GNU C与ANSI C异同
  - 10. 内建函数
    - GNU C提供大量内建函数
    - 所有内建函数名称通常以”__builtin”开始
    - 常见内建函数：
      - __builtin_return_address(LEVEL): 返回当前函数或其调用者的返回地址
      - __builtin_constant_p(EXP): 判断一个表达式是否为编译时能确定的常数
      - __builtin_expect(EXP, C): 为编译器提供分支预测信息，返回正数表达式EXP的值

# 内核编程语言

- GNU C与ANSI C异同
  - 11. do {} while(0);
    - 在Linux内核源代码中，有很多这样的语句
    - 常见于宏定义中

# 内核编程语言

- GNU C与ANSI C异同
  - 12. goto语句
    - goto语句是C语言中的著名争议话题
    - goto语句能使代码变得简洁高效
    - 在Linux内核源代码中，有很多这样的语句
    - 一般只限于错误处理中

# 认识Linux内核

- Linux内核起源和发展
  - UNIX操作系统
  - Minix操作系统
  - GNU计划
  - POSIX标准
  - Internet网络

# 认识Linux内核

- Linux内版本历史
  - V0.1 初稿
  - V1.0 开始支持386，单CPU
  - V1.2 多平台(Alpha、Sparc、MIPS等)支持
  - V2.0 更多新平台，开始支持SMP
  - V2.2 提升SMP系统性能，支持更多硬件
  - V2.4 提升SMP系统扩展性，集成桌面系统特性支持
  - V2.6 提升对企业级服务器、嵌入式等支持

# 认识Linux内核

- Linux内核主要分支
  - Hard Hat Linux
  - RTLinux
  - ucLinux
  - ThinLinux
  - MontaVista

# 认识Linux内核

- 基于Linux内核的常见发行版本
  - Red Hat/Fedora/RHES
  - CentOS
  - Debian
  - Ubuntu
  - SUSE
  - TurboLinux
  - RedFlag
  - Xteam

# 认识Linux内核

- Linux内核源代码树(Version 2.6.28.6)
  - arch: 与CPU体系结构相关代码
  - block: 块设备驱动管理相关代码
  - crypto: 加密算法、散列算法、压缩、CRC等
  - Documentations: 内核各部分通用解释文档
  - Drivers:设备驱动代码
  - firmware:固件相关代码
  - fs:当前内核支持的所有文件系统相关代码
  - include:公共头文件

# 认识Linux内核

- Linux内核源代码树(Version 2.6.28.6)
  - init:内核初始化代码
  - ipc:进程间通信代码
  - kernel:内核核心代码，比如：进程调度、定时器等
  - lib:内核库代码
  - Makefile:整个内核代码工程文件
  - mm:内存管理代码
  - net:网络相关代码，主要是协议层代码

# 认识Linux内核

- Linux内核源代码树(Version 2.6.28.6)
  - samples:例程代码
  - scripts:用于配置内核的脚本处理相关代码
  - security:用于SELinux相关代码
  - sound:音频设备驱动管理代码
  - tools:内核工具
  - usr:实现一个cpio
  - virt:虚拟机制相关代码

# 认识Linux内核

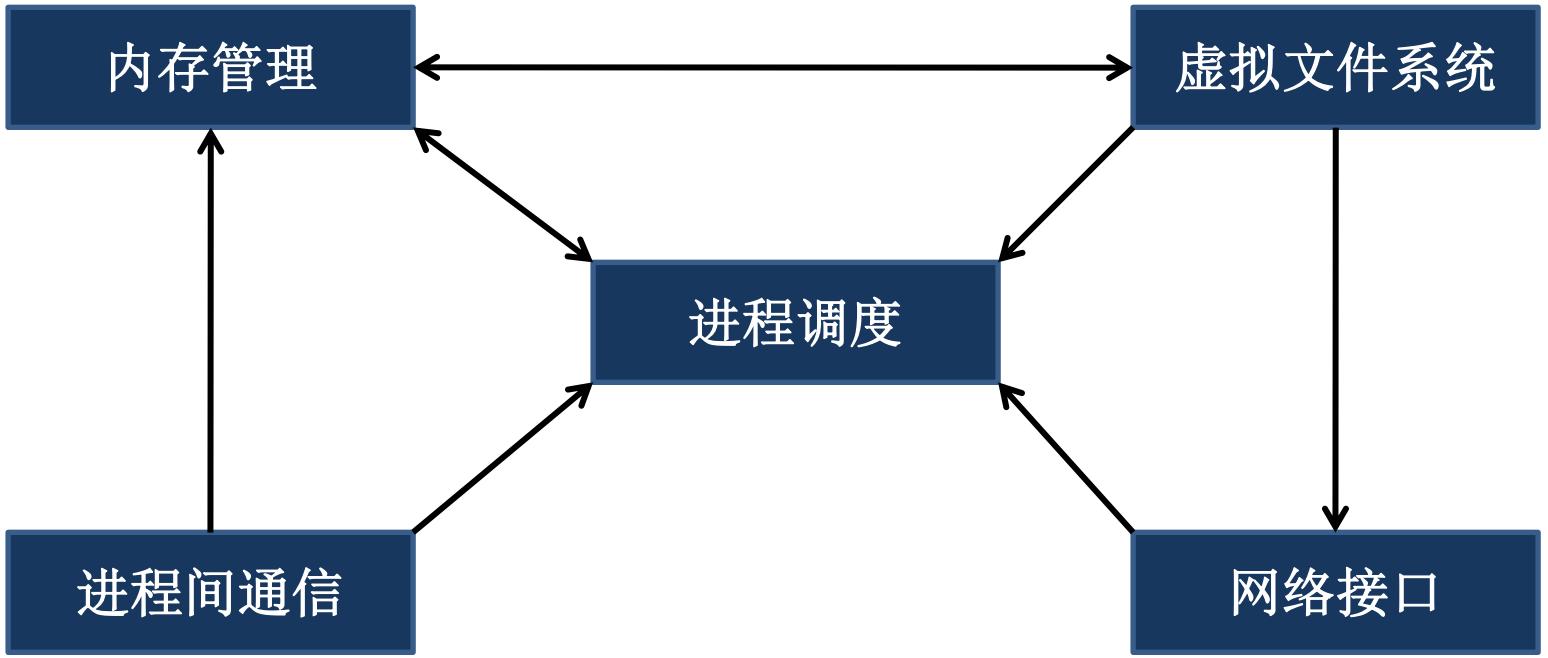
- Linux内核源代码树arch目录(S3C6410)
  - arm: 使用ARM体系结构芯片代码
    - boot: 引导程序代码
      - compressed: 内核压缩/解压缩代码
    - common: 公共代码
    - configs: 针对所有使用ARM体系结构芯片的默认内核配置参数
    - include: 与体系结构相关头文件
    - kernel: 与体系结构相关内核核心代码
    - lib: 与体系结构相关内核库代码

# 认识Linux内核

- Linux内核源代码树arch目录(S3C6410)
  - arm: 使用ARM体系结构芯片代码
    - mach-s3c64xx: 与三星S3C64xx系列芯片相关代码
    - mm: 与体系结构相关内存管理代码
    - plat-s3c: 针对三星s3c系列芯片内部资源功能代码
      - include: 三星s3c系列芯片相关头文件
    - plat-s3c64xx: 针对三星s3c64xx系列芯片内部资源功能代码
      - include: 三星S3C64xx系列芯片相关头文件
    - tools: 机器类型相关工具
      - mach-types: 列出当前内核代码中支持的所有机器类型

# 认识Linux内核

- Linux内核组成部分




表示依赖关系

# 认识Linux内核

- Linux内核空间和用户空间
  - usr：用户模式
  - fiq：快速中断模式
  - irq：外部中断模式
  - svc：管理模式
  - abort：数据访问终止模式
  - sys：系统模式
  - und：未定义指令中止模式

# 配置Linux内核

- Linux内核配置方法
  - make config（基于文本的最为传统的交互配置界面，不推荐使用）
  - make menuconfig（基于文本菜单的配置界面）
  - make xconfig（基于QT图形配置界面）
  - make gconfig（基于GTK+的图形配置界面）

# 配置Linux内核

- Linux内核配置方法
  - 内核配置文件(Kconfig文件)
    - .config文件(内核配置文件)
  - 内核代码工程文件(Makefile文件)
    - Makefile(内核源代码工程管理文件)

# 配置Linux内核

- Linux内核配置文件语法(Kconfig)
  - config: 为内核配置添加新的配置选项
  - menu: 为内核配置添加新的菜单项
  - bool: 设定内核配置项的配置类型
  - tristrate: 设定内核配置项的配置类型
  - help: 为内核配置项添加帮助说明
  - default: 设定内核配置项的默认值

# 配置Linux内核

- 添加新的内核配置选项
  - 添加新的内核配置选项
  - 使用make menuconfig命令测试并配置新的内核配置选项
  - 保存内核配置
  - 在.config文件中检查新配置选项

# 配置Linux内核

- Linux内核源代码工程文件语法(Makefile)
  - obj-变量
  - obj-y变量
  - obj-m变量

# 配置Linux内核

- 添加代码到Linux内核工程
  - 将代码添加到内核目录
  - 根据新的内核配置选项和代码文件名称修改Makefile文件
  - 重新编译内核
  - 测试内核

# 编译Linux内核

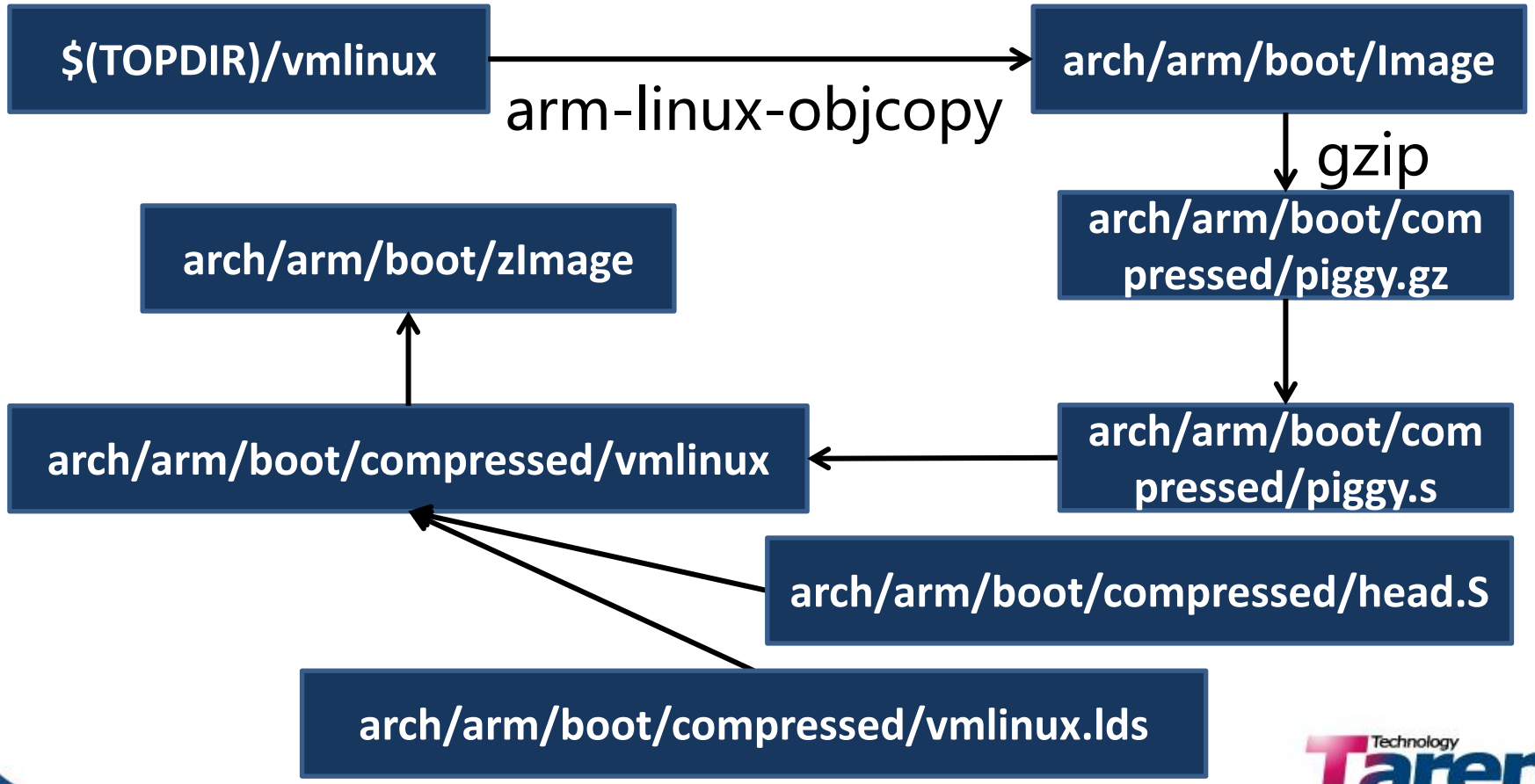
- Linux内核源代码编译
  - make zImage (生成压缩的内核镜像)
  - make Image (生成非压缩的内核镜像)
  - make xipImage (Execute-In-Place，常见生成用于Nor Flash的内核镜像)
  - make uImage (生成带u-boot封装的内核镜像)
  - make bootpImage (生成压缩的内核镜像和初始化RAM Disk镜像的组合镜像)
  - make modules (编译内核中所有独立模块)

# 编译Linux内核

- Linux内核编译生成文件
  - $(TOPDIR)/vmlinux (非压缩内核，ELF格式)
  - arch/$(ARCH)/boot/Image (非压缩内核，BIN格式)
  - arch/$(ARCH)/boot/compressed/vmlinux (具备自解压功能的压缩内核，ELF格式)
  - arch/$(ARCH)/boot/zImage (具备自解压功能的压缩内核，BIN格式)

# 编译Linux内核

- Linux内核编译生成镜像过程

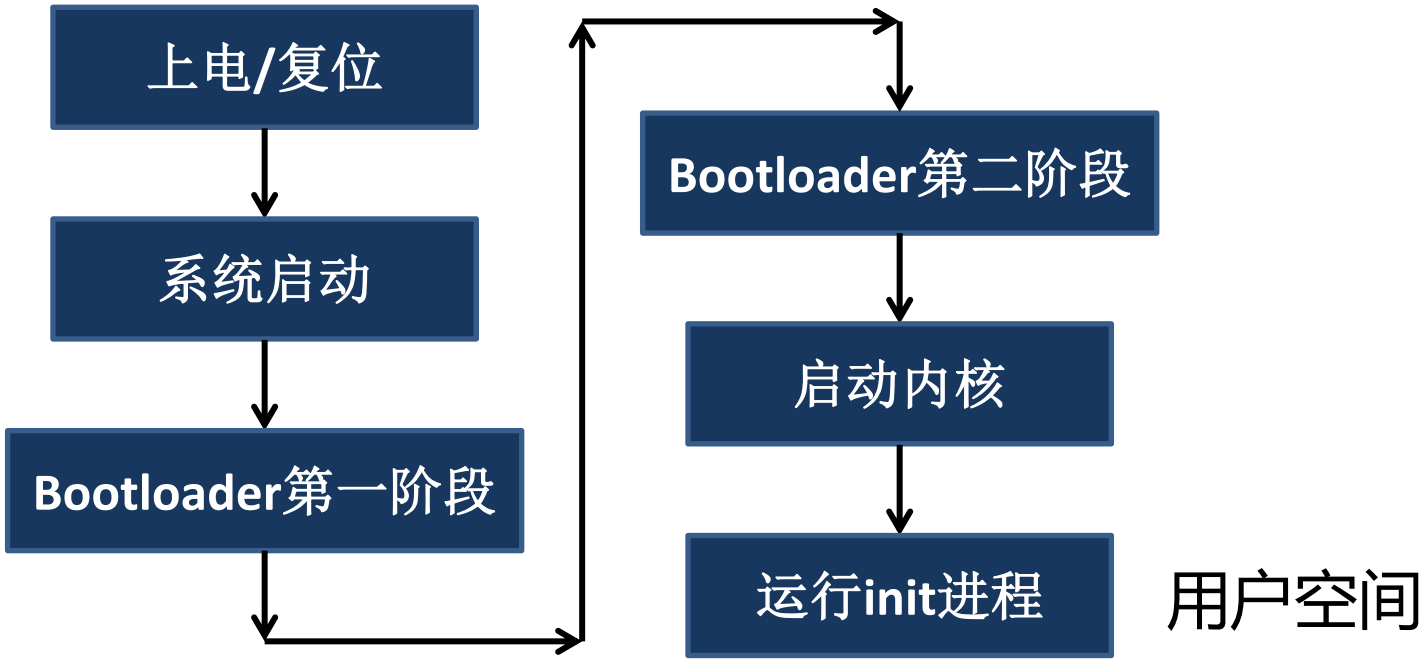


# 安装Linux内核

- Linux内核镜像安装
  - make install (安装非压缩的内核镜像)
  - make zinstall (安装压缩的内核镜像)
  - make modules_install (安装内核中所有独立模块到文件系统，可用INSTALL_MOD_PATH变量指定安装路径)

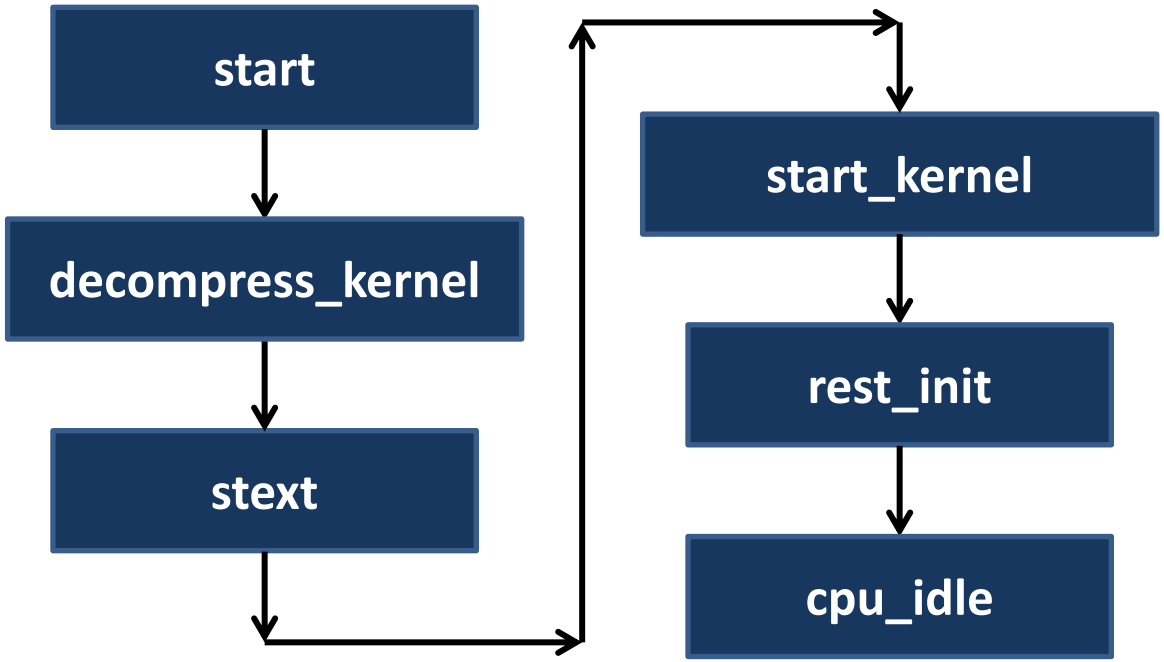
# Linux内核引导过程

- Linux系统引导流程(PC X86平台)



# Linux内核引导过程

- Linux内核引导流程



# Linux内核引导过程

- Linux内核引导流程
  - start: arch/arm/boot/compressed/head.S
  - decompress_kernel: arch/arm/boot/compressed/head.S
  - stext: arch/arm/kernel/head.S
  - start_kernel: init/main.c
  - rest_init: init/main.c
  - cpu_idle: arch/arm/kernel/process.c

# Linux内核代码地址重要常量

- Linux内核代码中几个重要常量
  - PAGE_OFFSET: 内核空间起始地址(0xC0000000)
  - TEXT_OFFSET: 内核代码段偏移量(0x00008000)
  - PHYS_OFFSET: 内核内存物理起始地址(0x50000000)
  - KERNEL_RAM_VADDR: 内核代码起始虚拟地址(0xC0008000)
  - KERNEL_RAM_PADDR: 内核代码起始物理地址(0x50008000)

# Linux内核Nand Flash分区表

- Linux内核代码中Nand Flash分区表
  - arch/arm/plat-s3c/include/plat/partition.h
    - struct mtd_partition s3c_partition_info[] = {
      - // Bootloader
      - // Kernel
      - cramfs
      - ubifs

# Linux内核编程

- 内核头文件
  - include/linux目录
  - include/asm目录
  - init.h和module.h头文件

# Linux内核编程

- 模块初始化函数
  - 模块初始化函数体

```
static int __init module_initialization_function(void)
{
    // 模块初始化代码
}
```

  - 模块初始化函数声明

```
module_init(module_initialization_function);
```

# Linux内核编程

- 模块退出函数
  - 模块退出函数体

```
static void __exit module_cleanup_function(void)
{
    // 模块退出代码
}
```

  - 模块退出函数声明

```
module_exit(module_cleanup_function);
```

# Linux内核编程

- __init标识
  - 该标识将指定的函数在连接的时候被放在.init.text代码区段，并在.initcall.init区段保存该函数指针，内核初始化时调用.initcall.init区段所有函数，在初始化完成后释放这两个区段内存
- __exit标识
  - 该标识将指定的函数在连接的时候被放在.exit.text代码区段，和__init作用类似，使对应函数在运行完成后自动回收内存

# Linux内核编程

- 添加内核代码方法
  - 将新内核代码源文件添加到内核源代码的指定目录
  - 修改指定目录下的Kconfig文件添加对新内核代码的配置
  - 修改指定目录下的Makefile文件添加对新内核代码的编译支持

# Linux内核编程

- 编写内核代码
  - 完成Hello World内核代码并放入指定内核代码目录
  - 为Hello World内核代码修改指定内核代码目录Kconfig文件并指定选项为bool配置选项
  - 使用make menuconfig命令配置Hello World
  - 修改指定内核代码目录Makefile文件，针对配置选项和源代码文件修改Makefile添加编译支持
  - 使用make zImage命令生成包含Hello World代码的新内核镜像文件
  - 测试包含Hello World代码的新内核镜像文件
  - 查看Hello World代码中的输出信息

# Linux内核编程

- 内核代码问题
  - 直接编译进内核将导致内核变大
  - 在现有内核中增加或删除功能，需要重新编译内核

# 配置u-boot通过TFTP服务加载内核

- 拷贝光盘中Linux内核文件zImage到TFTP服务器目录
- 从Nand Flash启动目标板进入u-boot
- 设置bootcmd环境变量
  - setenv bootcmd "tftpboot c0008000 zImage; bootm c0008000 "
- 保存环境变量
  - saveenv

# Linux内核编程

- 内核模块
  - 内核模块本身不被编译进内核镜像
  - 内核模块一旦被加载，和内核中的其它部分完全一样
  - 内核模块以.ko为文件名结尾

# Linux内核编程

- 内核模块操作命令
  - modprobe
  - lsmod
  - insmod
  - rmmod
  - modinfo

# Linux内核编程

- 内核模块操作命令
  - modprobe
    - 加载内核模块到内存运行，或者卸载一个正在内存中运行的内核模块
    - 在内核模块加载过程中，如果该内核模块有依赖模块，该命令将同时加载该内核模块的依赖模块
  - 用例：
    - modprobe modulename
    - modprobe –r modulename

# Linux内核编程

- 内核模块操作命令
  - lsmod
    - 该命令列出系统中正在运行的内核模块
    - 该命令实际通过读取/proc/modules文件中的内容并解析来获取相应信息
  - 用例：
    - lsmod

# Linux内核编程

- 内核模块操作命令
  - insmod
    - 该命令加载指定内核模块到内存中运行
    - 该命令不检测被加载内核模块是否有依赖模块，如果有，并且依赖模块还没有被加载，该加载操作将不能成功执行
  - 用例：
    - insmod modulename.ko

# Linux内核编程

- 内核模块操作命令
  - rmmod
    - 该命令卸载正在内存中运行的指定内核模块
  - 用例：
    - rmmod modulename

# Linux内核编程

- 内核模块操作命令
  - modinfo
    - 该命令用于查看内核模块文件中包含的模块信息
  - 用例：
    - modinfo modulename.ko

# Linux内核编程

- 编写内核模块
  - 修改Hello World代码内核配置文件指定该配置为tristate配置选项
  - 使用make menuconfig命令配置Hello World代码被编译为内核模块
  - 使用make zImage命令编译不包含Hello World代码的新内核
  - 使用make modules命令将Hello World代码编译为内核模块

# Linux内核编程

- 编写内核模块
  - 烧写新编译的不包含Hello World代码的内核镜像文件并引导新内核运行
  - 使用make module_install INSTALL_MOD_PATH=/opt/real6410/kernel/linux-2.6.28.6/_install命令安装所有内核模块
  - 进入_install目录，打包lib目录
  - 将lib目录打包文件上传到ARM Linux系统

# Linux内核编程

- 编写内核模块
  - 将lib目录打包文件解压缩到当前ARM Linux系统根目录
  - 使用modprobe/insmod/rmmod命令加载/卸载Hello World内核模块

# NFS服务器配置

- NFS服务器软件包安装
  - rpm -qa | grep nfs

```
[root@Fedora6 ~]# rpm -qa | grep nfs
system-config-nfs-1.3.19-1.1
nfs-utils-lib-1.0.8-7.2
nfs-utils-1.0.9-8.fc6
```

# NFS服务器配置

- NFS服务配置修改
  - vim /etc/exports

```
/opt/nfsroot *(rw,sync,no_root_squash)
```

- 图形界面配置方法
  - 图形界面：System -> Administration -> Server Settings -> NFS

# NFS服务器配置

- NFS服务启动
  - service nfs restart
  - /etc/init.d/nfs restart
  - chkconfig --level 5 nfs [on/off]
  - setup
  - 图形界面：System -> Administration -> Server Settings -> Services

# NFS服务器配置

- NFS服务测试
  - 创建挂载点
    - mkdir –p /mnt/nfs
  - 挂载nfs服务
    - mount –t nfs [本机IP]:/opt/nfsroot /mnt/nfs

# 配置内核通过NFS服务加载文件系统

- 解压缩光盘中真实文件系统文件qtopia.tar.gz到NFS服务器目录
  - tar –xvf qtopia.tar.gz –C /opt/nfsroot
- 从Nand Flash启动目标板进入u-boot
- 设置bootargs环境变量
  - setenv bootargs noinitrd root=/dev/nfs console=ttySAC0 init=/linuxrc nfsroot=192.168.1.178:/opt/nfsroot ip=192.168.1.20:192.168.1.178:192.168.1.1:255.255.255.0::eth0:on video=fb:WX4300F
- 保存环境变量
  - saveenv

# Linux内核编程

- 模块许可证声明
  - 模块许可证(license)声明描述内核模块的许可权限
  - 必须为内核模块指定许可证声明，否则，在模块加载时，会收到内核被污染(kernel tainted)的警告
  - 声明方法：
    - MODULE_LICENSE(“有效的许可证”);
  - 大多数情况下，内核模块遵循GPL兼容许可权

# Linux内核编程

- 模块许可证声明
  - 有效的许可证
    - "GPL"
    - "GPL V2"
    - "GPL and additional rights"
    - "Dual BSD/GPL"
    - "Dual MIT/GPL"
    - "Dual MPL/GPL"
    - "Proprietary" (非自由软件许可证)

# Linux内核编程

- 模块许可证声明
  - Linux 2.6内核模块最常见的许可证：
    - MODULE_LICENSE("Dual BSD/GPL");

# Linux内核编程

- 模块信息声明
  - 模块信息声明为内核模块可选内容
  - 常见模块信息声明：
    - MODULE_AUTHOR("模块作者");
    - MODULE_DESCRIPTION("模块描述内容");
    - MODULE_VERSION("模块版本号");
    - MODULE_ALIAS("模块别名");
    - MODULE_DEVICE_TABLE("设备类型", "设备类型表");
      // 告知用户空间该内核模块支持哪些设备
  - 模块许可证声明和模块信息声明内容可用modinfo命令查看

# Linux内核编程

- 内核导出符号
  - 内核符号表
    - __ksymtab和__ksymtab_strings数据区段
    - 在arch/arm/kernel/vmlinux.lds中为这两个数据区段保留空间

# Linux内核编程

- 内核导出符号
  - 内核符号

```
struct kernel_symbol
{
   unsinged long value; // 记录符号地址
   const char *name; // 记录符号名称
}
```

# Linux内核编程

- 内核导出符号
  - 允许内核模块导出模块中的函数或变量，供其它内核模块引用
  - 内核导出符号声明方法：
    - EXPORT_SYMBOL("符号名称");
    - EXPORT_SYMBOL_GPL("符号名称");
  - EXPORT_SYMBOL_GPL导出的符号只能被拥有GPL许可证支持的内核模块引用

# Linux内核编程

- 内核模块编译
  - 直接编译进内核
  - 在内核目录编译成内核模块
  - 直接在内核模块源代码目录编译成内核模块

# Linux内核编程

- 内核模块编译
  - obj-m指定需要编译的内核模块
  - 通过在内核模块源代码目录编写Makefile直接在内核源代码目录编译成内核模块
  - Makefile内容
    - obj-m := modulename.ko
  - 编译命令
    - make –C /opt/real6410/kernel/linux-2.6.28.6 M=$PMD modules

# Linux内核编程

- 内核模块编译
  - Makefile改进
    - 增加MODNAME变量
      - MODNAME = modulename
    - 增加KERNELDIR变量
      - KERNELDIR ?= /opt/real6410/kernel/linux-2.6.28.6
    - 添加默认伪目标
      - default:
    - 添加make命令
      - $(MAKE) –C $(KERNELDIR) M=$(PWD) modules

# Linux内核编程

- 内核模块编译
  - Makefile改进
    - 增加对编译不同体系结构目标内核模块的支持
      - ifeq ($(PLATFORM), TARGETPLATFORM)
      - KERNELDIR ?= PLATFORMKERNELDIR1
      - else
      - KERNELDIR ?= PLATFORMKERNELDIR2
      - endif
    - 增加垃圾清理伪目标
      - clean:
      - @rm –rf *.o *.ko *.mod.* .$(MODNAME).* .tmp* module* Module*

# Linux内核编程

- 内核printk函数
  - 内核printk和用户空间printf异同
    - 相同：输出信息
    - 不同：
      - printk只能在内核中使用，printf只能在应用程序中使用
      - printk允许指定输出信息严重程度，通过附加不同的“优先级”来对消息进行分类

# Linux内核编程

- 内核printk函数
  - printk优先级(按递减顺序)
    - KERN_EMERG "<0>" 用于处理紧急消息，通常是系统崩溃前的消息
    - KERN_ALERT "<1>" 需要立即处理的消息
    - KERN_CRIT "<2>" 严重情况
    - KERN_ERR "<3>" 错误情况
    - KERN_WARNING "<4>" 有问题的情况
    - KERN_NOTICE "<5>" 正常情况，但是仍然需要注意
    - KERN_INFO "<6>" 信息型消息
    - KERN_DEBUG "<7>" 用作调试信息

# Linux内核编程

- 内核printk函数
  - printk默认值
    - DEFAULT_MESSAGE_LOGLEVEL 4 //警告
      - 如果不指定prink优先级，将使用该默认值
    - MINIUM_CONSOLE_LOGLEVEL 1
      - 控制台输出信息最高优先级
    - DEFAULT_CONSOLE_LOGLEVEL 7
      - 控制台输出信息最小优先级
    - 控制台实际输出比设定优先级更高的信息

# Linux内核编程

- 内核printk函数
  - 控制台printk输出信息设定

```
int console_printk[4] = {
    DEFAULT_CONSOLE_LOGLEVEL,
    DEFAULT_MESSAGE_LOGLEVEL,
    MINIMUM_CONSOLE_LOGLEVEL,
    DEFAULT_CONSOLE_LOGLEVEL,
};
```

# Linux内核编程

- 模块参数
  - 模块参数声明
    - module_param(name, type, perm);
      - name: 参数名称
      - type: 参数数据类型
      - perm: 用户对模块参数的操作权限

# Linux内核编程

- 模块参数
  - 模块数组参数声明
    - module_param_array(name, type, nump, perm);
      - name: 数组参数名称
      - type: 数组参数数据类型
      - nump: 如果数组参数在加载时设置，该值为加载时设置的数据个数，不允许传递比数组允许个数更多的值
      - perm: 用户对模块参数的操作权限

# Linux内核编程

- 模块参数
  - 模块参数描述声明
    - MODULE_PARM_DESC(_parm, desc);
      - _parm: 待增加描述内容的模块参数
      - desc: 对模块参数的描述说明
    - 模块参数描述说明内容可以用modinfo查看

# Linux内核编程

- 模块参数
  - 模块参数支持的数据类型
    - bool 布尔型
    - invbool 布尔型反值
    - charp 字符指针
    - short 短整型
    - ushort 无符号短整型
    - int 整型
    - uint 无符号整型
    - long 长整型
    - ulong 无符号长整型

# Linux内核编程

- 模块参数
  - 模块参数用户操作权限
    - 应该使用<linux/stat.h>中定义的权限
      - S_I[R/W/X]USR 用户读、写、执行权限
      - S_IRWXU 用户读、写、执行权限
      - S_I[R/W/X]GRP 同组用户读、写、执行权限
      - S_IRWXG同组用户读、写、执行权限
      - S_I[R/W/X]OTH其他用户读、写、执行权限
      - S_IRWXO 其他用户读、写、执行权限
      - …

# Linux内核编程

- 模块参数
  - 模块参数对应文件系统位置
    - /sys/module/modulename/parameters目录

# Linux内核编程

- 模块参数
  - 模块参数赋值
    - 定义变量时初始化
    - 在模块加载时初始化
    - 直接修改模块参数文件内容

# Linux内核编程

- Linux内存管理
  - 内存是Linux内核管理的最重要资源之一
  - 内存管理子系统是Linux操作系统内核组成部分的最重要部分之一
  - Linux内存最小管理单位为页(page)，通常一页为4KB
  - Linux系统中，在初始化时，内核为每个物理内存页建立一个page的管理结构，操作物理内存时实际上就是操作page页

# Linux内核编程

- Linux内存管理
  - 地址
    - 物理地址
      - 出现在CPU地址总线上的寻址物理内存的地址信号，是地址变换的最终结果
    - 线性地址(虚拟地址)
      - 在32位CPU架构下，可以表示4G的地址空间，也就是0x00000000 - 0xFFFFFFFF
    - 逻辑地址
      - 实际上是一个相对地址，是程序代码经过编译之后在汇编程序中出现的地址

# Linux内核编程

- Linux内存管理
  - Linux内核地址转换
    - 出现在机器语言指令(程序编译后得到的二进制机器码序列)中的内存地址都是逻辑地址
    - 逻辑地址必须被转换为线性地址
    - MMU将线性地址转换成物理地址，最终实现对应物理内存的访问
  - 在Linux系统中，逻辑地址和线性地址(虚拟地址)是一致的

# Linux内核编程

- Linux内存管理
  - 段式、页式内存管理
    - 段式内存管理
      - 在X86体系结构CPU中采用段式内存管理方式
      - 在Linux系统中，避开了段式内存管理机制
    - 页式内存管理
      - 在大多数嵌入式RISC体系结构CPU中采用页式内存管理方式
      - 在Linux系统中，完全使用分页机制来管理线性地址

# Linux内核编程

- Linux内存管理
  - 页式内存管理
    - 页大小: $2^{12}$ 字节(4KB)
    - 页表索引: $2^{10}$ 项(1K)
    - 页目录索引: $2^{10}$ 项(1K)
    - 采用两级页式内存管理方式能管理的最大内存大小: $2^{32}$ = 4G

# Linux内核编程

- Linux内存管理
  - Linux页式内存管理
    - 在Linux 2.6.28.6中统一采用四级页管理结构
      - pte: 页表(Page Table Entry)
      - pmd: 页中间目录(Page Middle Directory)
      - pud: 页上级目录(Page Upper Directory)
      - pgd: 页全局目录(Page Global Directory)

# Linux内核编程

- 进程地址空间
  - 虚拟内存的好处
    - 每个进程都有独立的进程地址空间，大小为3GB
    - 用户看到的和接触的都是虚拟地址，无法看到真实的物理地址
    - 实现操作系统自我保护
    - 用户程序可以使用到比实际物理内存更大的空间

# Linux内核编程

- 进程地址空间
  - 进程空间划分
    - 用户空间占据0x00000000 - 0xBFFFFFFF，大小为3GB
    - 内核空间占据0xC0000000 - 0xFFFFFFFF，大小为1GB
    - 用户进程通常只能访问进程空间中的用户空间，不能访问内核空间，例外情况：用户进程通过系统调用访问内核空间
    - 每个进程用户空间完全独立、互不相干

# Linux内核编程

- 进程地址空间
  - 内存分配
    - 实际的物理内存仅当进程真实访问新获取的虚拟地址时，才会由“请页机制”产生“缺页异常”，从而进入分配内存的程序
    - “缺页异常”是虚拟内存机制的基本保证-----由它告诉内核为进程分配物理页，并建立页表。这样，虚拟地址才真实地映射了物理地址

# Linux内核编程

- 进程地址空间
  - 用户空间内存分配
    - malloc/free: 按字节分配内存
    - valloc/free: 分配的内存按页对齐

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - kmalloc/kfree: 分配的内存物理上连续，只能在低端内存分配
    - get_zeroed_page/free_page: 分配一个页面，并且该页面内容被清零，只能在低端内存分配
    - __get_free_pages/free_pages: 分配指定页数的低端内存，不能从高端内存分配

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - alloc_pages/__free_pages: 分配指定页数的内存，可以从高端内存，也可以从低端内存分配
    - vmalloc/vfree: 分配的内存在内核空间中连续，物理上无需连续。vmalloc由于不需要物理上也连续，所以性能很差，一般只有在必须申请大块内存时才使用，例如动态插入模块时
    - 对于申请的内存必须释放，否则将导致系统错误

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - kmalloc函数/kfree函数
      - void *kmalloc(size_t size, gfp_t flags);
        - » size: 待分配内存大小(按字节计)
        - » flags: 分配标志，用于控制kmalloc行为
        - » 返回: 分配到的内核虚拟地址，失败返回NULL
      - void kfree(const void *objp)
        - » objp: 由kmalloc返回的内核虚拟地址

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - get_zeroed_page函数/free_page宏
      - unsigned long get_zeroed_page(gfp_t gfp_mask);
        - » gfp_mask: 分配标志，用于控制kmalloc行为
        - » 返回: 指向分配到的已经被清零的内存页面第一个字节的指针，失败返回NULL
      - void free_pages(unsigned long addr);
        - » addr: 由get_zeroed_page返回的内核虚拟地址

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - __get_free_pages函数/free_pages函数
      - unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
        - » gfp_mask:分配标志，用于控制__get_free_pages行为
        - » order: 请求或释放的页数的2的幂，例如：操作1页该值为0，操作16页该值为4。如果该值太大会导致分配失败，该值允许的最大值依赖于体系结构
        - » 返回: 指向分配到的连续内存区第一个字节的指针，失败返回NULL

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - __get_free_pages函数/free_pages函数
      - void free_pages(unsigned long addr, unsigned int order);
        - » addr: 由__get_free_pages返回的内核虚拟地址
        - » order: __get_free_pages分配内存时使用的值
      - int get_order(unsigned long size);
        - » size: 需要计算order值的大小(按字节计算)
        - » 返回: __ get_free_pages等函数需要的order值

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - alloc_pages宏/__free_pages函数
      - struct page *alloc_pages_node(gfp_t gfp_mask, unsigned int order);
        - » gfp_mask:分配标志，用于控制__get_free_pages行为
        - » order: 请求或释放的页数的2的幂，例如：操作1页该值为0，操作16页该值为4。如果该值太大会导致分配失败，该值允许的最大值依赖于体系结构
        - » 返回: 指向分配到的物理页面中的第一个页的struct page指针，失败返回NULL

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - alloc_pages宏/__free_pages函数
      - void __free_pages(struct page *page, unsigned int order);
        - » page: 由alloc_pages返回的指向第一个页的struct page指针
        - » order: alloc_pages分配内存时使用的值

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - vmalloc函数/vfree函数
      - void *vmalloc(unsigned long size);
        - » size: 待分配的内存大小，自动按页对齐
        - » 返回: 分配到的内核虚拟地址，失败返回NULL
      - void vfree(const void *addr);
        - » addr: 由vmalloc返回的内核虚拟地址

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - 内核内存分配标志
      - GFP_KERNEL: 表示该次内存分配由内核态进程调用，这是内核内存的正常分配，该分配方式最常用。如果空闲空间不够，该次分配将使得进程进入睡眠，等待空闲页出现
      - GFP_ATOMIC: 用于分配请求不是来自于进程上下文，而是来自于中断、任务队列处理、内核定时器等中断上下文的情况，此时不能进入睡眠

# Linux内核编程

- 进程地址空间
  - 内核空间内存分配
    - 内核内存分配标志
      - __GFP_DMA: 用于分配用于DMA(Direct memory access-直接内存访问)功能的内存区(通常物理地址在16M以下)
      - __GFP_HIGHMEM: 用于分配的内存可以位于高端内存的情况

# Linux内核编程

- 内核地址空间
  - 内核空间由内核负责映射，是固定的，不随进程改变

# Linux内核编程

- 内核地址空间
  - 内核地址空间划分

| 直接内存映射区 (Direct Memory Region) | 8MB隔离区 | 动态内存映射区 (Vmalloc Region) | 8KB隔离区 | 永久内存映射区 (PKMap Region) | 固定映射区 (Fixing Mapping Region) | 4KB隔离区 |
|---|---|---|---|---|---|---|
| 最大896MB | | 最大120MB | | 4MB | 4MB | |

# Linux内核编程

- 内核地址空间
  - 直接内存映射区
    - 从3G开始，最大896M的线性地址空间，称为直接内存映射区，该区域的线性地址和物理地址之间存在线性转换关系：
    - 线性地址 = 3G +（物理地址 - 物理地址起始值）
    - high_memory是具体物理内存的上限对应的虚拟地址

# Linux内核编程

- 内核地址空间
  - 动态内存映射区
    - 动态映射区由内核函数vmalloc来分配，特点是线性空间连续、但是对应的物理空间不一定连续。vmalloc函数分配的线性地址所对应的物理页可能位于低端内存，也可能位于高端内存
    - 如果物理内存小于896M，则：
      - hign_memory = 0xC0000000 + 物理内存大小
    - 如果物理内存大于或等于896M，则：
      - high_memory = 0xC0000000 + 896M
    - 该区域从(high_memory+8M)内核地址开始分配

# Linux内核编程

- 内核地址空间
  - 永久内存映射区
    - 该区域可访问高端内存
      - 使用alloc_page(__GFP_HIGHMEM)分配高端内存页
      - 使用kmap函数将分配到的高端内存映射到该区域

# Linux内核编程

- 内核地址空间
  - 固定映射区
    - PKMAP上面，有4M的线性空间，被称为固定映射区，它和4G顶端只有4K的隔离带。该区域中每个地址项都服务于特殊用途，比如：ACPI_BASE等

# Linux内核编程

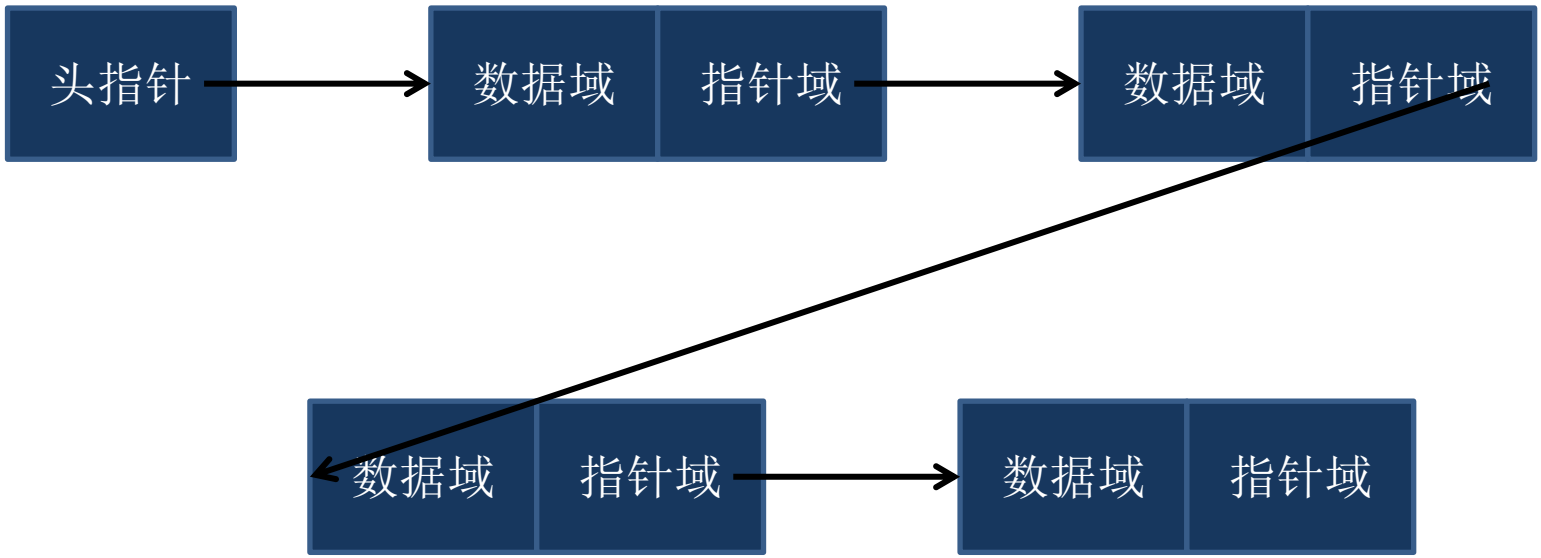
- 内核链表
  - 链表定义
    - 链表是一种常用数据结构，它通过指针将一系列数据节点链接成一条数据链
  - 链表与数组比较
    - 相对于数组，链表具有更好的动态性，建立链表时无需预先知道数据总量，可以随机分配空间，可以高效地在链表中任意位置实时插入或删除数据

# Linux内核编程

- 内核链表
  - 链表开销
    - 主要开销是访问的顺序性和组织数据链的空间损失
  - 链表构成
    - 通常链表数据结构至少包含两个域：数据域与指针域，前者用于存储数据，后者用于建立与下一个节点的联系
  - 链表分类
    - 按照指针域的组织和各个节点之间的关系，通常将链表分为单链表、双链表、循环链表等多种类型

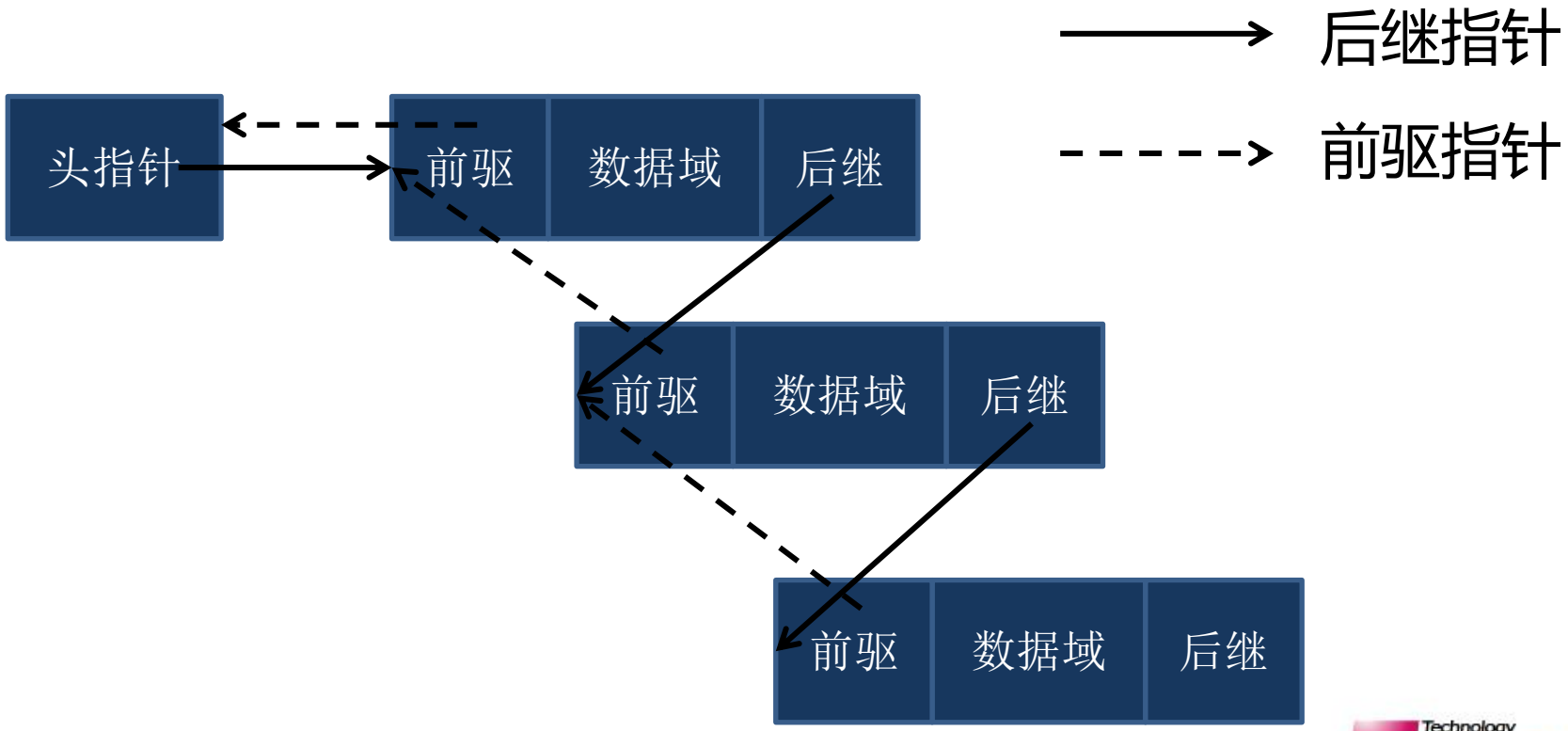
# Linux内核编程

- 内核链表
  - 单向链表



# Linux内核编程

- 内核链表
  - 双向链表



# Linux内核编程

- 内核链表
  - 内核链表定义
    - 在<linux/list.h>中定义

```
struct list_head {
        struct list_head *next, *prev;
};
```

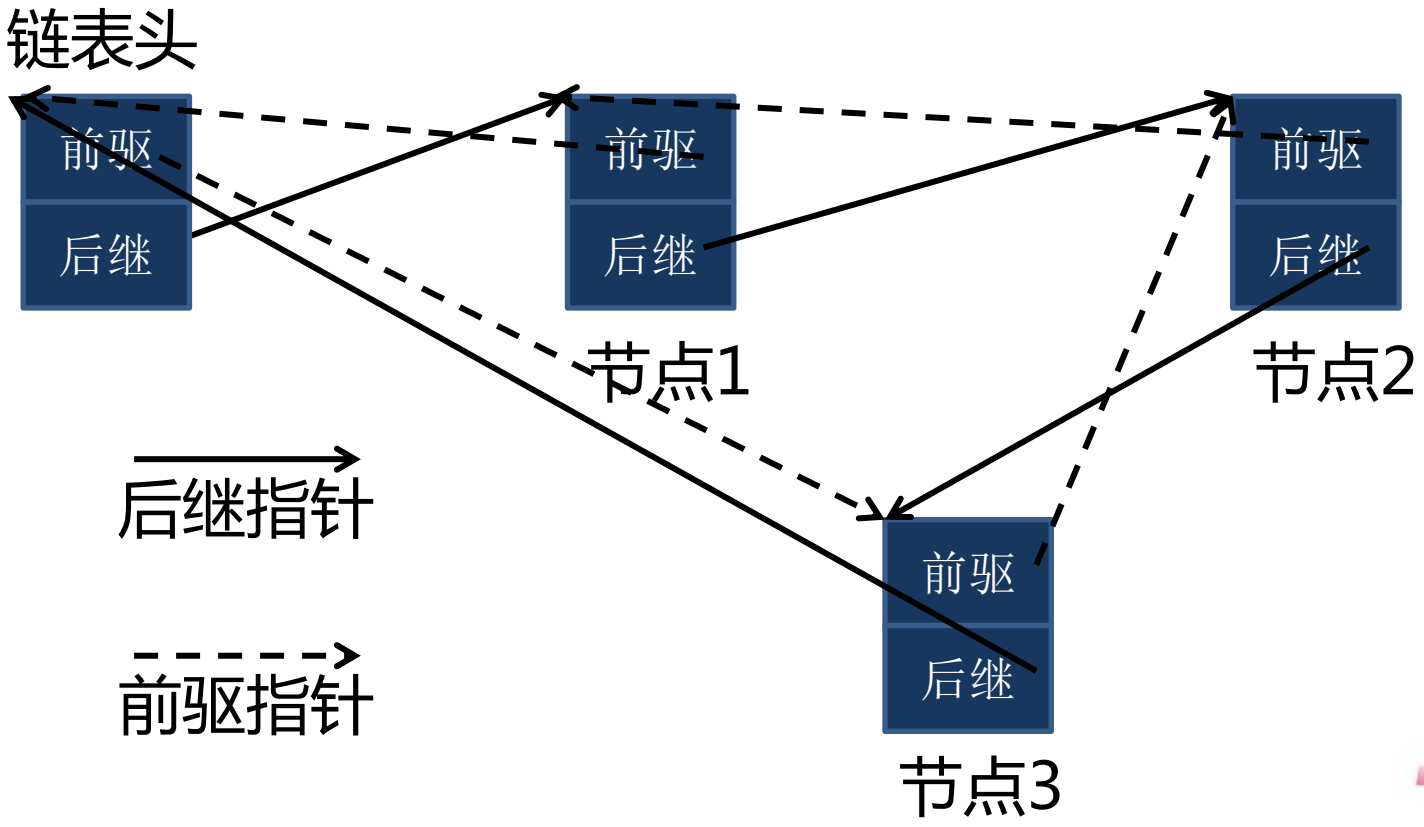
    - 在list_head结构中包含两个指向list_head结构的指针next和prev，所以，内核链表具备双链表功能
    - 在实际使用中，它通常被组织成双向循环链表

# Linux内核编程

- 内核链表
  - 内核链表和普通链表比较
    - 内核链表结构体不包含数据域，只包含维护链表的指针域
    - 内核链表具有通用性，和具体数据结构无关
    - 内核链表常被用作双向循环链表
    - 内核链表被包含在其它数据结构体中使用

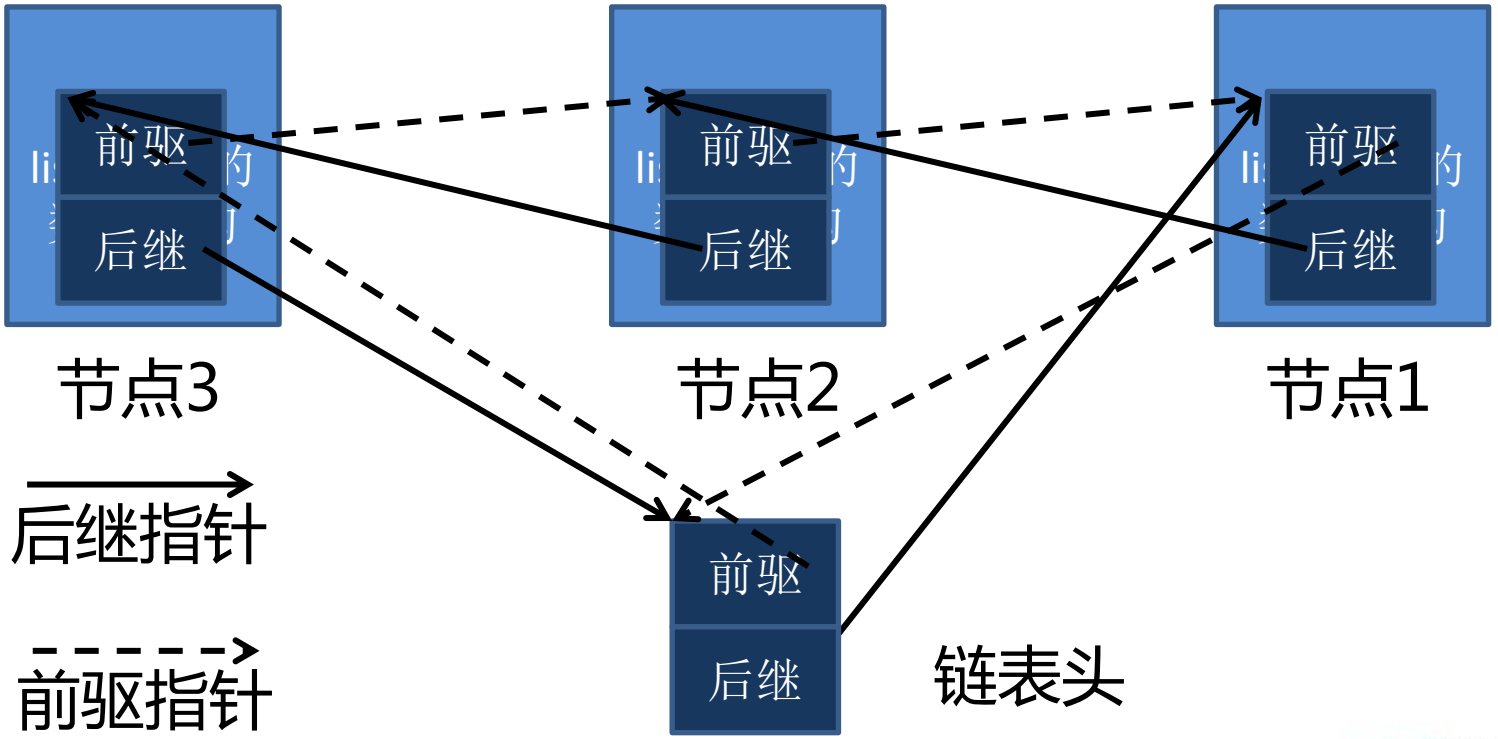
# Linux内核编程

- 内核链表
  - 内核链表和普通链表比较



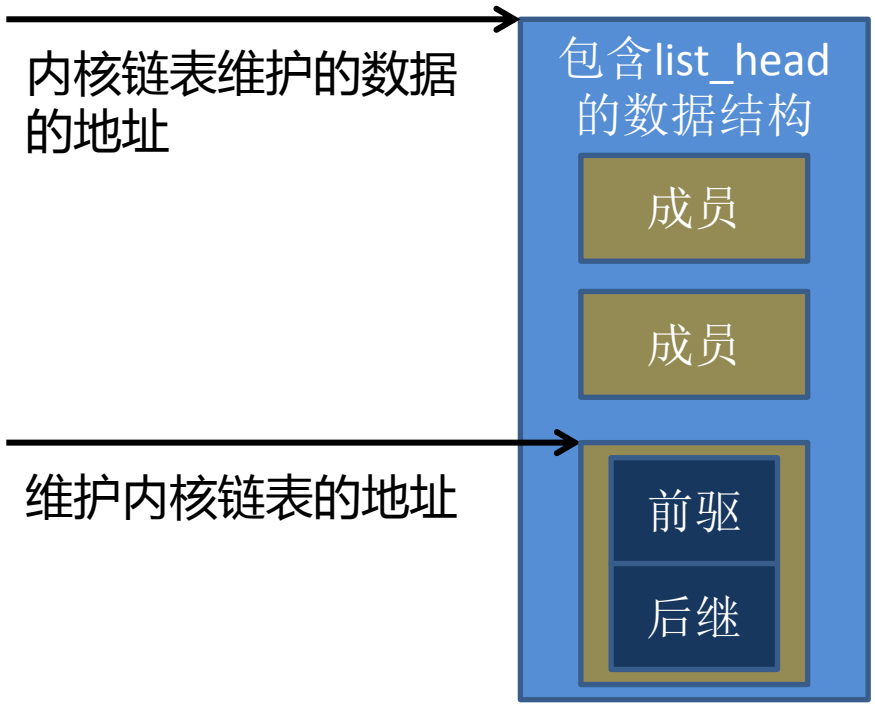
# Linux内核编程

- 内核链表
  - 内核链表和普通链表比较



# Linux内核编程

- 内核链表
  - 内核链表和普通链表比较



# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 初始化链表头 INIT_LIST_HEAD函数
    - 插入节点 list_add函数
    - 删除节点 list_del函数
    - 提取数据结构 list_entry宏
    - 遍历链表 list_for_each宏

# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 初始化链表头 INIT_LIST_HEAD函数
      - void INIT_LIST_HEAD(struct list_head *list);
        - » list: 待初始化链表头

# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 插入节点 list_add函数
      - void list_add(struct list_head *new, struct list_head *head);
      - void list_add_tail(struct list_head *new, struct list_head *head);
        - » new: 待插入到链表的新节点
        - » head: 待插入到链表的链表头

# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 删除节点 list_del函数
      - void list_del(struct list_head *entry);
        - » entry: 待删除的节点

# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 提取数据结构 list_entry宏
      - #define list_entry(ptr, type, member) container_of(ptr, type, member)
        - » ptr: 当前链表节点指针
        - » type: 包含该链表的数据结构体类型
        - » member: 在数据结构体类型中的list_head成员名称
        - » 返回: 获取的数据结构体指针
    - 实际是通过已知数据结构体中链表节点指针ptr，获取包含该链表节点的数据结构体指针

# Linux内核编程

- 内核链表
  - 内核链表主要操作
    - 遍历链表 list_for_each宏
      - #define list_for_each(pos, head) for (pos = (head)->next; prefetch(pos->next), pos != (head); pos = pos->next)
      - #define list_for_each_safe(pos, n, head) for (pos = (head)->next, n = pos->next; pos != (head); pos = n, n = pos->next)
        - » pos: list_head指针类型的循环变量
        - » n: list_head指针类型的循环变量
        - » head: 待遍历链表的链表头

# Linux内核编程

- 内核链表
  - klist链表
    - klist链表是对list_head的扩展
    - klist数据类型定义
      - 定义在<linux/klist.h>中

```
struct klist {
    spinlock_t k_lock; // klist操作保护锁
    struct list_head k_list; // klist链表
    void (*get)(struct klist_node *); // 获取klist_node节点
    void (*put)(struct klist_node *); //添加klist_node节点
} __attribute__ ((aligned (4)));
```

# Linux内核编程

- 内核链表
  - klist链表
    - klist数据类型定义
      - 定义在<linux/klist.h>中

```
struct klist_node {
    void *n_klist; /* never access directly */
    struct list_head n_node; // 节点所属链表入口
    struct kref n_ref; // 节点引用计数
    struct completion n_removed;
};
```

# Linux内核编程

- 内核定时器
  - 时钟中断
    - 由系统的定时硬件以周期性的时间间隔发生，这个间隔(也就是频率)由内核根据常数HZ来确定
  - HZ常数
    - 它是一个与体系结构无关的常数，可以配置50-1200之间，可以在内核中配置
  - tick
    - 它是HZ的倒数，也就是每发生一次硬件定时器中断的时间间隔。如HZ为200，tick为5毫秒

# Linux内核编程

- 内核定时器
  - jiffies核心变数
    - 它是Linux核心变数(32位变数，unsigned long)，它被用记录自开机以来，已经过多少个tick。每发生一次硬件定时器中断，jiffies变数会被加1

# Linux内核编程

- 内核定时器
  - 定时器的作用
    - 内核定时器由用户控制某个函数(定时器函数)在特定的未来某个时刻执行
    - 内核定时器注册的处理函数只执行一次(不是循环执行的)

# Linux内核编程

- 内核定时器
  - 内核定时器定义
    - 在<linux/timer.h>中定义

```
struct timer_list
{
    struct list_head entry; // 内核使用
    unsigned long expires; // 超时时候jiffies的值
    void (*function)(unsigned long); // 超时处理函数
    unsigned long data; // 内核调用超时处理函数时传递给它的参数
    struct tvec_base *base; // 内核使用
};
```

# Linux内核编程

- 内核定时器
  - 内核定时器操作
    - 静态初始化
      - TIMER_INITIALIZER(_function, _expires, _data)
        - » _function: 定时器处理函数
        - » _expires: 定时器超时jiffies值
        - » _data: 传递给定时器处理函数的参数
    - 定义定时器
      - DEFINE_TIMER(_name, _function, _expires, _data);
        - » _name: 待定义的内核定时器变量名称

# Linux内核编程

- 内核定时器
  - 内核定时器操作
    - 动态初始化
      - void init_timer(struct timer_list *timer);
        - » timer: 待初始化的内核定时器
    - 添加定时器
      - void add_timer(struct timer_list *timer);
        - » timer: 待添加到内核的内核定时器
    - 删除定时器
      - 该函数是在定时器超时前将定时器删除
      - 实际上，在定时器超时后，系统会自动将其删除
      - void del_timer(struct timer_list *timer);
        - » timer: 待删除的内核定时器

# Linux内核编程

- 内核定时器
  - 内核定时器操作
    - 修改定时器
      - 该函数实际相当于执行del_timer()，timer->expires=expires, add_timer()三个步骤
      - int mod_timer(struct timer_list *timer, unsigned long expires);
        - » timer: 待修改的内核定时器
        - » expires: 待修改内核定时器的新超时值
        - » 返回: 返回0表示定时器处于不活动状态，返回1表示定时器处于活动状态

# Linux内核编程

- 内核定时器
  - 内核代码中的延时函数
    - 短延时
    - 毫秒延时
    - 长延时
    - 睡眠延时

# Linux内核编程

- 内核定时器
  - 短延时
    - 该类型的延时都是忙等待
    - 纳秒级延时
      - void ndelay(unsigned long nsecs);
    - 微秒级延时
      - void udelay(unsigned long usecs);
    - 毫秒级延时
      - void mdelay(unsigned long msecs);

# Linux内核编程

- 内核定时器
  - 毫秒延时
    - 在内核中，最好不要直接使用mdelay()函数，这将无谓地耗费CPU资源，应该使用下列函数(这些函数会使得调用进程睡眠由函数参数指定的时间)：
    - 毫秒级延时
      - void msleep(unsigned int millisecs);
      - unsigned long msleep_interruptible(unsigned int millisecs);
    - 秒级延时
      - void ssleep(unsigned int seconds);

# Linux内核编程

- 内核定时器
  - 长延时
    - 对于精度要求不高的延时，可以直接比较当前的jiffies和目标jiffies，直到未来的jiffies达到目标jiffies
    - unsigned long time_before(unsigned long source, unsigned long target);
    - unsigned long time_after(unsigned long target, unsigned long source);
    - 例如：

```
unsigned long waittime = jiffies + delay_seconds * HZ;
while( time_before( jiffies, waitime ) );
```

# Linux内核编程

- 内核定时器
  - 睡眠延时
    - 睡眠延时显然是比忙等待更好的方式，睡眠延时在等待的时间到来之前进程处于睡眠状态，CPU资源可以被释放供其它进程使用
    - signed long schedule_timeout(signed long timeout);
      - timeout: 需要睡眠的jiffies计数值，通常用HZ来计算
      - 返回: 返回非0表示超时时间到达返回
      - 返回0表示进入可打断睡眠时被打断返回
    - 在调用schedule_timeout 前应该修改进程状态

# Linux内核编程

- 内核定时器
  - 睡眠延时
    - 对应schedule_timeout有几个扩展函数:
    - signed long schedule_timeout_interruptible(signed long timeout);
    - signed long schedule_timeout_uninterruptible(signed long timeout);
    - signed long schedule_timeout_killable(signed long timeout);

# Linux内核编程

- 内核定时器
  - 睡眠延时
    - 内核中还有两个可以将当前进程添加到等待队列中，从而在等待队列上睡眠的函数。当超时发生时，进程将被唤醒
    - long sleep_on_timeout(wait_queue_head_t *q, long timeout);
    - long interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout);
      - q: 进程被放入的等待队列头
      - timeout: 需要睡眠的jiffies计数值，通常用HZ来计算
      - 返回: 返回非0表示超时时间到达返回
      - 返回0表示进入可打断睡眠时被打断返回

# Linux内核编程

- 进程控制
  - 进程和程序定义
    - 进程: 是一个执行中的程序，是动态的实体
    - 程序: 是存放在存储设备上的一系列代码和数据的可执行镜像，是一个静止的实体

# Linux内核编程

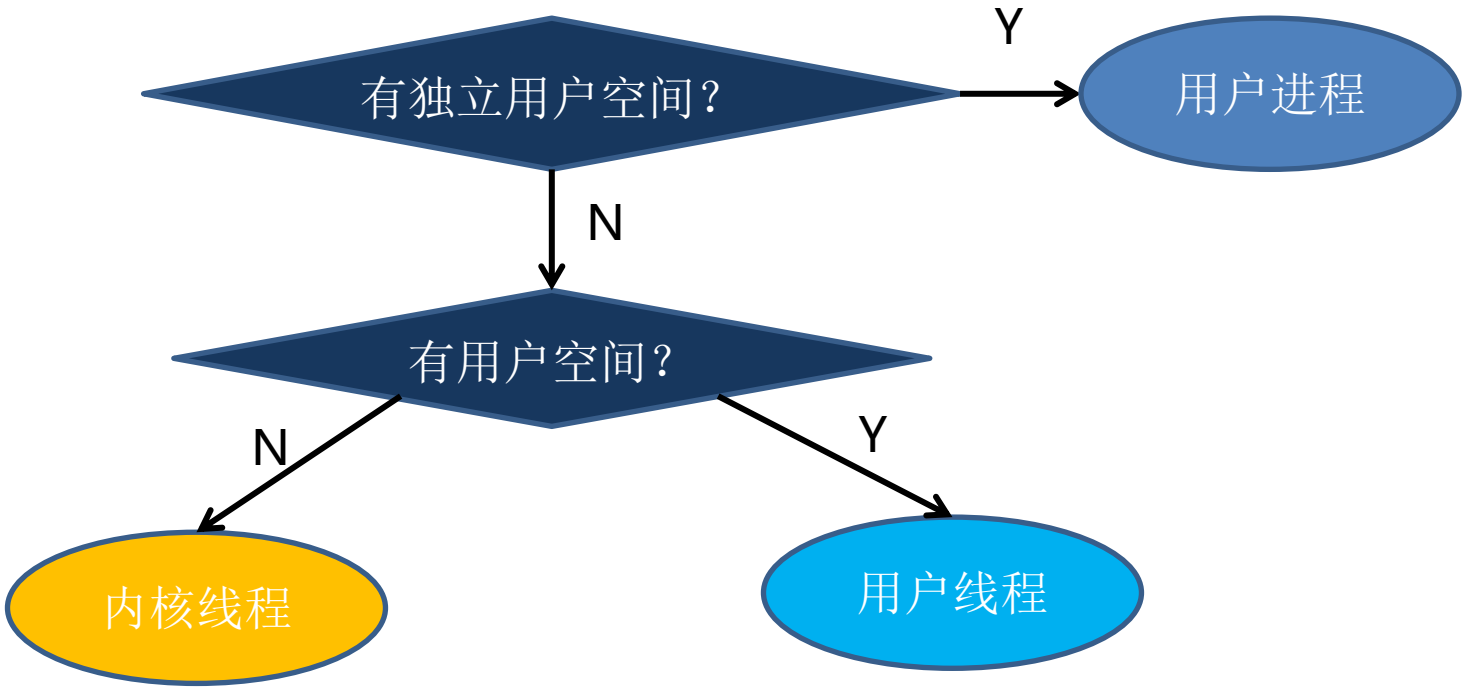
- 进程控制
  - 进程四要素
    - 进程有一段代码供其执行，这段代码可以非当前进程私有，可以是与其它进程共享的代码片段
    - 进程有进程私有的内核空间堆栈
    - 进程在内核中有一个代表它的task_struct数据结构，既通常所说的“进程控制块”，内核通过这个数据结构来组织和调度进程
    - 进程有独立的用户空间

# Linux内核编程

- 进程控制
  - Linux系统中的进程
    - 内核线程(核心进程)
      - 进程描述符、PID、进程正文段、核心堆栈
    - 用户进程
      - 进程描述符、PID、进程正文段、核心堆栈、用户空间的数据段和堆栈
    - 用户线程
      - 进程描述符、PID、进程正文段、核心堆栈、同父进程共享用户空间的数据段和堆栈

# Linux内核编程

- 进程控制
  - Linux系统中的进程



# Linux内核编程

- 进程控制
  - task_struct数据结构
    - 在Linux系统中，线程、进程都通过task_struct数据结构来描述，它包含大量描述线程、进程的信息
    - 在<linux/sched.h>中定义

# Linux内核编程

- 进程控制
  - task_struct数据结构
    - pid_t pid; // 进程号
    - volatile long state; // 进程状态
    - int exit_state; // 进程退出状态
    - struct mm_struct *mm; // 进程用户空间描述指针，对于内核线程该指针为空
    - unsigned int policy; // 进程调度策略

# Linux内核编程

- 进程控制
  - task_struct数据结构
    - int prio; // 进程优先级
    - int static_prio; // 静态优先级
    - struct sched_rt_entity rt; // 进程实时调度实体
    - struct task_struct *real_parent; // 真实的父进程
    - struct task_struct *parent; // 对于信号SIGCHLD, wait4()等报告的接收容器
    - char comm[TASK_COMM_LEN]; // 不包含路径的执行程序的名称

# Linux内核编程

- 进程控制
  - 进程号
    - 当前系统进程号最大为0x8000，即32768
    - 在<linux/threads.h>中定义

# Linux内核编程

- 进程控制
  - 进程状态
    - TASK_RUNNING
      - 进程正在被执行，或已经就绪随时可执行，当进程刚被创建时，处于该状态
    - TASK_INTERRUPTIBLE
      - 处于等待中的进程，待等待条件为真时被唤醒，也可以被信号或者中断唤醒
    - TASK_UNINTERRUPTIBLE
      - 处于等待中的进程，待资源有效时唤醒，但不可以由其它进程通过信号或者中断唤醒
    - TASK_KILLABLE
      - 内核2.6.25后新增的睡眠状态，类似于TASK_INTERRUPTIBLE，可以被致命信号(SIGKILL)唤醒

# Linux内核编程

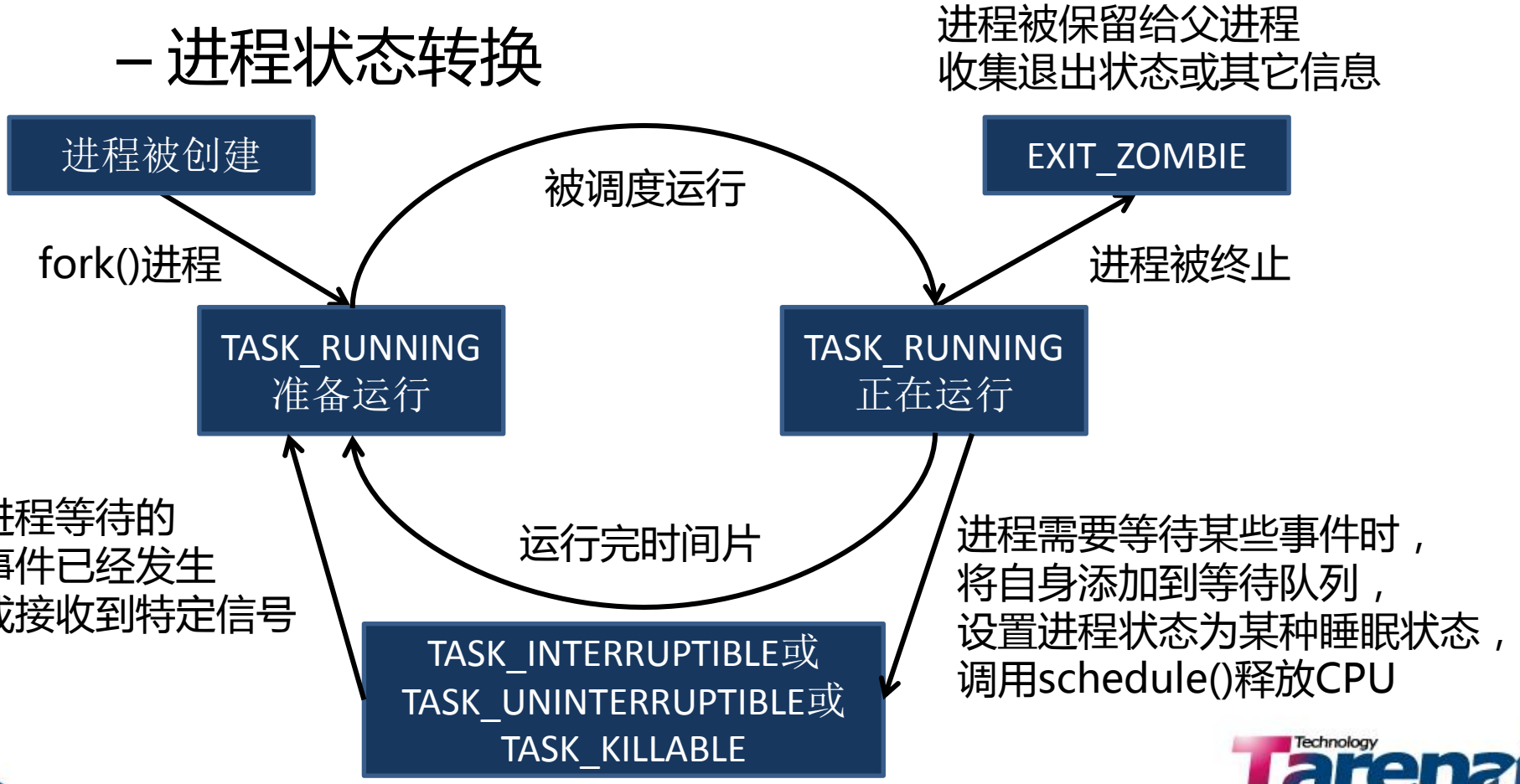
- 进程控制
  - 进程状态
    - TASK_STOPPED
      - 进程暂时中止执行，当接收到SIGSTOP和SIGTSTP等信号时，进程进入该状态，接收到SIGCONT信号后，进程重新回到TASK_RUNNING状态
    - TASK_TRACED
      - 正处于被调试状态的进程
    - TASK_DEAD
      - 进程调用do_exit退出后，处于该状态

# Linux内核编程

- 进程控制
  - 进程退出状态
    - EXIT_ZOMBIE
      - 僵死状态，表示进程的执行被终止，但是，父进程还没有发布waitpid()系统调用来收集有关死亡的进程的信息
    - EXIT_DEAD
      - 僵死撤销状态，表示进程的最终状态，父进程已经使用wait4()或waitpid()系统调用来收集了信息，因此进程将由系统删除

# Linux内核编程

- 进程控制
  - 进程状态转换



# Linux内核编程

- 进程控制
  - 进程优先级
    - 数值越大，进程优先级越低
    - 取值范围：0---(MAX_PRIO - 1) // 139
      - 0---(MAX_RT_PRIO - 1) // 0-99 实时进程
      - MAX_RT_PRIO --- (MAX_PRIO -1) // 100-139 非实时进程
    - 优先级常量
      - #define MAX_USR_RT_PRI0 100
      - #define MAX_RT_PRIO MAX_USR_RT_PRIO
      - #define MAX_PRO (MAX_RT_PRIO + 40)
      - #define DEFAULT_PRIO (MAX_RT_PRIO + 20)

# Linux内核编程

- 进程控制
  - 进程静态优先级
    - 数值越大，进程静态优先级越低
    - 作用: 决定进程初始时间片的大小
    - 不论是实时进程还是非实时进程都一样，只不过实时进程的该值不参与优先级运算

# Linux内核编程

- 进程控制
  - 时间片
    - rt->time_slice成员表示进程运行占用的时间片
    - 进程的缺省时间片与进程静态优先级有关
    - 内核将100-139的优先级映射到200ms-10ms的时间片上，优先级数值越大，分配的时间片越小

# Linux内核编程

- 进程控制
  - current指针
    - 该指针总是指向当前正在运行的进程的task_struct结构体
  - 当进程被创建，为进程申请task_struct(大约1KB)任务结构空间时，系统将连同系统的堆栈空间一起分配，分配该空间时以8KB为单位来分配

# Linux内核编程

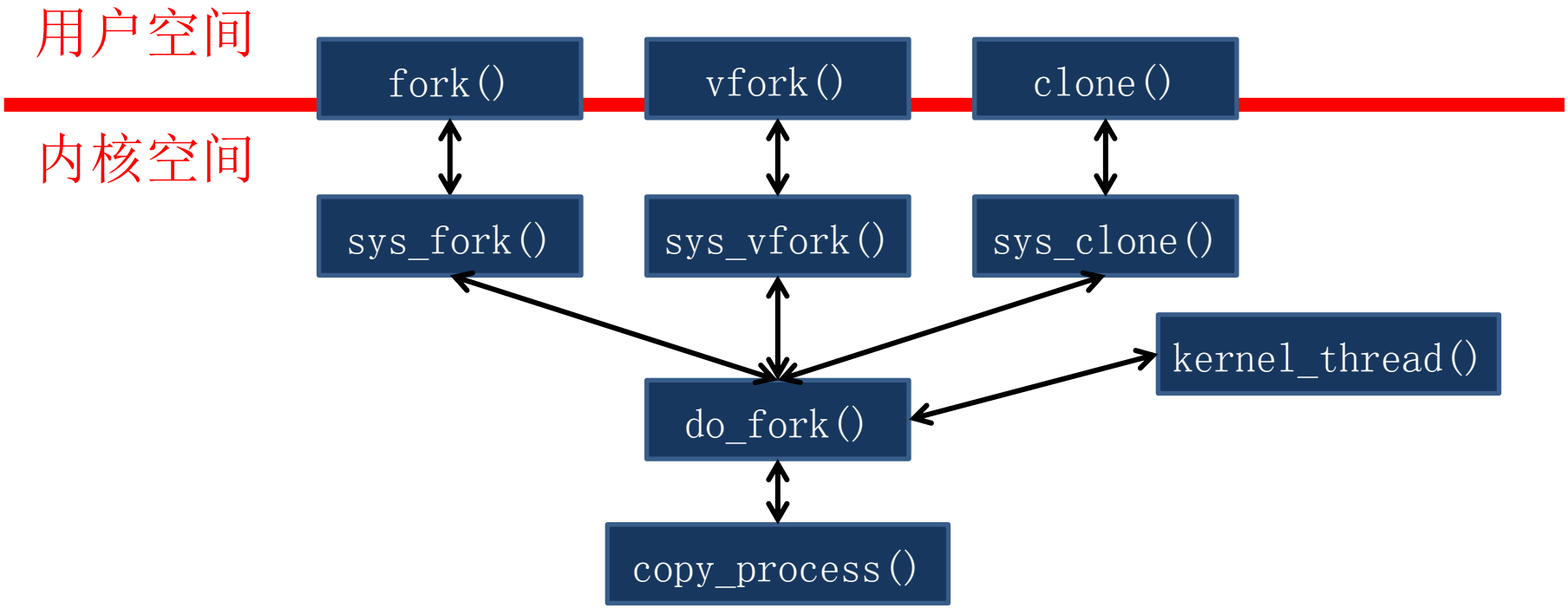
- 进程控制
  - 进程创建过程
    - fork()系统调用
      - 创造的子进程复制了父亲进程的资源，包括内存的内容task_struct内容(2个进程的pid不同)
    - vfork()系统调用
      - 创建出来的不是真正意义上的进程，而是一个线程，缺少进程四要素中的独立用户空间

# Linux内核编程

- 进程控制
  - 进程创建过程
    - clone()系统调用
      - 可以让用户有选择性的继承父进程的资源。用户可以选择像vfork()系统调用一样创建的“子进程”和父进程共享一个虚存空间，从而使创造的是线程；也可以不和父进程共享，甚至可以选择创造出来的进程和父进程不再是父子关系，而是兄弟关系

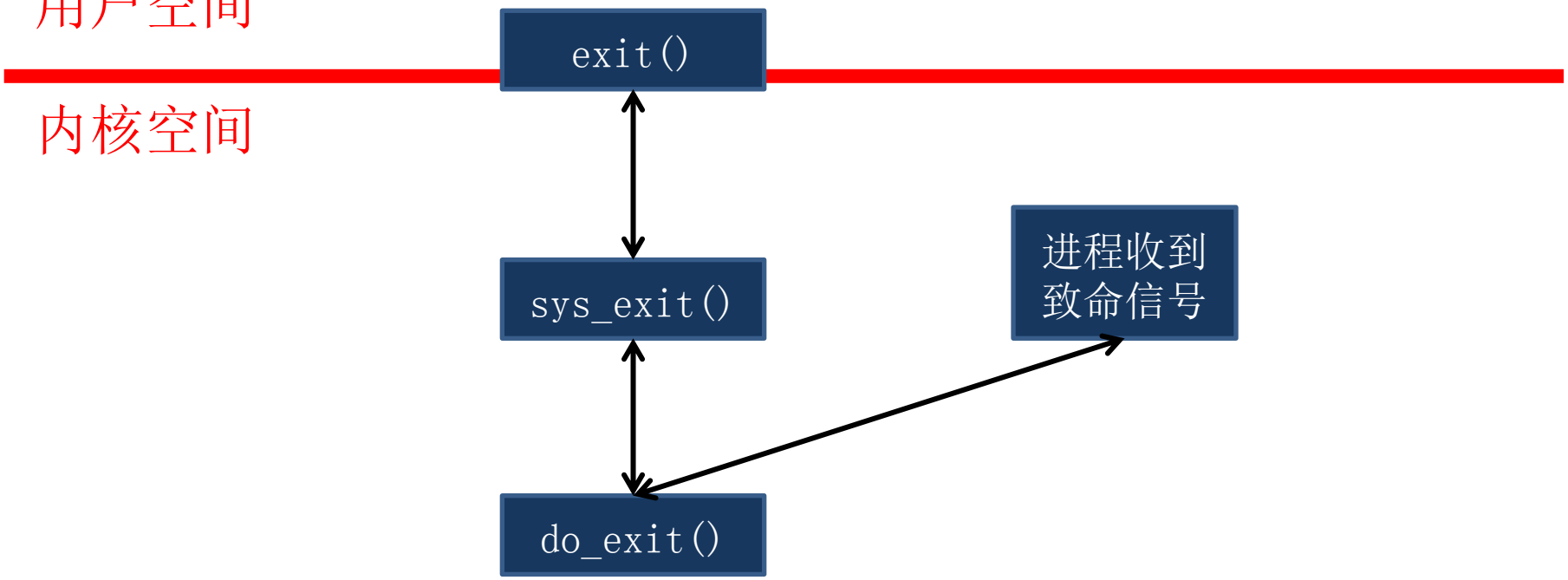
# Linux内核编程

- 进程控制
  - 进程创建过程



# Linux内核编程

- 进程控制
  - 进程销毁过程



# Linux内核编程

- 进程调度
  - 调度定义
    - 从准备就绪的进程中用一定规则选出最适合的一个来执行

# Linux内核编程

- 进程调度
  - 调度策略
    - SCHED_NORMAL
      - 传统的调度策略，适用于交互式的分时应用。一般进程都是此策略
    - SCHED_FIFO
      - 即先来先服务。适用于对时间性要求比较强，而且运行时所用的时间比较短的进程，实时应用程序适合该策略
    - SCHED_RR
      - RR(Round Robin)是轮流的意思，适合程序比较大，每次运行都需要花费很长时间的进程

# Linux内核编程

- 进程调度
  - 调度策略
    - SCHED_BATCH
      - 适用于具有batch风格(批处理)，CPU密集型进程，非交互型，该类进程不抢占另外的进程
    - SCHED_IDLE
      - 适用于优先级非常低的进程，该类进程仅在系统空闲时才会被调度执行

# Linux内核编程

- 进程调度
  - 调度时机
    - 调度在何时发生？也就是schedule()函数应该在何时被调用
    - 调度发生方式：
      - 主动式(自愿式)
      - 被动式(被抢占，非自愿式)

# Linux内核编程

- 进程调度
  - 调度时机
    - 主动式(自愿式)
      - 在内核中
        - » 当进程需要等待资源而暂时停止运行时，通过调用schedule()或者schedule_timeout()自愿放弃CPU，调度其它进程，调度前应该将进程状态置于挂起(睡眠)
        - » 例如：current->state = TASK_INTERRUPTIBLE;

          schedule();
      - 在用户空间
        - » 通过系统调用pause()或nanosleep()而自愿放弃CPU

# Linux内核编程

- 进程调度
  - 调度时机
    - 被动式(被抢占，非自愿式)
      - 被动式可以是用户抢占，也可以是内核抢占

# Linux内核编程

- 进程调度
  - 调度时机
    - 用户抢占
      - 当内核即将返回用户空间的时候，如果need_resched()测试TIF_NEED_RESCHED标志被设置，会导致schedule()被调用，此时就会发生抢占，该抢占称为用户抢占
        - » 当从用户进程进行系统调用并返回时(系统调用在内核空间中完成)
        - » 当从中断处理程序返回时(中断处理在内核空间中完成)
        - » 当从异常处理程序返回时(异常处理在内核空间中完成)

# Linux内核编程

- 进程调度
  - 调度时机
    - 内核抢占
      - 在支持内核抢占的系统中，更高优先级的进程/线程可以抢占正在内核空间运行的低优先级进程/线程

# Linux内核编程

- 进程调度
  - 调度时机
    - 内核抢占
      - 不允许内核抢占的特例：
        - » 内核正进行中断处理。此时进程调度函数schedule()会对此作出判断，如果是在中断中调用，会打印出错信息
        - » 内核正在进行中断上下文的Bottom Half(中断底半部)处理。硬件中断返回前会执行软中断，此时处于中断上下文中
        - » 进程正持有spinlock自旋锁、readlock/writelock读写锁等情况。当持有这些锁时，不应该被抢占，否则由于抢占将导致其它CPU长期不能获得锁而出现死等
        - » 内核正在执行调度程序。抢占的原因就是为了进行新的调度，所以没道理将正在运行中的调度程序抢占掉并再次执行调度程序

# Linux内核编程

- 进程调度
  - 调度时机
    - 内核抢占
      - 为保证Linux内核在以上情况下不会被抢占，抢占式内核使用了一个变量preempt_count，称为内核抢占计数。这个变量被设置在进程的thread_info结构中。每当内核要进入以上几种状态时，变量preempt_count就加1，指示内核不允许抢占。每当内核中以上几种状态退出时，变量preempt_count就减1，同时进行可抢占的判断与调度

# Linux内核编程

- 进程调度
  - 调度时机
    - 内核抢占
      - 内核抢占可能发生的时机
        - » 中断处理程序完成，返回内核空间之前
        - » 当内核代码再一次处于可抢占状态的时候，如解锁及使能软中断等

# Linux内核编程

- 进程调度
  - 调度时机
    - 调度标志 TIF_NEED_RESCHED
      - 内核通过调用need_resched()来判断当前thread_info结构体中的flags标志中的调度标志TIF_NEED_RESCHED标志位来表明是否需要重新执行一次调度
      - 该标志设置的时机
        - » 当某个进程耗尽它的时间片时，会设置这个标志
        - » 当一个优先级更高的进程进入可执行状态的时候，也会设置这个标志

# Linux内核编程

- 进程调度
  - 调度步骤
    - 清理当前运行中的进程
    - 选择下一个要运行的进程
    - 设置新进程的运行环境
    - 进程上下文切换

# Linux内核编程

- 系统调用
  - 系统调用定义
    - 系统调用是Linux内核中设置的一组用于实现各种功能的子程序
    - 在当前Linux 2.6.28.6中大约有360个系统调用
    - 系统调用的C语言定义在arch/arm/include/asm/unistd.h中
    - 系统调用的入口地址跳转表在arch/arm/kernel/calls.S中

# Linux内核编程

- 系统调用
  - 系统调用调用方法
    - 通过系统调用命令来调用
  - 系统调用和普通函数调用的区别
    - 系统调用: 由操作系统内核实现，运行于内核态
    - 普通函数调用: 由函数库或者用户自己提供，运行于用户态
  - 标准LIBC
    - 是对系统调用进行包装和扩展后的标准C语言函数，它与系统调用关系紧密

# Linux内核编程

- 系统调用
  - 系统调用分类
    - 进程控制
    - 文件系统
    - 系统控制
    - 内存管理
    - 网络管理
    - socket控制
    - 用户管理
    - 进程间通信

# Linux内核编程

- 系统调用
  - 系统调用分类
    - 进程间通信
      - 信号
      - 消息
      - 管道
      - 信号量
      - 共享内存

# Linux内核编程

- 文件系统和设备文件系统
  - 在Linux系统中字符设备和块设备都很好体现了“一切都是文件”的设计思想，掌握文件系统和设备文件系统成为设备驱动开发的必须要掌握的基础知识
  - 设备驱动最终通过操作系统的文件系统调用或C库函数中的文件操作函数(本质上也是基于系统调用)被访问
  - 设备驱动不可避免要与文件系统打交道

# Linux内核编程

- 文件系统和设备文件系统
  - 文件操作相关系统调用
    - create/open/close
    - read/write/lseek
    - ioctl
    - select
    - mmap
    - fcntl

# Linux内核编程

- 文件系统和设备文件系统
  - 标准C库函数文件操作相关函数
    - fopen/fclose
    - fread/fwrite/fseek
    - fgetc/fputc/fgets/fputs
    - fscanf/fprintf
    - fgetpos/fsetpos

# Linux内核编程

- 文件系统和设备文件系统
  - Linux文件系统目录结构
    - Linux系统根目录("/"，系统中处于最高一级的目录)
      列出当前系统所有目录入口

# Linux内核编程

- 文件系统和设备文件系统
  - Linux文件系统与设备驱动
    - Linux系统的VFS(Virtual File System，即虚拟文件系统)为应用程序操作文件提供统一的访问方法，应用程序完全不必关心文件存储的具体存储设备及该存储设备被格式化为什么样的具体文件系统类型
    - 根目录下的dev目录中存储着当前系统中的设备文件，应用程序正是通过对这些文件的读写和控制操作来实现访问实际的硬件设备

# Linux内核编程

- 文件系统和设备文件系统
  - Linux系统内核中的文件描述
    - struct file结构体
      - 文件结构体代表一个打开的文件(设备对应设备文件)，系统中每个打开的文件在内核空间都有一个关联的struct file
      - 它由内核在打开文件时创建，并传输给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动代码中，struct file的指针通常被命名为file或filp

# Linux内核编程

- 文件系统和设备文件系统
  - Linux系统内核中的文件描述
    - struct inode结构体
      - VFS inode包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是Linux管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁

# Linux内核编程

- 文件系统和设备文件系统
  - devfs(设备文件系统)
    - devfs由Linux 2.4内核引入
    - devfs允许驱动程序在设备初始化时在/dev目录下创建设备文件，卸载设备时将设备文件删除
    - 在Linux 2.6.13开始的内核中不再支持它

# Linux内核编程

- 文件系统和设备文件系统
  - udev用户工具
    - udev由Linux 2.6内核引入
    - udev是一套工具，用于在系统中添加/删除硬件时处理/dev目录以及所有用户空间的行为

# Linux内核编程

- 文件系统和设备文件系统
  - udev用户工具
    - udev完全在用户态工作，利用设备加入或移除时内核所发送的热插拔(hotplug event)来工作
    - 在热插拔时，设备的详细信息会由内核输出到位于/sys的sysfs文件系统
    - udev的设备命名策略、权限控制和事件处理都是在用户态下完成，它利用sysfs中的信息来进行创建设备文件节点等工作

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 根目录下的proc目录中显示出当前系统中的procfs文件系统中的所有内容，系统初始化时将procfs挂载到该目录
    - procfs文件系统是一个虚拟文件系统，用于内核向用户导出信息，通过它可以在内核空间和用户空间之间进行通信

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 可以通过对procfs文件系统中虚拟文件的读写作为与内核中实体进行通信的一种手段，与真实的磁盘文件不同的是，这些虚拟文件的内容是内核运行时动态创建的，并不存在于磁盘文件中
    - Linux系统的许多命令本身都是通过分析/proc下的文件来实现的，比如：ps，top，uptime和free等等命令

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - procfs文件系统相关数据结构和文件操作定义在<linux/proc_fs.h>中
    - proc_dir_entry目录数据结构成员

      mode_t mode; // 文件权限保护位

      struct module *owner; // 当前拥有者模块

      read_proc_t *read_proc; // 读函数

      write_proc_t *write_proc; // 写函数

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 创建目录 proc_mkdir
    - 创建文件 create_proc_entry
    - 创建只读文件 create_proc_read_entry
    - 删除文件或目录 remove_proc_entry
    - 文件读 read_proc_t
    - 文件写 write_proc_t

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 创建目录 proc_mkdir
      - struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
        - » name: 待创建的目录名称
        - » parent: 父目录的proc_dir_entry结构体指针，如果该指针为NULL，则在/proc目录创建
        - » 返回: 创建好的目录的proc_dir_entry结构体指针，失败返回NULL

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 创建文件 create_proc_entry
      - struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent);
        - » name: 待创建的文件名称
        - » mode: 待创建的文件权限保护位，默认0755
        - » parent: 父目录的proc_dir_entry结构体指针，如果该指针为NULL，则在/proc目录创建
        - » 返回: 创建好的文件的proc_dir_entry结构体指针，失败返回NULL

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 创建只读文件 create_proc_read_entry
      - struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent, read_proc_t *read_proc, void *data);
        - » name: 待创建的文件名称
        - » mode: 待创建的文件权限保护位，默认0755
        - » parent: 父目录的proc_dir_entry结构体指针，如果该指针为NULL，则在/proc目录创建
        - » read_proc_t: 文件的读函数指针
        - » data: 将被读出的数据指针
        - » 返回: 创建好的文件的proc_dir_entry结构体指针，失败返回NULL

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 删除文件或目录 remove_proc_entry
      - void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
        - » name: 待删除的文件或目录名称
        - » parent: 待删除的文件或目录所在父目录的proc_dir_entry结构体指针，如果该指针为NULL，则在/proc目录删除文件或目录

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 文件读 read_proc_t
      - 当用户读取procfs文件系统中指定文件内容时，该数据产生函数被调用
      - typedef int (read_proc_t)(char *page, char **start, off_t off, int count, int *eof, void *data);
        - » page: 要返回给用户的信息存放页面，最大数据不超过一个页面大小，即PAGE_SIZE
        - » start: 一般不使用
        - » off: 读数据偏移
        - » count: 用户要读取的数据长度
        - » eof: 读到文件结尾时，需要把*eof设置为1
        - » data: 一般不使用

# Linux内核编程

- 文件系统和设备文件系统
  - procfs
    - 文件写 write_proc_t
      - 当用户向procfs文件系统中指定文件写入内容时，该数据接收函数被调用
      - typedef int (write_proc_t)(struct file *file, const char __user *buffer, unsigned long count, void *data);
        - » file: 该procfs文件对应的struct file结构，一般忽略
        - » buffer: 用户要写入的数据在用户空间存储的指针
        - » count: 用户要写入的数据大小
        - » data: 一般不使用

# Linux内核编程

- 文件系统和设备文件系统
  - sysfs
    - 根目录下的sys目录中显示出当前系统中的sysfs文件系统中的所有内容，系统初始化时将sysfs挂载到该目录
    - sysfs是一个虚拟的文件系统，它可以产生一个包括所有系统硬件的层级视图，与提供进程和状态信息的proc文件系统十分类似

# Linux设备驱动

- Linux设备驱动的作用
  - 控制硬件正常工作
    - 内核是操作系统的基本部分，而操作系统不能直接控制硬件，所以，需要设备驱动来作为操作系统和硬件设备的桥梁
  - 为Linux内核提供调用接口
    - 内核是为应用程序提供服务的，其本质是函数的集合，内核为应用程序对硬件设备访问提供统一的预定义接口，也为访问驱动程序提供预定义的接口

# Linux设备驱动

- 需要具备的知识结构
  - Linux设备驱动程序设计方法 (约%45)
  - Linux内核相关知识 (约%30)
  - 硬件相关知识 (约%25)

- 学习方法
  - 理论学习->动手实践->理论学习->动手实践->...

# Linux设备驱动

- Linux设备驱动分类
  - 字符设备驱动 (char device driver)
  - 块设备驱动 (block device driver)
  - 网络设备驱动 (network device driver)

# Linux设备驱动

- Linux设备驱动分类
  - 字符设备 (char device)
    - 采用字节流方式访问的设备称为字符设备，通常只能采用顺序访问方式，也有极少数可以前后移动访问指针的设备(如：帧捕捉卡等设备)
    - 系统标准字符设备，例如：字符终端、串口等设备
    - 常见待开发设备驱动的字符设备，例如：触摸屏、自定义键盘、视频捕捉设备、音频设备等

# Linux设备驱动

- Linux设备驱动分类
  - 块设备 (block device)
    - 该类设备通常在物理上不能按字节处理数据，只能通过一个或多个长度是512字节(或一个更大的2的n次幂的数据尺寸)的整块数据进行读、写、擦除等控制操作
    - 在Linux系统中，允许为块设备传送任意数据尺寸的字节。因此，实际上块设备和字符设备的区别仅仅在于驱动应该提供给内核的接口不同，内核内部管理数据的方式(会采用缓存机制等)也不同

# Linux设备驱动

- Linux设备驱动分类
  - 块设备 (block device)
    - 该类设备必须支持mount文件系统
    - 系统标准块设备，例如：磁盘、光盘、USB存储设备、Nand Flash存储设备等设备
    - 常见待开发设备驱动的块设备，基本上不用自己开发，Linux已经提供了几乎所有块设备驱动

# Linux设备驱动

- Linux设备驱动分类
  - 网络设备 (network device)
    - 在Linux系统中，网络设备是一类特殊设备，采用数据包传输方式访问设备，系统对该类设备提供对发送数据和接收数据的缓存，提供流量控制机制、对多协议的支持
    - 该类设备可通过ifconfig来创建和配置设备
    - 网络驱动同块驱动最大的不同在于网络驱动异步接收外界数据，而块驱动只对内核的请求做出响应

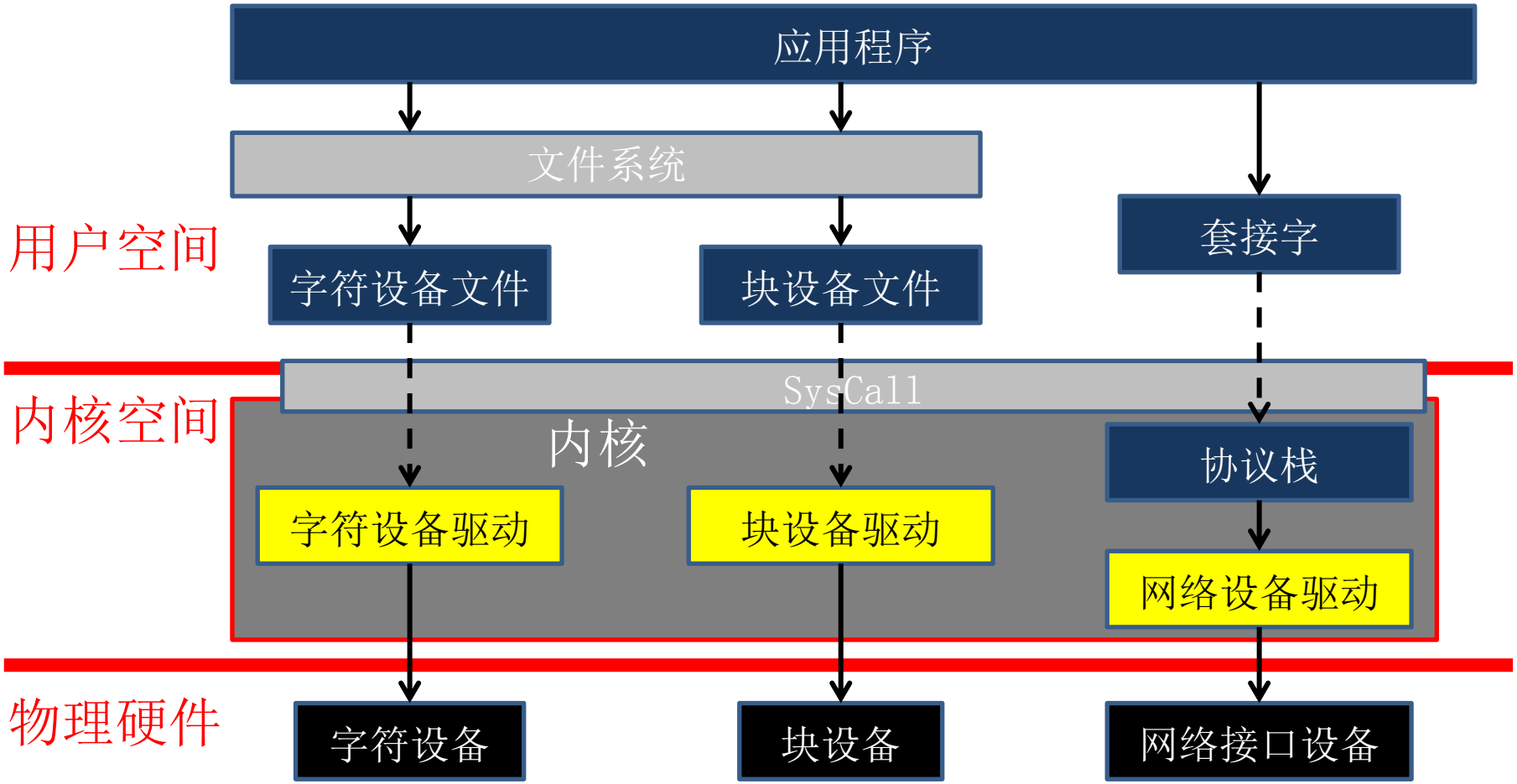
# Linux设备驱动

- Linux设备驱动分类
  - 网络设备 (network device)
    - 在Linux系统中，任何网络事务都通过接口来进行处理，一个接口通常是一个网络硬件设备，也可以是一个纯粹的网络软件设备，一个网络接口负责发送和接收数据报文
    - 系统标准网络设备，例如：回环接口
    - 常见待开发网络驱动的网络设备，该类设备驱动通常需要移植或开发

# Linux设备驱动

- Linux中的应用程序操作硬件设备
  - 在Linux系统中，应用程序通过设备文件(也叫设备节点) 使用统一的文件操作接口实现对驱动程序的操作，进一步实现对硬件设备(字符设备和块设备)的操作

# Linux设备驱动



# Linux字符设备驱动

- Linux字符设备驱动
  - 设备号
  - 设备文件
  - 数据结构
  - 设备注册
  - 设备操作

# Linux字符设备驱动

- 设备号
  - 应用程序通过字符设备文件来操作字符设备
  - 查看系统中的字符设备文件
    - 进入系统目录/dev
    - ls -l | grep "^c"
    - 该命令列出系统中的所有字符设备
    - 其中第五、六列内容为设备的主、次设备号
  - 文件/proc/devices中列出当前系统中处于活动状态的设备

# Linux字符设备驱动

- 设备号
  - 设备号的作用
    - 主设备号
      - 用于标识设备类型，内核代码依据该号码对应设备文件和对应的设备驱动程序
    - 次设备号
      - 用于标识同类型的不同设备个体，驱动程序依据该号码辨别具体操作的是哪个设备个体

# Linux字符设备驱动

- 设备号
  - 设备号数据类型
    - 数据类型dev_t用于定义设备号
    - 它本质上是一个unsigned int数据类型
    - 高12位是主设备号
    - 低20位是次设备号

# Linux字符设备驱动

- 设备号
  - 设备号数据类型
    - 内核为设备号数据类型提供一系列操作
    - 在<linux/kdev_t.h>中定义
    - 提取主设备号宏
      - unsigned int MAJOR(dev_t dev);
    - 提取次设备号宏
      - unsigned int MINOR(dev_t dev);
    - 合成设备号宏
      - dev_t MKDEV(unsigned int ma, unsigned int mi);

# Linux字符设备驱动

- 设备号
  - 设备号的分配
    - 静态分配
      - 内核源代码Documentation/devices.txt文件
        - » 该文件列出本内核源代码发行包中已经被使用和可以使用的主设备号
      - 在devices.txt中寻找可用的主设备号
    - 动态分配
      - 在驱动模块被加载时向内核动态申请主设备号

# Linux字符设备驱动

- 设备号
  - 设备号的静态分配
    - 优点
      - 简单
      - 驱动加载前已经知道设备号，可以提前创建设备文件
    - 缺点
      - 保留的可用主设备号资源有限
      - 一旦该驱动程序被广泛使用，该主设备号可能会引起设备号冲突，从而导致驱动程序无法加载

# Linux字符设备驱动

- 设备号
  - 设备号的静态分配
    - int register_chrdev_region(dev_t from, unsigned count, const char *name);
      - from: 待申请的设备号
      - count: 待申请的设备号数目
      - name: 设备名称(出现在/proc/devices)
      - 返回: 成功返回0，失败返回负值
    - 向内核申请从from开始的count个设备号(主设备号不变，次设备号增加)

# Linux字符设备驱动

- 设备号
  - 设备号的动态分配
    - 优点
      - 简单
      - 便于驱动推广
    - 缺点
      - 驱动程序被加载前设备号还没有分配，所以，无法知道设备号，也就不能提前为设备创建设备文件
    - 解决
      - 在驱动程序加载后，从/proc/devices文件中查询设备号

# Linux字符设备驱动

- 设备号
  - 设备号的动态分配
    - int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);
      - dev: 保存分配到的设备号
      - baseminor: 希望分配的起始次设备号
      - count: 需要分配的设备号数目
      - name:设备名称(出现在/proc/devices)
      - 返回: 成功返回0，失败返回负值
    - 请求内核动态分配count个设备号，且次设备号从baseminor开始

# Linux字符设备驱动

- 设备号
  - 设备号的注销
    - 设备号是内核中的一种有限资源
    - 不论使用何种方法分配设备号，都应该在不再使用的时候释放这些设备号

# Linux字符设备驱动

- 设备号
  - 设备号的注销
    - void unregister_chrdev_region(dev_t from, unsigned count);
      - from: 待注销的设备号
      - count:待注销设备号的数目

# Linux字符设备驱动

- 设备文件
  - 设备文件创建
    - 手工创建
    - 脚本创建
    - 自动创建

# Linux字符设备驱动

- 设备文件
  - 设备文件创建
    - 手工创建
      - mknod devicefilename type major minor
        - » devicefilename: 待创建的设备文件名称
        - » type: 待创建设备文件类型，通常是c或b
        - » major: 待创建设备文件的主设备号
        - » minor: 待创建设备文件的次设备号
      - 例如:
        - » mknod /dev/chardevicedriver c 240 0

# Linux字符设备驱动

- 字符设备相关数据结构
  - 在Linux字符设备驱动程序中，有4个非常重要的数据结构
    - 文件结构 struct file
    - inode结构 struct inode
    - 文件操作结构 struct file_operations
    - 字符设备结构 struct cdev

# Linux字符设备驱动

- 字符设备相关数据结构
  - 文件结构 struct file
    - 代表打开的设备文件
    - 在<linux/fs.h>中定义
    - 重要成员
      - const struct file_operations *f_op; // 可以在该文件上执行的所有操作的集合
      - unsigned int f_flags; // 文件被打开时传递的标志
      - loff_t f_pos; // 文件读写位置
      - void *private_data; // 文件私有数据

# Linux字符设备驱动

- 字符设备相关数据结构
  - inode结构 struct inode
    - 用于记录文件物理信息，不同于struct file
    - 一个文件可以对应多个file结构，但是只有一个inode结构
    - 在<linux/fs.h>中定义
    - 重要成员
      - dev_t i_rdev; // 设备对应的设备号
      - struct cdev *i_cdev; // 字符设备结构体

# Linux字符设备驱动

- 字符设备相关数据结构
  - 文件操作结构 struct file_operations
    - 实际是一个函数指针的集合，这些函数定义了能够对设备进行的操作
    - 这些指针指向驱动中的函数，每个函数完成一个特别的操作，不支持的操作指针留空
    - 在<linux/fs.h>中定义

# Linux字符设备驱动

- 字符设备相关数据结构
  - 字符设备结构 struct cdev
    - 内核使用该结构来表示一个字符设备
    - 在<linux/cdev.h>中定义
    - 重要成员
      - struct kobject kobj; // 设备对象
      - struct module *owner; // 该设备的拥有者驱动模块
      - struct file_operations *ops; // 设备操作集合
      - struct list_head list; // 内核维护的字符设备链表成员
      - dev_t dev; // 字符设备号
      - unsigned int count; // 设备个数

# Linux字符设备驱动

- 字符设备注册
  - 在Linux 2.6中，使用cdev来描述字符设备
  - 定义在<linux/cdev.h>中
  - 实现在内核源代码fs/char_dev.c中
    - 分配cdev
    - 初始化cdev
    - 添加cdev
    - 删除cdev

# Linux字符设备驱动

- 字符设备注册
  - 分配cdev
    - struct cdev *cdev_alloc(void);
      - 返回: 内核分配的cdev对象指针，失败返回NULL

# Linux字符设备驱动

- 字符设备注册
  - 初始化cdev
    - void cdev_init(struct cdev *cdev, const struct file_operations *fops);
      - cdev: 待初始化的cdev对象
      - fops: 可以在设备上执行的操作函数集合

# Linux字符设备驱动

- 字符设备注册
  - 添加cdev
    - int cdev_add(struct cdev *cdev, dev_t dev, unsigned count);
      - cdev: 待添加到内核的cdev对象
      - dev: 设备号
      - count: 待添加的设备个数
      - 返回: 成功返回0，失败返回负值

# Linux字符设备驱动

- 字符设备注册
  - 删除cdev
    - void cdev_del(struct cdev *cdev);
      - cdev: 待从内核删除的cdev对象

# Linux字符设备驱动

- 字符设备操作
  - 常见字符设备操作
    - open
    - release
    - read
    - write
    - llseek
    - ioctl
    - poll
    - fasync
    - mmap

# Linux字符设备驱动

- 字符设备操作
  - int (*open)(struct inode *inode, struct file *filp);
    - 该函数是设备文件上的第一个操作，并不要求驱动程序一定要实现该函数；如果该项为NULL，设备的打开操作永远成功
  - int (*release)(struct inode *inode, struct file *filp);
    - 该函数当设备文件被关闭时调用；与open方法类似，release操作也可以没有

# Linux字符设备驱动

- 字符设备操作
  - ssize_t (*read)(struct file *filp, char __user *buf, size_t count, loff_t *f_pos);
    - 从设备中读取数据
  - ssize_t (*write)(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);
    - 向设备发送数据

# Linux字符设备驱动

- 字符设备操作
  - loff_t (*llseek)(struct file *filp, loff_t off, int whence);
    - 修改文件的当前读写位置，并将新位置作为返回值
  - int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned int arg);
    - 执行设备控制

# Linux字符设备驱动

- 字符设备操作
  - unsigned int (*poll)(struct file *filp, struct poll_table_struct *pts);
    - 对应select系统调用，查询设备状态
  - int (*fasync) (int fd, struct file *filp, int mode);
    - 该操作用来通知设备它的FASYNC标志的改变

# Linux字符设备驱动

- 字符设备操作
  - int (*mmap)(struct file *filp, struct vm_area_struct vas);
    - 将设备内存映射到进程虚拟地址空间中

# Linux字符设备驱动

- 字符设备操作
  - open操作
    - 如果该操作为空，将使得open系统调用总是成功
    - open操作是驱动程序用来为以后的设备操作完成设备初始化工作的
      - 初始化设备
      - 标明次设备号
    - int (*open)(struct inode *inode, struct file *filp);
      - inode: 待操作的设备文件inode结构体指针
      - filp: 待操作的设备文件file结构体指针
      - 返回: 成功返回0，失败返回负值

# Linux字符设备驱动

- 字符设备操作
  - release操作
    - 如果该操作为空，将使得close系统调用总是成功
    - 该操作也称为close
      - 关闭设备
    - int (*release)(struct inode *inode, struct file *filp);
      - inode: 待操作的设备文件inode结构体指针
      - filp: 待操作的设备文件file结构体指针
      - 返回: 成功返回0，失败返回负值

# Linux字符设备驱动

- 设备文件
  - 设备文件创建
    - 脚本创建
      - 使用一个简单脚本替代insmod命令来获取内核分配的主设备号，再使用获取的主设备号创建设备文件
    - 自动创建
      - 在驱动程序被加载时，由其自身创建相应的设备节点
      - 该方法实现需要系统中有udev支持，对于使用busybox的嵌入式系统可以使能busybox自带的mdev来支持

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 手动创建设备文件
      - 不管是直接使用命令还是使用脚本，其实质都是调用mknod命令创建设备文件
    - 自动创建设备文件
      - 实际上内核提供了一组函数，可以用来在模块加载的时候在/dev目录下创建相应的设备节点，并在模块卸载时删除该设备节点

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 定义在<linux/device.h>中
    - class结构: 该结构体类型变量对应一个设备类，被创建的类存放在/sys目录下面
    - device结构: 该结构体类型变量对应设备，被创建的设备存放于/sys目录下面
    - 在加载驱动模块时，用户空间中的udev会自动响应device_create()函数，在/sys下寻找对应的类，从而为这个设备在/dev目录下创建设备文件

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 内核版本问题
      - 在内核2.4版本中使用devfs_register
      - 在内核2.6早期版本中使用class_device_register
      - 当前内核2.6.28.6中使用class_create和device_create

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 类创建和销毁
      - 定义在<linux/device.h>中
      - 实现在内核源代码drivers/base/class.c中

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 类创建
      - 为设备驱动创建一个设备类
      - struct class *class_create(struct module *owner, const char *name);
        - » owner: 创建设备类的驱动模块拥有者
        - » name: 待创建的设备类的类名称
        - » 返回: 创建好的设备类的指针，失败返回NULL

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 类销毁
      - 销毁设备驱动创建的对应设备类
      - void class_destroy(struct class *cls);
        - » cls: 待销毁的设备类

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 设备创建和销毁
      - 定义在<linux/device.h>中
      - 实现在内核源代码drivers/base/core.c中

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 设备创建
      - 为设备创建对应的设备文件
      - struct device *device_create(struct class *class, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...);
        - » class: 待创建的设备所属设备类
        - » parent: 指向可能存在的父设备的指针
        - » devt: 待创建设备的设备号(包括主设备号和次设备号)
        - » drvdata: 设备保留的驱动私有数据指针
        - » fmt: 待创建的设备文件名称
        - » 返回: 创建好的device的指针，失败返回NULL

# Linux字符设备驱动

- 设备文件
  - 自动创建
    - 设备销毁
      - 删除设备对应的设备文件
      - void device_destroy(struct class *class, dev_t devt);
        - » class: 待销毁的设备所属设备类
        - » devt: 待销毁设备的设备号(包括主设备号和次设备号)

# Linux字符设备驱动

- cdev改进
  - 为设备驱动支持多个设备个体做准备，针对cdev进行改进
    - 将代表字符设备的cdev对象包含在设备驱动定义的私有数据结构体中
    - 对设备驱动私有数据结构体采用内核内存分配方式为其分配内存
    - 将为每个设备添加cdev对象和创建设备节点封装为一个独立函数

# Linux字符设备驱动

- private_data改进
  - 为设备驱动支持多个设备个体做准备，针对private_data进行改进
    - 在设备打开操作中通过inode中保存的i_cdev获取代表当前设备的cdev对象
    - 通过代表当前设备的cdev对象得到包含该对象的设备私有数据结构体
    - 将设备私有数据结构体指针保存到struct file的private_data成员中
    - 在其它设备操作中直接使用保存在struct file的private_data成员中的当前设备私有数据结构体

# Linux字符设备驱动

- 支持多个设备个体
  - 为设备驱动支持多个设备个体对驱动进行改进
    - 循环调用为每个设备添加cdev对象和创建设备节点而封装的独立函数实现在系统中添加对多个设备个体的支持

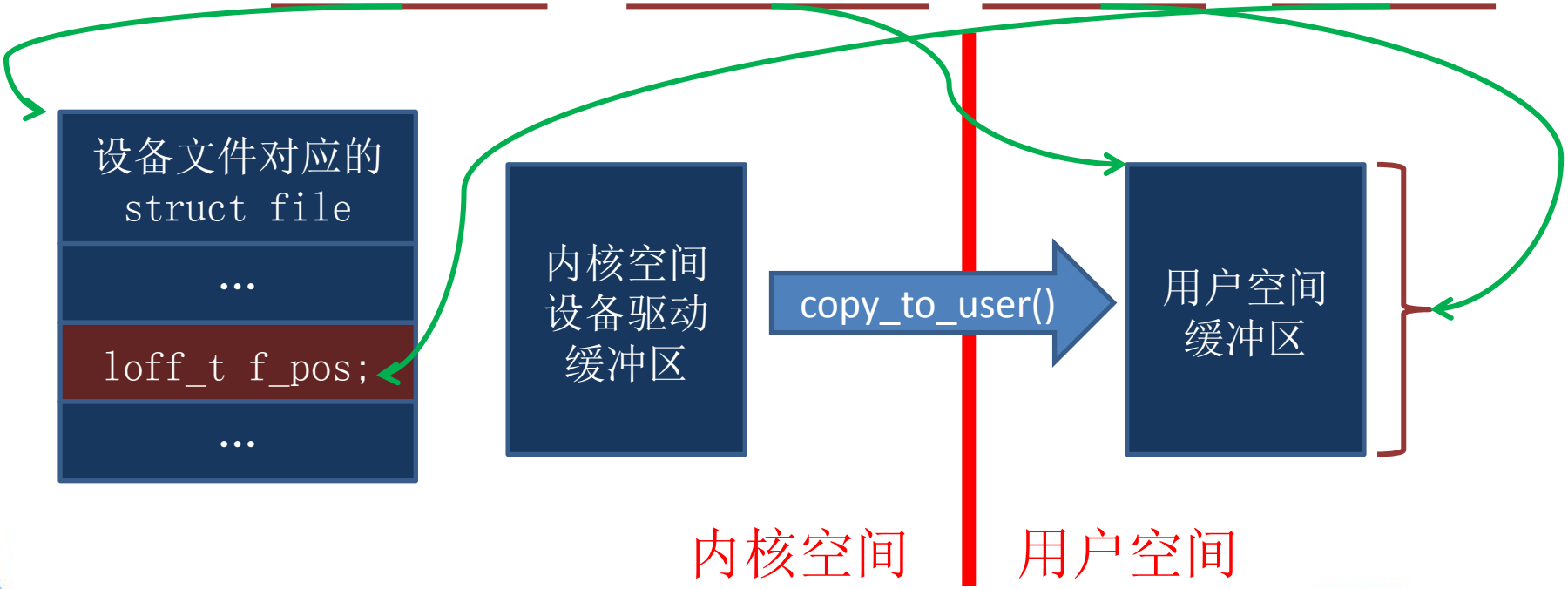
# Linux字符设备驱动

- 字符设备操作
  - 设备读操作
    - 如果该操作为空，将使得read系统调用返回负EINVAL失败，正常返回实际读取的字节数
    - ssize_t (*read)(struct file *filp, char __user *buf, size_t count, loff_t *f_pos);
      - filp: 待操作的设备文件file结构体指针
      - buf: 待写入所读取数据的用户空间缓冲区指针
      - count: 待读取数据字节数
      - f_pos: 待读取数据文件位置，读取完成后根据实际读取字节数重新定位
      - 返回: 成功实际读取的字节数，失败返回负值

# Linux字符设备驱动

- 字符设备操作
  - 设备读操作

(*read)(struct file *filp, char __user *buf, size_t count, loff_t *f_pos);



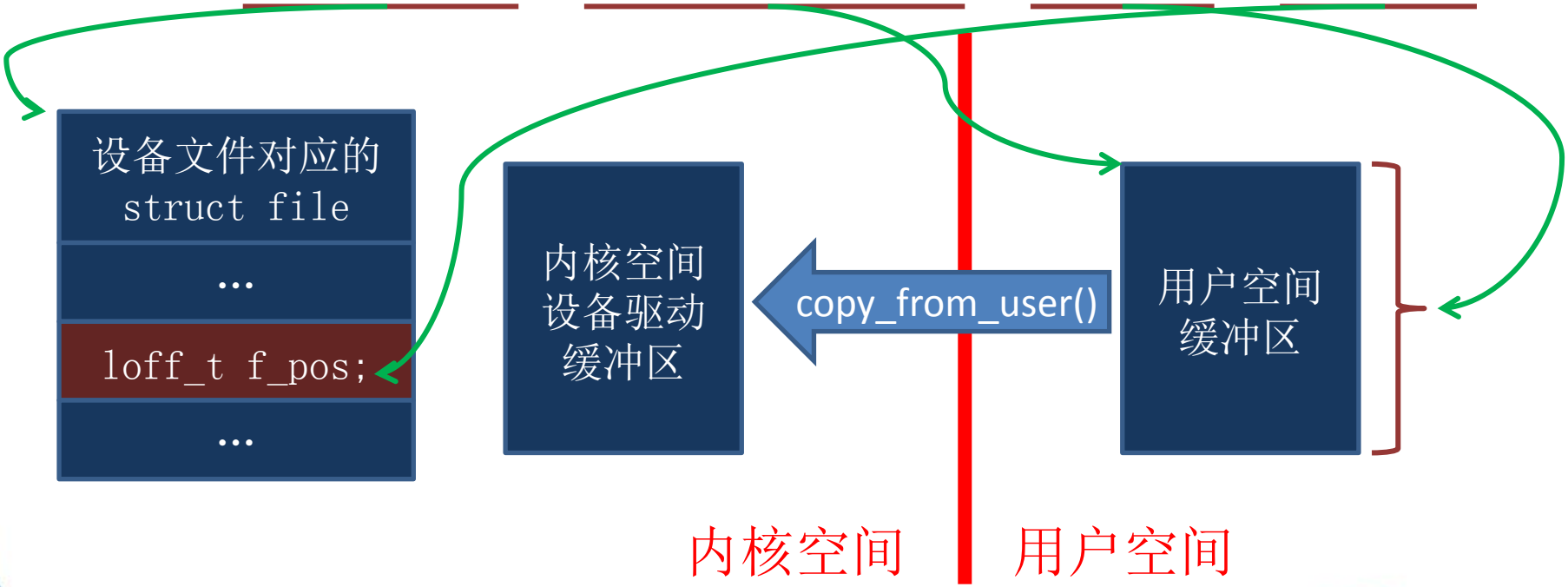
# Linux字符设备驱动

- 字符设备操作
  - 设备写操作
    - 如果该操作为空，将使得write系统调用返回负EINVAL失败，正常返回实际写入的字节数
    - ssize_t (*write)(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);
      - filp: 待操作的设备文件file结构体指针
      - buf: 待写入所读取数据的用户空间缓冲区指针
      - count: 待读取数据字节数
      - f_pos: 待读取数据文件位置，写入完成后根据实际写入字节数重新定位
      - 返回: 成功实际写入的字节数，失败返回负值

# Linux字符设备驱动

- 字符设备操作
  - 设备写操作

(*write)(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);



# Linux字符设备驱动

- 字符设备操作
  - 设备读和写操作
    - 读和写方法完成类似的工作
      - read: 从设备中读取数据到用户空间
      - write: 将数据从用户空间写入到设备
    - 读和写具有相同参数
      - filp: 文件指针
      - buf: 用户空间缓冲区
      - count: 请求读/写数据量
      - f_pos: 文件访问位置

# Linux字符设备驱动

- 字符设备操作
  - 设备读和写操作
    - 读和写方法的buf都是用户空间指针，它不能在内核模块中直接使用
    - 内核为驱动程序提供在内核空间和用户空间传递数据的方法
      - 定义在arch/arm/include/asm/uaccess.h中
      - 用户空间->内核空间
      - 内核空间->用户空间
      - 用户空间内存可访问性验证
      - 数据传送函数会检查用户空间内核是否可访问

# Linux字符设备驱动

- 字符设备操作
  - 用户空间->内核空间
    - copy_from_user函数
    - unsigned long copy_from_user(void *to, const void *from, unsigned long n);
      - to: 目标地址(内核空间)
      - from: 源地址(用户空间)
      - n: 将要拷贝数据的字节数
      - 返回: 成功返回0，失败返回没有拷贝成功的数据字节数

# Linux字符设备驱动

- 字符设备操作
  - 用户空间->内核空间
    - get_user宏
    - int get_user(data, ptr);
      - data: 可以是字节、半字、字、双字类型的内核变量
      - ptr: 用户空间内存指针
      - 返回: 成功返回0，失败返回非0

# Linux字符设备驱动

- 字符设备操作
  - 内核空间->用户空间
    - copy_to_user函数
    - unsigned long copy_to_user(void *to, const void *from, unsigned long n);
      - to: 目标地址(用户空间)
      - from: 源地址(内核空间)
      - n: 将要拷贝数据的字节数
      - 返回: 成功返回0，失败返回没有拷贝成功的数据字节数

# Linux字符设备驱动

- 字符设备操作
  - 内核空间->用户空间
    - put_user宏
    - int put_user(data, ptr);
      - data: 可以是字节、半字、字、双字类型的内核变量
      - ptr: 用户空间内存指针
      - 返回: 成功返回0，失败返回非0

# Linux字符设备驱动

- 字符设备操作
  - 用户空间内存可访问性验证
    - access_ok宏
    - int access_ok(int type, const void *addr, unsigned long size);
      - type: 取值为VERIFY_READ或VERIFY_WRITE，表示是读用户内存还是写用户内存
      - addr: 待验证的用户内存地址
      - size: 待验证的用户内存长度
      - 返回值: 返回非0代表用户内存可访问，返回0代表失败(存取有问题)，如果失败，ioctl操作应该返回-EFAULT

# Linux字符设备驱动

- 设备移位操作 llseek
  - 对应lseek系统调用的设备移位操作为llseek
  - 默认情况为允许设备移位操作
  - 大部分字符设备提供的都是数据流而不是一个数据区，比如串口，对于这些设备而言移位操作毫无意义
  - 设备可选择是否支持移位操作
    - 不支持设备移位操作
    - 支持设备移位操作

# Linux字符设备驱动

- 设备移位操作 llseek
  - 不支持设备移位操作
    - 对于不支持设备移位操作的设备驱动，应该通知内核设备不支持移位操作llseek
    - 支持方法:
      - 在open操作中调用nonseekable_open()函数
      - 同时指定llseek为no_llseek()函数

# Linux字符设备驱动

- 设备移位操作 llseek
  - 不支持设备移位操作
    - nonseekable_open()函数
      - 实现在内核源代码fs/open.c中
      - int nonseekable_open(struct inode *inode, struct file *filp);
        - » inode: 待操作的设备文件inode结构体指针
        - » filp: 待操作的设备文件file结构体指针
        - » 返回: 成功返回0，失败返回负值

# Linux字符设备驱动

- 设备移位操作 llseek
  - 不支持设备移位操作
    - no_llseek()函数
      - 实现在内核源代码fs/read_write.c中
      - loff_t no_llseek(struct file *filp, loff_t offset, int origin);
        - » filp: 待操作的设备文件file结构体指针
        - » offset: 待操作的定位偏移值
        - » origin: 待操作的定位起始位置
        - » 返回: 直接返回-ESPIPE(非法的定位操作)

# Linux字符设备驱动

- 设备移位操作 llseek
  - 支持设备移位操作
    - loff_t (*llseek)(struct file *filp, loff_t off, int whence);
      - filp: 待操作的设备文件file结构体指针
      - off: 待操作的定位偏移值(可正可负)
      - whence: 待操作的定位起始位置
      - 返回: 返回移位后的新文件读/写位置，并且新位置总为正值

# Linux字符设备驱动

- 设备移位操作 llseek
  - 支持设备移位操作
    - 定位起始位置
      - SEEK_SET: 0，表示文件开头
      - SEEK_CUR: 1，表示当前位置
      - SEEK_END: 2，表示文件尾

# Linux字符设备驱动

- 设备移位操作 llseek
  - 支持设备移位操作
    - 完成设备移位操作函数
      - 应该检查用户请求的定位操作合法性，若不合法，应该返回-EINVAL
    - 在读/写操作中完成读/写后更新文件位置
      - 应该根据当前文件位置检查当前读/写操作合法性

# Linux字符设备驱动

- 设备移位操作 llseek
  - 支持设备移位操作
    - 完成设备移位操作函数
      - 定义新位置变量
      - swith(whence)重新计算新位置
      - 判断如果新位置为负或超过缓冲区最大值返回-EINVAL
      - 用新位置更新filp中的f_pos
      - 返回新位置

# Linux字符设备驱动

- 设备移位操作 llseek
  - 支持设备移位操作
    - 在读/写操作中完成读/写后更新文件位置
      - 定义实际可读/写字节数变量
      - 根据当前文件位置和缓冲区最大值判断新的实际可读/写字节数变量值
      - 根据实际可读/写字节数变量读/写数据
      - 根据实际可读/写字节数变量更新文件位置
      - 返回实际可读/写字节数变量值

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 如果该操作为空，并且ioctl系统调用传递任何事先未定义的请求，系统调用将返回一个错误
  - 该操作实现ioctl系统调用向设备发出特定控制命令的处理，几个ioctl命令被内核识别处理

# Linux字符设备驱动

- 设备控制操作 ioctl
  - int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
    - inode: 待操作的设备文件inode结构体指针
    - filp: 待操作的设备文件file结构体指针
    - cmd: 接收到的设备控制命令
    - arg: 控制命令可能携带的参数
    - 返回: 成功返回0，失败返回负值

      如果access_ok失败， ioctl操作应该返回-EFAULT

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 设备控制命令
    - 设备控制命令为无符号整型数
    - 相关定义在<asm-generic/ioctl.h>中
    - 由4个段组成:

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 设备控制命令
    - 设备类型
      - 在内核源代码Documentation/ioctl/ioctl-number.txt文件包含内核发行包中已经使用的设备类型
      - 幻数(类型): 表明该命令属于哪个设备的命令，参考上述txt文件之后选择
      - 该段8位宽
    - 序列号
      - 控制命令字
      - 该段8位宽

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 设备控制命令
    - 数据方向
      - 代表数据传送方向
      - 数据传送方向是从应用程序角度来看
      - 该段2位宽
      - 数据方向常量:
        - » _IOC_NONE: 无数据传送
        - » _IOC_READ: 从驱动读数据
        - » _IOC_WRITE: 写数据到驱动

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 设备控制命令
    - 数据尺寸
      - 控制命令所指定参数的数据大小
      - 该值与体系结构相关，通常是13或14位宽

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 定义ioctl命令码相关宏
    - _IO(type, nr) // 定义无参数的命令
    - _IOR(type, nr, size) // 定义从驱动读数据的命令
    - _IOW(type, nr, size) // 定义写数据到驱动的命令
    - _IOWR(type, nr, size) // 定义双向传递数据的命令
      - type: 数据类型
      - nr: 序列号(控制命令字)
      - size: 数据尺寸(实际为数据类型)

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 从ioctl命令码提取数据段相关宏
    - _IOC_TYPE(cmd): 提取数据类型
    - _IOC_NR(cmd): 提取控制命令字
    - _IOC_DIR(cmd): 提取数据方向
    - _IOC_SIZE(cmd): 提取数据尺寸

# Linux字符设备驱动

- 设备控制操作 ioctl
  - 在内核空间和用户空间传递数据
    - copy_from_user
    - copy_to_user
    - get_user
    - put_user

# Linux字符设备驱动

- 设备控制操作 ioctl
  - ioctl函数处理
    - ioctl函数实现通常使用一个switch语句来判断不同命令码的操作，当没有匹配的命令码时，通常返回-EINVAL
    - 如果控制命令字参数是一个整数，可以直接使用；如果是指针，使用前应该检查该用户空间指针所指向的内存是否可访问

# Linux设备驱动调试

- 由于设备驱动程序运行于内核空间，因此关于它的调试方法与用户空间程序有着截然不同的区别
  - 内核对设备驱动调试支持
  - 使用printk函数
  - 使用procfs文件系统
  - 使用seq_file接口
  - 使用strace工具
  - 查看Oops信息
  - 使用gdb
  - 使用kgdb

# Linux设备驱动调试

- 内核对设备驱动调试支持
  - 设备驱动的调试需要内核的支持
  - 内核配置菜单中的Kernel hacking菜单项中有内核调试需要的相关选项

# Linux设备驱动调试

- 使用printk函数
  - printk会将信息打印到console或者/var/log/messages文件中
  - 在开发驱动时，往往会加入大量printk用于辅助调试
  - 在发布驱动时，应该去掉这些printk语句

# Linux设备驱动调试

- 使用printk函数
  - 使用printk函数的小技巧
    - 使用宏定义控制自定义printk函数
      - 在驱动调试时，重定义printk函数为自定义printk函数(可变参数的宏)
      - 在驱动发布时，自定义printk函数被定义为空
    - 例如:

```
#ifdef TDEBUG
#define TarenaDebug(fmt, args...) printk(KERN_ALERT
   "Tarena Driver:"fmt, ##args)
#else
    #define TarenaDebug(fmt, args...)
#endif
```

# Linux设备驱动调试

- 使用printk函数
  - 使用printk函数的小技巧
    - 控制自定义printk函数的宏定义从Makefile传递
    - 例如:

```
MTDEBUG = y
ifeq ($(MTDEBUG), y)
MTDFLAGS = -O2 -g -DTDEBUG
else
MTDFLAGS = -O2
endif
EXTRA_CFLAGS += $(MTDFLAGS)
```

# Linux设备驱动调试

- 使用procfs文件系统
  - 文件系统procfs也常被用于内核向用户导出信息，设备驱动程序中可以使用它输出调试信息
  - 具体操作参考前面文件系统和设备文件系统中关于procfs文件系统介绍

# Linux设备驱动调试

- 使用strace工具
  - 主要用来跟踪一个进程的系统调用或信号产生的情况
  - 可用它跟踪设备驱动执行情况
  - 它不仅可以从命令行调试一个新开始的程序，也可以绑定到一个已有的PID上来调试一个正在运行的程序

# Linux设备驱动调试

- 使用strace工具
  - 常用参数:
    - -f: 除了跟踪当前进程外，还跟踪其子进程
    - -o file: 将输出信息写到file中，而不是显示到标准错误输出(stderr)
    - -p pid: 绑定到一个由pid指定的正在运行的进程，常用于跟踪后台进程
    - -D: 在每行输出前加上相对时间戳，即每条系统调用所耗费时间

# Linux设备驱动调试

- 查看Oops信息
  - oops可以看作是内核级的段错误(segmentation fault)信息
  - 应用程序如果进行了非法内存访问或执行了非法指令，会出现segfault信号
  - 如果内核自己犯了非法内存访问或执行了非法指令，则会打出oops信息

# Linux设备驱动调试

- 查看Oops信息
  - 典型导致oops的情况:

    *(int *)0 = 0;
  - 在驱动中如果发现硬件和软件的运行情况与预期的不一致，可以通过使用上述语句抛出oops，便于提供出错时的上下文信息

# 并发与竞态

- 并发与竞态定义
  - 并发
    - 多个执行单元同时发生
  - 竞态
    - 并发的多个执行单元对共享资源(硬件资源或软件上的全局变量)的访问导致的竞争状态

# 并发与竞态

- 主要发生情况
  - 对称多处理器(SMP)的多个CPU之间
    - 由于它们使用共同的系统总线，所以访问共同的外设和存储
  - 单CPU内核抢占它的进程
    - 由于内核支持抢占调度，如果一个进程在内核执行的时候可能被另一个更高优先级进程打断，进程与抢占它的进程访问共享资源的情况类似于SMP的多个CPU

# 并发与竞态

- 主要发生情况
  - 中断(硬中断、软中断、Tasklet、Bottom Half)与进程之间
    - 中断可以打断正在执行的进程，如果中断处理程序访问进程正在访问的资源，则竞态也会发生
    - 中断本身也可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态

# 并发与竞态

- 解决办法
  - 解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其它的执行单元被禁止访问
  - 访问共享资源的代码区域称为临界区(Critical Section)，临界区需要以某种互斥机制加以保护
  - 在Linux系统中，中断屏蔽、原子操作、自旋锁和信号量是设备驱动程序中可以采用的互斥机制

# 并发与竞态

- 互斥机制
  - 中断屏蔽
  - 原子操作
  - 自旋锁
  - 读写锁
  - 顺序锁
  - 读-拷贝-更新

# 并发与竞态

- 互斥机制
  - 信号量
  - 读写信号量
  - 完成量
  - 互斥体

# 并发与竞态

- 中断屏蔽
  - 是在单CPU情况下避免竞态的一种简单方法，在进入临界区之前屏蔽中断，退出临界区之前恢复中断
  - 一般情况下，CPU都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核路径不被中断处理程序抢占，可以防止某些竞态条件的发生
  - 屏蔽中断之后，中断与进程之间的并发不再发生；由于Linux内核的进程调度等操作都依赖于中断来实现，内核抢占进程之间的并发不再发生

# 并发与竞态

- 中断屏蔽
  - 中断控制操作
    - 定义在<linux/irqflags.h>中
      - local_irq_enable()
      - local_irq_disable()
        - » 前者打开中断，后者关闭中断；两者必须成对使用
      - local_irq_save()
      - local_irq_restore()
        - » 前者关闭中断并保存中断状态，后者恢复前者关闭中断时保存的中断状态并打开中断；两者必须成对使用

# 并发与竞态

- 中断屏蔽
  - 一般使用方法

```
local_irq_disable(); // 关闭中断
critical section code; // 执行临界区代码
local_irq_enable(); // 打开中断
```

# 并发与竞态

- 中断屏蔽
  - 使用说明
    - 由于系统的异步I/O、进程调度等很多重要的系统操作都依赖于中断，所以，中断对于内核运行至关重要，在屏蔽中断期间所有的中断都无法得到处理
    - 因此，长时间屏蔽中断是危险的，有可能造成数据丢失，甚至系统崩溃
    - 所以，要求屏蔽中断之后，当前的内核代码执行路径应当尽快执行完临界区的代码

# 并发与竞态

- 原子操作
  - 原子操作定义
    - 原子操作指的是在执行路径中不会被别的代码路径中断操作
    - 原子操作依赖底层CPU的原子操作来实现，所有这些函数都与CPU架构密切相关
    - 在Linux内核中提供两类原子操作
      - 位原子操作
      - 整型原子操作

# 并发与竞态

- 原子操作
  - 位原子操作
    - 定义在arch/arm/include/asm/bitops.h中
    - 设置位
      - set_bit(nr, void *addr);
        - » 将addr地址内数据的第nr位设置为1
    - 清除位
      - clear_bit(nr, void *addr);
        - » 将addr地址内数据的第nr位设置为0

# 并发与竞态

- 原子操作
  - 位原子操作
    - 改变位
      - change_bit(nr, void *addr);
        - » 将addr地址内数据的第nr位反置
    - 测试位
      - int test_bit(nr, void *addr);
        - » 获取addr地址内数据的第nr位的值
    - 测试并操作
      - int test_and_set_bit(nr, void *addr);
      - int test_and_clear_bit(nr, void *addr);
      - int test_and_change_bit(nr, void *addr);

# 并发与竞态

- 原子操作
  - 整型原子操作
    - 定义在arch/arm/include/asm/atomic.h中
    - 整型原子变量数据类型定义

```
typedef struct {
        volatile int counter;
} atomic_t;
```

# 并发与竞态

- 原子操作
  - 整型原子变量操作
    - 设置原子变量的值
      - atomic_t v = ATOMIC_INIT(i);
        - » 定义一个原子变量v并初始化为i
      - atomic_set(atomic_t *v, int i);
        - » 设置原子变量v的值为i
    - 获取原子变量的值
      - int atomic_read(atomic_t *v);
        - » 返回原子变量的值

# 并发与竞态

- 原子操作
  - 整型原子变量操作
    - 原子变量加/减
      - atomic_add(int i, atomic_t *v);
        - » 原子变量的值增加i
      - atomic_sub(int i, atomic_t *v);
        - » 原子变量的值减少i

# 并发与竞态

- 原子操作
  - 整型原子变量操作
    - 原子变量自增/自减
      - atomic_inc(atomic_t *v);
        - » 原子变量的值增加1
      - atomic_dec(atomic_t *v);
        - » 原子变量的值减少1

# 并发与竞态

- 原子操作
  - 整型原子变量操作
    - 测试并操作
      - int atomic_inc_and_test(atomic_t *v);
      - int atomic_dec_and_test(atomic_t *v);
      - int atomic_sub_and_test(int i, atomic_t *v);
        - » 这组操作对原子变量执行自增、自减或减操作(没有加操作)后测试其是否为0，如果为0返回true，否则返回false

# 并发与竞态

- 原子操作
  - 整型原子变量操作
    - 操作并返回
      - int atomic_add_return(int i, atomic_t *v);
      - int atomic_sub_return(int i, atomic_t *v);
      - int atomic_inc_return(atomic_t *v);
      - int atomic_dec_return(atomic_t *v);
        - » 这组操作对原子变量执行加、减、自增或自减操作，并返回操作后的新值

# 并发与竞态

- 自旋锁
  - 自旋锁最多只能被一个执行单元持有
  - 自旋锁不会引起进程睡眠，如果一个执行进程试图获取一个已经被持有的自旋锁，那么进程就会一直进行忙循环，一直等待下去，等在这里看是否该自旋锁的持有者已经释放了锁，“自旋”近似于在“原地转圈”的意思

# 并发与竞态

- 自旋锁
  - 自旋锁数据类型定义
    - 定义在<linux/spinlock_types.h>中

```
typedef struct {
    raw_spinlock_t raw_lock;
    ...
} spinlock_t;

typedef struct {
    volatile unsigned int lock;
} raw_spinlock_t;
```

# 并发与竞态

- 自旋锁
  - 自旋锁操作
    - 定义在<linux/spinlock.h>中
    - 定义自旋锁
      - spinlock_t lock;
        - » 定义自旋锁类型变量
    - 初始化自旋锁
      - 自旋锁在使用前必须初始化
      - spin_lock_init(spinlock_t *lock);
        - » 动态初始化自旋锁

# 并发与竞态

- 自旋锁
  - 自旋锁操作
    - 获取自旋锁
      - spin_lock(spinlock_t *lock);
        - » 获取自旋锁，如果成功，立即获得自旋锁，并马上返回；如果失败，它将一直自旋在那里，直到该自旋锁的持有者释放该锁
      - int spin_trylock(spinlock_t *lock);
        - » 试图获取自旋锁，如果能获得锁，则立即获得锁并返回true；否则立即返回false

# 并发与竞态

- 自旋锁
  - 自旋锁操作
    - 释放自旋锁
      - spin_unlock(spinlock_t *lock);
        - » 释放自旋锁，它必须和spin_lock()或spin_trylock()成对使用

# 并发与竞态

- 自旋锁
  - 一般使用方法

```
spinlock_t lock; // 定义自旋锁
...
spin_lock_init(&lock); // 初始化自旋锁
...
spin_lock(&lock); // 获取自旋锁
critical section code; // 执行临界区代码
spin_unlock(&lock); // 释放自旋锁
```

# 并发与竞态

- 自旋锁
  - 衍生自旋锁
    - 虽然自旋锁可以保证临界区不受别的CPU或者是本CPU的抢占进程打扰，但是，还会受中断和中断底半部的影响
    - 所以，产生如下衍生:

      spin_lock_irq(spinlock_t *lock); // local_irq_disable() + spin_lock()

      spin_unlock_irq(spinlock_t *lock); // spin_unlock() + local_irq_enable()

# 并发与竞态

- 自旋锁
  - 衍生自旋锁

spin_lock_irqsave(spinlock_t *lock, unsigned long flags); // lock_irq_save() + spin_lock()

spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags); // spin_unlock() + local_irq_restore()

spin_lock_bh(spinlock_t *lock); // local_bh_disable() + spin_lock()

spin_unlock_bh(spinlock_t *lock); // spin_unlock() + local_bh_enable()

# 并发与竞态

- 自旋锁
  - 自旋锁使用注意事项
    - 只有占用锁的时间极短的情况下使用；由于获取锁是一直等待，所以临界区较大或有共享设备的时候，使用自旋锁会降低系统性能
    - 自旋锁可能导致死锁
      - 递归调用，也就是已经拥有自旋锁的CPU想第二次获取锁
      - 获取自旋锁之后再被阻塞；所以，在自旋锁占有期间，不能调用可能引起阻塞的函数；例如: kmalloc()、copy_from_user()、copy_to_user()等等

# 并发与竞态

- 读写锁
  - 自旋锁不关心临界区操作是读还是写，它都一视同仁
  - 实际上，对于共享资源并发访问时，多个执行单元同时读取它不会引发问题
  - 读写自旋锁是一种比自旋锁粒度更小的锁机制：对于写操作，只能最多有一个写进程；对于读操作，可以同时有多个读执行单元；但是，读和写也不能同时进行

# 并发与竞态

- 读写锁
  - 读写锁数据类型定义
    - 定义在<linux/spinlock_types.h>中

```
typedef struct {
    raw_rwlock_t raw_lock;
    ...
} rwlock_t;


typedef struct {
    volatile unsigned int lock;
} raw_rwlock_t;
```

# 并发与竞态

- 读写锁
  - 读写锁操作
    - 定义在<linux/spinlock.h>中
    - 定义和初始化
      - rwlock_t lock;
        - » 定义读写锁类型变量
      - rwlock_init(rwlock_t *lock);
        - » 动态初始化读写锁

# 并发与竞态

- 读写锁
  - 读写锁操作
    - 读锁定
      - read_lock(rwlock_t *lock);
        - » 当临界区中执行读操作时使用读锁定获取读写锁
    - 读解锁
      - read_unlock(rwlock_t *lock);
        - » 对应读写锁读锁定的解锁操作，必须和read_lock()成对使用

# 并发与竞态

- 读写锁
  - 读写锁操作
    - 写锁定
      - write_lock(rwlock_t *lock);
        - » 当临界区中执行写操作时使用写锁定获取读写锁
      - write_trylock(rwlock_t *lock);
        - » 当临界区中执行写操作时使用写锁定尝试获取读写锁
    - 写解锁
      - write_unlock(rwlock_t *lock);
        - » 对应读写锁写锁定的解锁操作，必须和write_lock()或write_trylock()成对使用

# 并发与竞态

- 读写锁
  - 衍生读写锁
    - 类似于自旋锁，读写锁也有针对中断和中断底半部操作的衍生读写锁
      - 读操作衍生读写锁
      - 写操作衍生读写锁

# 并发与竞态

- 读写锁
  - 衍生读写锁
    - 读操作衍生读写锁
      - read_lock_irq(rwlock_t *lock);
      - read_unlock_irq(rwlock_t *lock);
      - read_lock_irqsave(rwlock_t *lock, unsigned long flags);
      - read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
      - read_lock_bh(rwlock_t *lock);
      - read_unlock_bh(rwlock_t *lock);

# 并发与竞态

- 读写锁
  - 衍生读写锁
    - 写操作衍生读写锁
      - write_lock_irq(rwlock_t *lock);
      - write_unlock_irq(rwlock_t *lock);
      - write_lock_irqsave(rwlock_t *lock, unsigned long flags);
      - write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
      - write_lock_bh(rwlock_t *lock);
      - write_unlock_bh(rwlock_t *lock);

# 并发与竞态

- 读写锁
  - 一般使用方法

```
rwlock_t lock; // 定义一个读写锁
…
rwlock_init(&lock); // 初始化读写锁
…
// 读时使用读写锁
read_lock(&lock); //读写锁读锁定
critical section code; // 执行读操作临界区代码
read_unlock(&lock); //读写锁读解锁
…
// 写时使用锁
write_lock(&lock); //读写锁写锁定
critical section code; // 执行写操作临界区代码
write_unlock(&lock); //读写锁写解锁
```

# 并发与竞态

- 顺序锁
  - 顺序锁是对读写锁的一种优化
  - 特点是读写可以同时进行，仅写写互斥
  - 如果读时发生写，读操作必须重新读取数据

# 并发与竞态

- 顺序锁
  - 顺序锁数据类型定义
    - 定义在<linux/seqlock.h>中

```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;
```

# 并发与竞态

- 顺序锁
  - 顺序锁操作
    - 定义在<linux/seqlock.h>中
    - 定义和初始化
      - seqlock_t sl;
        - » 定义顺序锁类型变量
      - seqlock_init(seqlock_t *lock);
        - » 初始化顺序锁

# 并发与竞态

- 顺序锁
  - 顺序锁操作
    - 读顺序锁
      - unsigned read_seqbegin(seqlock_t *lock);
        - » 返回: 当前sequence值
    - 读顺序锁检查
      - int read_seqretry(seqlock_t *sl, unsigned start);
        - » start:  read_seqbegin获取的sequence值
        - » 返回: 如果临界区读操作执行期间发生过写操作，返回非0，否则返回0

# 并发与竞态

- 顺序锁
  - 顺序锁操作
    - 写锁定
      - write_seqlock(seqlock_t *sl);
        - » 当临界区执行写操作时使用写锁定获取顺序锁
      - write_tryseqlock(seqlock_t *sl);
        - » 当临界区执行写操作时使用写锁定尝试获取顺序锁
    - 写解锁
      - write_sequnlock(seqlock_t *sl);
        - » 对应顺序锁写锁定的解锁操作，必须和write_seqlock()或write_tryseqlock()成对使用

# 并发与竞态

- 顺序锁
  - 一般使用方法
    - 读操作
      - 读
        - » unsigned rsl = read_seqbegin(&sl);
      - 读检查
        - » read_seqretry( &sl, rsl );
      - 在对共享资源读访问之后再次调用检查，检查在读访问期间是否有写操作；如果有写操作，读执行单元就需要重新进行读操作

# 并发与竞态

- 顺序锁
  - 一般使用方法
    - 读操作
      - 读操作应用看起来更像：

```
do {
    seqnum = read_seqbegin( &sl ); // 获取执行读操作时的sequence值
    critical section code ; // 执行读操作临界区代码
} while(read_seqretry(&sl, seqnum)); // 检查读操作期间是否发生过写操作
```

# 并发与竞态

- 顺序锁
  - 一般使用方法
    - 写操作

```
write_seqlock( &sl ); // 顺序锁写锁定
critical section code; // 执行写操作临界区代码
write_sequnlock( &sl ); // 顺序锁写解锁
```

# 并发与竞态

- 顺序锁
  - 顺序锁使用限制
    - 要求被保护的共享资源不可以包含指针；因为，如果写操作使得指针失效，读操作单元对该已经失效的指针的操作将导致内核产生Oops信息

# 并发与竞态

- 读-拷贝-更新
  - 即RCU(Read Copy Update)
  - 定义在<linux/rcupdate.h>中
  - 原理：写前复制副本，在副本中作修改，在所有引用该共享数据的CPU都退出对共享数据的操作时，将指向原来的数据的指针重新指向新的被修改后的数据

# 并发与竞态

- 信号量
  - 信号量(semaphore)是用于保护临界区的一种常用方法
  - 内核的信号量在概念和原理上与用户态的信号量是一样的，但是它不能在内核之外使用，内核信号量实际上是一种睡眠锁

# 并发与竞态

- 信号量
  - 如果有一个任务想要获得已经被占用的信号量时，信号量会将这个进程放入一个等待队列，然后让其睡眠
  - 当持有信号量的进程将信号释放后，处于等待队列的进程将被唤醒，并让其获得信号量

# 并发与竞态

- 信号量
  - 内核信号量数据类型定义
    - 定义在<linux/semaphore.h>中

```
struct semaphore {
    spinlock_t lock;
    unsigned int count;
    struct list_head wait_list;
};
```

# 并发与竞态

- 信号量
  - 信号量操作
    - 定义在<linux/semaphore.h>中
    - 定义信号量
      - struct semaphore sem;
        - » 定义信号量类型变量

# 并发与竞态

- 信号量
  - 信号量操作
    - 初始化信号量
      - sema_init(struct semaphore *sem, int val);
        - » 用于初始化信号量，并设置信号量sem的值为val
      - init_MUTEX(struct semaphore *sem);
        - » 用于初始化信号量，并将信号量sem的值设置为1
      - init_MUTEX_LOCKED(struct semaphore *sem);
        - » 用于初始化信号量，并将信号量sem的值设置0；也就是在创建时就处于已锁状态
      - DECLARE_MUTEX(name);
        - » 定义一个名称为name的信号量，并将信号量初始化为1

# 并发与竞态

- 信号量
  - 信号量操作
    - 获取信号量
      - void down(struct semaphore *sem);
        - » 获取信号量sem；该函数可能导致进程睡眠，因此不能在中断上下文中使用
      - int down_interruptible(struct semaphore *sem);
        - » 获取信号量sem；如果信号量不可用，进程将被设置为TASK_INTERRUPTIBLE类型的睡眠状态
        - » 该函数由返回值来区分正常返回还是被信号中断返回；如果返回0，代表获取信号量正常返回；如果返回非0，代表被信号打断

# 并发与竞态

- 信号量
  - 信号量操作
    - 获取信号量
      - int down_killable(struct semaphore *sem);
        - » 获取信号量sem；如果信号量不可用，进程将被设置为TASK_KILLABLE类型的睡眠状态
      - int down_trylock(struct semaphore *sem);
        - » 该函数尝试获取信号量sem；如果能够立即获得，它就获得信号量并返回0；否则，返回非0值；它不会导致调用者睡眠，可以在中断上下文中使用

# 并发与竞态

- 信号量
  - 信号量操作
    - 释放信号量
      - void up(struct semaphore *sem);
        - » 该函数释放信号量sem；实质上是把sem的值加1，如果sem的值为非正数，表明有任务等待该信号量，因此需要唤醒等待者

# 并发与竞态

- 信号量
  - 一般使用方法

```
DECLARE_MUTEX( semm ); // 定义一个互斥信号量
...
down( &semm ); // 获取信号量，保护临界区
critical section code; // 执行临界区代码
up( &semm ); // 释放信号量
```

# 并发与竞态

- 信号量
  - 信号量和自旋锁
    - 信号量的实现依赖自旋锁，为保证信号量数据结构存取的原子性，在多CPU中需要自旋锁来互斥
    - 信号量可能允许多个持有者，而自旋锁在任何时候只能允许一个持有者；互斥信号量(只能有一个持有者)例外，允许有多个持有者的信号量叫计数信号量

# 并发与竞态

- 信号量
  - 信号量和自旋锁
    - 信号量适用于保持时间较长的情况；而自旋锁适用于保持时间短的情况
    - 在实际应用中自旋锁控制的代码通常只有短短几行，而持有自旋锁的时间也一般不超过两次上下文切换的时间，因为线程一旦进行切换，就至少需要花费切入切出两次，自旋锁的占有时间如果远远长于两次上下文切换，应该选择信号量

# 并发与竞态

- 信号量
  - 信号量用于同步
    - 如果信号量被初始化为0，则它可以用于同步
    - 同步意味着一个执行单元的继续执行需要等待另一个执行单元完成某件事情，保证执行的先后顺序

# 并发与竞态

- 读写信号量
  - 读写信号量和信号量的关系类似于读写锁和自旋锁的关系
  - 读写信号量数据类型定义
    - 定义在<linux/rwsem-spinlock.h>中

```
struct rw_semaphore {
    __s32 activity;
    spinlock_t wait_lock;
    struct list_head wait_list;
    ...
};
```

# 并发与竞态

- 读写信号量
  - 读写信号量操作
    - 实现在内核源代码kernel/rwsem.c中
    - 定义和初始化读写信号量
      - struct rw_semaphore rws;
        - » 定义读写信号量类型变量
      - init_rwsem(struct rw_semaphore *rws);
        - » 初始化读写信号量

# 并发与竞态

- 读写信号量
  - 读写信号量操作
    - 读信号量获取
      - down_read(struct rw_semaphore *rws);
        - » 当临界区中执行读操作时使用读信号量获取操作得到读写信号量
      - down_read_trylock(struct rw_semaphore *rws);
        - » 当临界区中执行读操作时尝试使用读信号量获取操作得到读写信号量
    - 读信号量释放
      - up_read(struct rw_semaphore *rws);
        - » 对应读写信号量读信号量获取操作的读信号量释放操作，必须和down_read()或down_read_trylock()成对使用

# 并发与竞态

- 读写信号量
  - 读写信号量操作
    - 写信号量获取
      - down_write(struct rw_semaphore *rws);
        - » 当临界区中执行写操作时使用写信号量获取操作得到读写信号量
      - down_write_trylock(struct rw_semaphore *rws);
        - » 当临界区中执行写操作时尝试使用写信号量获取操作得到读写信号量
    - 写信号量释放
      - up_write(struct rw_semaphore *rws);
        - » 对应读写信号量写信号量获取操作的写信号量释放操作，必须和down_write()或down_write_trylock()成对使用

# 并发与竞态

- 读写信号量
  - 一般使用方法

```
struct rw_semaphore rws; // 定义一个读写信号量
...
init_rwsem( &rws ); // 初始化读写信号量
...
// 读时获取信号量
down_read( &rws ); // 读写信号量读信号量获取
critical section code; // 执行读操作临界区代码
up_read( &rws ); // 读写信号量读信号量释放
...
// 写时获取信号量
down_write( &rws ); // 读写信号量写信号量获取
critical section code ; // 执行写操作临界区代码
up_write( &rws ); // 读写信号量写信号量释放
```

# 并发与竞态

- 完成量
  - 完成量是一种同步机制
  - 它用于一个执行单元等待另一个执行单元执行完某事
  - 完成量数据类型定义
    - 定义在<linux/completion.h>中

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

# 并发与竞态

- 完成量
  - 完成量操作
    - 定义在<linux/completion.h>中
    - 定义和初始化完成量
      - struct completion compl;
        - » 定义完成量类型变量
      - init_completion(struct completion *compl);
        - » 初始化完成量
      - DECLARE_COMPLETION(compl);
        - » 定义并初始化完成量

# 并发与竞态

- 完成量
  - 完成量操作
    - 等待完成量
      - void wait_for_completion(struct completion *compl);
        - » 等待一个completion被唤醒
    - 唤醒完成量
      - void complete(struct completion *compl);
      - void complete_all(struct completion *compl);
        - » 前者只唤醒一个等待完成量的执行单元，后者唤醒所有等待同一个完成量的执行单元

# 并发与竞态

- 互斥体
  - 互斥体数据类型定义
    - 定义在<linux/mutex.h>中

```
struct mutex {
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
    ...
};
```

# 并发与竞态

- 互斥体
  - 互斥体操作
    - 定义在<linux/mutex.h>中
    - 定义和初始化互斥体
      - struct mutex mutx;
        - » 定义互斥体类型变量
      - mutex_init(struct mutex *mutx);
        - » 初始化互斥体

# 并发与竞态

- 互斥体
  - 互斥体操作
    - 获取互斥体
      - mutex_lock(struct mutex *mutx);
      - mutex_lock_interruptible(struct mutex *mutx);
      - mutex_trylock(struct mutex *mutx);
        - » 其中，mutex_trylock()尝试获得互斥体；如果能够立即获得，它就获得互斥体并返回真，否则，立即返回假；它不会导致调用者睡眠，可以在中断上下文使用

# 并发与竞态

- 互斥体
  - 互斥体操作
    - 释放互斥体
      - mutex_unlock(struct mutex *mutx);
        - » 持有互斥体的执行单元必须在执行完临界区后释放互斥体

# 并发与竞态

- 互斥体
  - 一般使用方法

```
struct mutex mutx; // 定义互斥体
...
mutex_init(&mutx); // 初始化互斥体
...
mutex_lock(&mutx); // 获取互斥体
critical section code; // 执行临界区代码
mutex_unlock(&mutx); //释放互斥体
```

# 并发与竞态

- 内核等待队列
  - 在Linux驱动程序中，可以使用等待队列来实现进程阻塞
  - 等待队列可以看作保存进程的容器，在阻塞进程时，将进程放入等待队列，当进程被唤醒时，从等待队列中取出进程
  - 实际上，信号量等对进程的阻塞在内核中也依赖等待队列来实现

# 并发与竞态

- 内核等待队列
  - 等待队列数据类型定义
    - 定义在<linux/wait.h>中

```
struct __wait_queue_head {
   spinlock_t lock;
   struct list_head task_list;
};

typedef struct __wait_queue_head wait_queue_head_t;
```

# 并发与竞态

- 内核等待队列
  - 等待队列数据类型定义

```
struct __wait_queue {
    unsigned int flags;
    #define WQ_FLAG_EXCLUSIVE 0x01;
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};


typedef struct __wait_queue wait_queue_t;
```

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 定义在<linux/wait.h>中
    - 定义和初始化队列头
      - wait_queue_head_t wqh;
        - » 定义等待队列头类型变量
      - init_waitqueue_head(wait_queue_head_t *wqh);
        - » 初始化等待队列头
      - DECLARE_WAIT_QUEUE_HEAD(name);
        - » 定义并初始化等待队列头

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 定义和初始化队列
      - DECLARE_WAITQUEUE(name, task);
        - » 定义并初始化一个名称为name的等待队列，task通常被设置为代表当前进程的current指针
    - 添加/移除等待队列
      - add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
      - remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
        - » 前者是将等待队列wait添加到等待队列头q指向的等待队列数据链中，后者从中移除

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 等待事件
      - wait_event(queue, condition);
        - » 当condition为真时，立即返回；否则进程进入TASK_UNINTERRUPTIBLE类型的睡眠状态，并挂在queue指定的等待队列数据链上
      - wait_event_interruptible(queue, condition);
        - » 当condition为真时，立即返回；否则进程进入TASK_INTERRUPTIBLE类型的睡眠状态，并挂在queue指定的等待队列数据链上

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 等待事件
      - wait_event_killable(queue, condition);
        - » 当condition为真时，立即返回；否则进程进入TASK_KILLABLE类型的睡眠状态，并挂在queue指定的等待队列数据链上
      - wait_event_timeout(queue, condition, timeout);
        - » 当condition为真时，立即返回；否则进程进入TASK_UNINTERRUPTIBLE类型的睡眠状态，并挂在queue指定的等待队列数据链上；当阻塞时间timeout超时后，立即返回

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 等待事件
      - wait_event_interruptible_timeout(queue, condition, timerout);
        - » 当condition为真时，立即返回；否则进程进入TASK_INTERRUPTIBLE类型的睡眠状态，并挂在queue指定的等待队列数据链上；当阻塞时间timeout超时后，立即返回

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 唤醒队列
      - wake_up(wait_queue_head_t *queue);
        - » 唤醒由queue指向的等待队列数据链中的所有睡眠类型的等待进程
      - wake_up_interruptible(wait_queue_head_t *queue);
        - » 唤醒由queue指向的等待队列数据链中的所有睡眠类型为TASK_INTERRUPTIBLE的等待进程

# 并发与竞态

- 内核等待队列
  - 等待队列操作
    - 在等待队列中睡眠
      - sleep_on(wait_queue_head_t *q);
        - » 让进程进入不可中断的睡眠，并将它放入q指定的等待队列数据链中
      - interruptible_sleep_on(wait_queue_head_t *q);
        - » 让进程进入可中断的睡眠，并将它放入q指定的等待队列数据链中

# 并发与竞态

- 内核等待队列
  - 一般使用方法
    - 定义和初始化等待队列，将进程状态改变，并将等待队列添加到等待队列数据链中
      - 改变进程状态的方法：
        - » 调用set_current_state(state_value)函数
        - » 调用set_task_state(task, state_value)函数
        - » 直接采用current->state = TASK_INTERRUPTIBLE，类似于赋值语句
    - 通过schedule()调用放弃CPU，调度其它进程执行
    - 进程被其它地方唤醒，将等待队列移出等待队列头指向的数据链

# Linux字符设备驱动

- 阻塞/非阻塞
  - 实际上，应用程序并不关心驱动里面read/write具体实现，只管调用并获取返回值
  - 如果设备没有准备好数据给应用程序读或者没有准备好接收用户程序写，驱动程序应当阻塞进程，使它进入睡眠，直到请求可以得到满足

# Linux字符设备驱动

- 阻塞/非阻塞
  - 阻塞性设备驱动实现
    - 阻塞读
      - 在阻塞型驱动程序中，如果进程调用read设备操作，但是设备没有数据或数据不足，进程应该被阻塞；当新数据到达后，唤醒被阻塞进程
    - 阻塞写
      - 在阻塞型驱动程序中，如果进程调用write设备操作，但是设备没有足够的空间供其写入数据，进程应该被阻塞；但设备中的数据被读走后，缓冲区中空出部分空间，应该唤醒被阻塞进程

# Linux字符设备驱动

- 阻塞/非阻塞
  - 应用程序非阻塞读
    - 阻塞方式是文件读写操作的默认方式
    - 应用程序可以通过使用O_NONBLOCK标志来人为地设置读写操作为非阻塞方式
      - 定义在<asm-generic/fcntl.h>中
      - 如果设置了O_NONBLOCK标志，read和write的处理行为相同

# Linux字符设备驱动

- 设备轮询操作
  - 系统调用和驱动内设备操作对应

| 应用程序执行 | 设备操作 |
| --- | --- |
| open | open |
| close | release |
| read | read |
| write | write |
| lseek | llseek |
| ioctl | ioctl |
| select | poll |
| mmap | mmap |

# Linux字符设备驱动

- 设备轮询操作
  - select系统调用
    - 用于多路监控，当没有一个文件满足要求时，select调用将引起进程阻塞

# Linux字符设备驱动

- 设备轮询操作
  - 对应select系统调用
  - unsigned int(*poll)(struct file *filp, struct poll_table_struct *wait);
  - unsigned int(*poll)(struct file *filp, poll_table *wait);
    - filp: 文件指针
    - wait: 轮询表指针

# Linux字符设备驱动

- 设备轮询操作
  - poll设备操作的任务
    - 调用poll_wait()函数将等待队列添加到poll_table轮询表
    - 返回描述设备是否可读或可写的掩码

# Linux字符设备驱动

- 设备轮询操作
  - poll_table数据类型定义
    - 定义在<linux/poll.h>中

```
typedef struct poll_table_struct {
   poll_queue_proc qproc;
} poll_table;


typedef void (*poll_queue_proc)(struct file *,
   wait_queue_head_t *, struct poll_table_struct *);
```

# Linux字符设备驱动

- 设备轮询操作
  - poll_wait()函数
    - 定义在<linux/poll.h>中
    - void poll_wait(struct file *filp, wait_queue_head_t *wait_address, poll_table *p);
      - filp: 执行操作的设备文件指针
      - wait_address: 设备驱动的等待队列头
      - p: 内核传递的轮询表指针
    - poll_wait()函数调用不会引起阻塞，它仅仅将当前进程添加到wait参数指定的等待队列数据链中

# Linux字符设备驱动

- 设备轮询操作
  - poll操作返回值
    - 通常返回下列定义“或”的结果：
      - POLLIN　　设备可无阻塞读
      - POLLOUT　　设备可无阻塞写
      - POLLRDNORM　数据可读
      - POLLWRNORM　数据可写
    - 设备可读通常返回： POLLIN | POLLRDNORM
    - 设备可写通常返回： POLLOUT | POLLWRNORM

# Linux字符设备驱动

- 设备轮询操作
  - poll操作一般结构

```
xxx_poll()
{
   unsigned int mask = 0; // 定义返回值
   ...
   // 调用poll_wait()把进程添加到轮询表
   ...
   if( device is ready for read ) {
            mask = POLLIN | POLLRDNORM;
   }
   ...
   return mask;
}
```

# Linux字符设备驱动

- 异步通知
  - 概念
    - 阻塞: 设备未就绪则阻塞进程
    - 非阻塞: 设备未就绪则直接返回状态
    - 轮询: 由应用程序通过查询状态判断设备是否就绪
    - 异步: 一旦设备就绪，主动通知应用程序

# Linux字符设备驱动

- 异步通知
  - 信号
    - 在Linux系统中，异步通知使用信号来实现
    - 定义在内核源代码arch/arm/include/asm/signal.h中
    - 除SIGSTOP和SIGKILL两个信号外，进程能够忽略或捕获其它所有信号
    - 如果一个信号没有被进程捕获，内核将采用默认行为处理

# Linux字符设备驱动

- 异步通知
  - 支持异步通知机制应用程序实现
    - 完成F_SETOWN控制命令
      - 该命令设置设备文件的文件拥有者为本进程，这样从设备驱动发出的信号才能被本进程接收到
    - 完成F_SETFL控制命令
      - 该命令设置文件支持FASYNC模式，即异步通知模式
    - 完成signal()系统调用
      - 该系统调用连接SIGIO信号和对应的信号处理函数

# Linux字符设备驱动

- 异步通知
  - 支持异步通知机制设备驱动实现
    - 支持F_SETOWN控制命令
      - 在控制命令中需要将filp->f_owner设置为对应进程ID；这个任务由内核完成，设备驱动无需处理
    - 支持F_SETFL控制命令
      - 每当FASYNC标志改变时，设备驱动中的fasync()函数将得以执行，所以，驱动中应该实现fasync()函数
    - 释放信号
      - 在设备驱动中，当设备资源可以获得时，设备驱动应该通过调用kill_fasync()函数释放相应的信号

# Linux字符设备驱动

- 异步通知
  - 异步事件通知队列
    - fasync_struct数据类型定义
      - 定义在<linux/fs.h>中
      - 设备驱动中需要声明一个该数据类型变量
    - 设备驱动中对fasync_struct数据结构的使用
      - 常见用法：

```
struct xxx_cdev {
    struct cdev cdev;
    ...
    struct fasync_struct *async_queue;
};
```

# Linux字符设备驱动

- 异步通知
  - 设备驱动中的fasync()函数
    - fasync()设备操作
      - int (*fasync)(int fd, struct file *filp, int on);
        - » fd: 文件描述符
        - » filp: 文件结构体指针
        - » on: 异步事件通知队列操作标志
        - » 返回: 正值表示成功，0表示未改变，负值表示失败

# Linux字符设备驱动

- 异步通知
  - 设备驱动中的fasync()函数
    - 仅需要调用fasync_helper()函数
    - fasync()函数常见实现：

```
static int xxx_fasync(int fd, struct file *filp, int mode)
{
   struct xxx_cdev *dev = filp->private_data;

   return fasync_helper( fd, filp, mode, &dev->async_queue );
}
```

# Linux字符设备驱动

- 异步通知
  - fasync_helper()函数
    - 初始化fasync_struct异步事件通知队列，包括分配内存和设置属性
    - 释放初始化时为fasync_struct分配的内存

# Linux字符设备驱动

- 异步通知
  - kill_fasync()函数
    - 在设备资源可以获得时，应该调用该函数释放SIGIO信号；可读时为POLL_IN；可写时为POLL_OUT
    - 实现在内核源代码fs/fcntl.c中
    - void kill_fasync(struct fasync_struct **fp, int sig, int band);
      - fp: 异步时间通知队列
      - sig: 待释放的信号
      - band: 带宽，即POLL_IN或POLL_OUT

# Linux字符设备驱动

- 异步通知
  - 信号释放
    - 例如，在设备资源可读时实现：

```
static ssize_t xxx_write(struct file *filp, const char __user
   *buf, size_t count, loff_t *f_pos)
{
   struct xxx_dev *dev = filp->private_data;
   ...
   if( dev->async_queue) {
      kill_fasync( &dev->async_queue, SIGIO, POLL_IN );
   }
   ...
}
```

# Linux字符设备驱动

- 异步通知
  - 删除异步事件通知
    - 在文件关闭时，应该调用驱动的fasync函数将文件从异步事件通知队列数据链中删除
    - 常见实现：

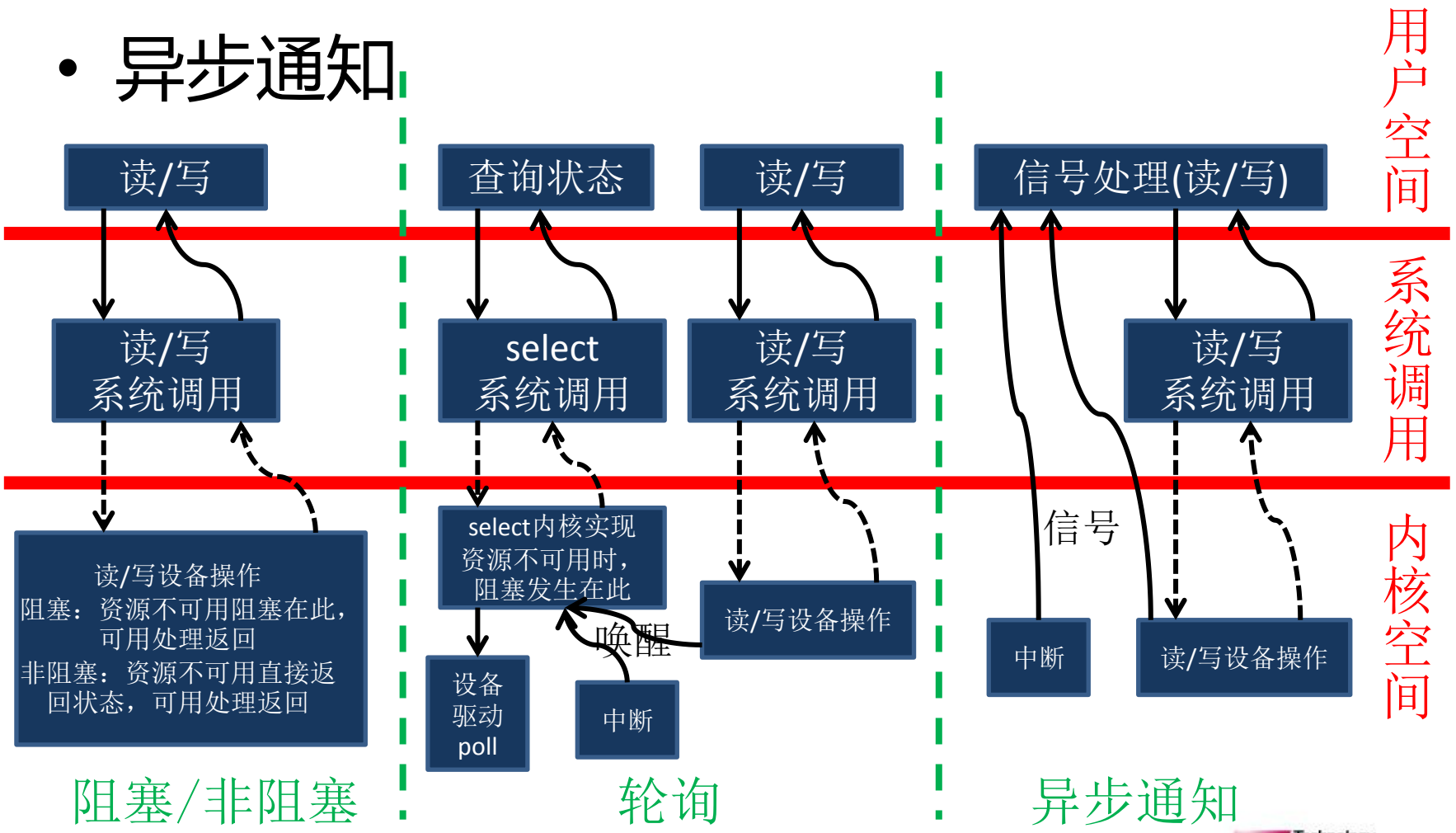
```
static int xxx_release(struct inode *inode, struct file
  *filp)
{
  ...
  xxx_fasync( -1, filp, 0 );
  ...
}
```

# Linux字符设备驱动

- 异步通知
  - 异步事件发起者问题
    - 对于多于一个文件被使能异步通知事件的应用程序进程
    - 当进程收到SIGIO信号，它并不知道是哪个文件有新数据提供
    - 此时，应用程序必须仍然依靠select()来找出具体是哪个文件发生变化

# Linux字符设备驱动

- 异步通知



# Linux字符设备驱动

- 中断和定时器
  - 中断定义
    - 中断是指CPU在执行过程中，出现了某些突发事件时CPU必须暂停执行当前的程序，转去处理突发事件，处理完毕后CPU又返回原程序被中断的位置并继续执行

# Linux字符设备驱动

- 中断和定时器
  - 中断分类
    - 按来源分类
      - 内部中断：来自于CPU内核，如软件中断指令、溢出、除法错误等
      - 外部中断：来自于CPU外部，由外设提出请求
    - 按是否可以屏蔽分类
      - 可屏蔽中断：可以通过屏蔽位禁止的中断，屏蔽后，中断不再得到响应，但是中断会继续产生
      - 不可屏蔽中断：不能被屏蔽的中断

# Linux字符设备驱动

- 中断和定时器
  - 中断分类
    - 按入口方法分类
      - 向量中断：CPU为不同的中断分配不同的中断号，每个中断号有对应的执行代码入口地址
      - 非向量中断：多个中断共享一个入口地址，由软件通过中断标志来识别具体是哪个中断

# Linux字符设备驱动

- 中断和定时器
  - Linux中断处理程序结构
    - 在Linux系统中，中断处理程序分解为两个半部：顶半部(Top Half)和底半部(Bottom Half)
      - 顶半部：完成尽可能少的比较紧急的功能，往往只是简单的读取寄存器中的中断状态并清除中断标志后就进行“登记中断”的工作，也就是将底半部处理程序挂到该设备的底半部执行队列中去，该过程不可中断
      - 底半部：它将完成中断事件的绝大多数任务，该部分任务不是非常紧急，并且相对比较耗时，该部分可以被新的中断打断

# Linux字符设备驱动

- 中断和定时器
  - 中断编程
    - 实现在内核源代码kernel/irq/manage.c中
      - 中断申请
      - 中断处理函数
      - 中断释放

# Linux字符设备驱动

- 中断和定时器
  - 中断申请
    - int request_irq(unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char *name, void *dev_id);
      - irq: 待申请的中断号
      - handler: 待注册的中断处理函数
      - irqflags: 中断标志
      - name: 中断设备的名称
      - dev_id: 传递给中断处理函数的指针，通常用于共享中断时传递设备结构体指针
      - 返回: 成功返回0，失败返回负值
        - » -EINVAL: 表示申请的中断号无效或者中断处理函数指针为空
        - » -EBUSY: 表示中断已经被占用并且不能共享

# Linux字符设备驱动

- 中断和定时器
  - 中断申请
    - 常见中断标志
      - IRQF_SHARED: 表示多个设备共享中断
      - IRQF_SAMPLE_RANDOM: 用于随机数种子的随机采样
      - IRQF_TRIGGER_RISING: 上升沿触发中断
      - IRQF_TRIGGER_FALLING: 下降沿触发中断
      - IRQF_TRIGGER_HIGH: 高电平触发中断
      - IRQF_TRIGGER_LOW: 低电平触发中断

# Linux字符设备驱动

- 中断和定时器
  - 中断处理函数
    - static irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
      - irq: 中断号
      - dev_id: 通常传递设备结构体指针
      - 返回: 常见返回值如下：
        - » IRQ_NONE: 未做处理
        - » IRQ_HANDLED: 正常后处理应该返回该值

# Linux字符设备驱动

- 中断和定时器
  - 中断释放
    - void free_irq(unsigned int irq, void *dev_id);
      - irq: 待释放的IRQ号
      - dev_id: 传递给中断处理函数的指针

# Linux字符设备驱动

- 中断和定时器
  - 中断编程
    - 使能和屏蔽中断
      - 指定中断号
      - 本CPU全部中断
      - 软中断和tasklet

# Linux字符设备驱动

- 中断和定时器
  - 使能和屏蔽中断
    - 指定中断号
      - 实现在内核源代码kernel/irq/manage.c中
      - void disable_irq(unsigned int irq);
        - » 屏蔽irq指定的中断
      - void disable_irq_nosync(unsigned int irq);
        - » 屏蔽irq指定的中断，该函数立即返回，不等待可能正在执行的中断处理程序
      - void enable_irq(unsigned int irq);
        - » 使能irq指定的中断

# Linux字符设备驱动

- 中断和定时器
  - 使能和屏蔽中断
    - 本CPU全部中断
      - 定义在<linux/irqflags.h>中
        - » local_irq_disable()
        - » local_irq_enable()
        - » local_irq_save()
        - » local_irq_restore()

# Linux字符设备驱动

- 中断和定时器
  - 使能和屏蔽中断
    - 软中断和tasklet
      - 实现在内核源代码kernel/softirq.c中
        - » local_bh_disable()
        - » local_bh_enable()

# Linux字符设备驱动

- 中断和定时器
  - 中断编程
    - 底半部机制
      - 在Linux中实现底半部的机制主要是
        - » tasklet
        - » 工作队列
        - » 软中断

# Linux字符设备驱动

- 中断和定时器
  - tasklet
    - 定义在<linux/interrupt.h>中
    - tasklet定义
      - DECLARE_TASKLET( taskletname, tasklet_func, data);
        - » taskletname: 待定义的tasklet名字
        - » tasklet_func: tasklet处理函数
        - » data: 待传入tasklet处理函数的参数

# Linux字符设备驱动

- 中断和定时器
  - tasklet
    - tasklet处理函数
      - void tasklet_func(unsigned long data);
    - tasklet调用
      - 在中断处理函数中调用tasklet_schedule()函数
      - void tasklet_schedule(struct tasklet_struct *taskletname);

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列(work queue)是Linux kernel中将工作推后执行的一种机制；这种机制和BH或Tasklet不同之处在于工作队列是把推后的工作交由一个内核线程去执行，因此工作队列的优势就在于它允许重新调度甚至睡眠
    - 工作队列是2.6内核开始引入的机制，在2.6.20之后，工作队列的数据结构发生了一些变化，被拆分成两个部分
    - 在此，主要对2.6.20之后的版本做介绍

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作数据类型定义
      - 定义在<linux/workqueue.h>中

```
struct work_struct {
    atomic_long_t data; // 记录工作状态和指向工作者线程的指针
    struct list_head entry; // 工作数据链成员
    work_func_t func; // 工作处理函数，由用户实现
};
```

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作数据类型定义
      - 定义在<linux/workqueue.h>中

```
typedef void (*work_func_t)(struct work_struct
  *work); // 工作函数原型
struct delayed_work {
  struct work_struct work; // 工作结构体
  struct timer_list timer; // 推后执行的定时器
}; // 处理延迟执行的工作的结构体
```

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列操作
      - 初始化工作
        - » INIT_WORK(struct work_struct *work, work_func_t func);
          - 初始化工作队列并指定工作队列处理函数
        - » INIT_DELAYED_WORK(struct delayed_work *work, work_func_t func);
          - 初始化延迟工作队列并指定工作队列处理函数

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作操作
      - 调度工作
        - » int schedule_work(struct work_struct *work);
          - 调度工作，即把工作处理函数提交给缺省的工作队列和工作者线程
        - » int schedule_delayed_work(struct delayed_work *work, unsigned long delay);
          - 调度延迟工作，即把工作处理函数提交给缺省的工作队列和工作者线程，并指定延迟时间(同内核定时器延迟处理)

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作操作
      - 刷新工作队列
        - » void flush_scheduled_work(void);
          - 刷新缺省工作队列，此函数会一直等待，直到队列中的所有工作都被执行完成
      - 取消延迟工作
        - » int cancel_delayed_work(struct delayed_work *work);
          - 取消缺省工作队列中处于等待状态的延迟工作

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作操作
      - 取消工作
        - » int cancel_work_sync(struct work_struct *work);
          - 取消缺省工作队列中处于等待状态的工作，如果工作处理函数已经开始执行，该函数会阻塞直到工作处理函数完成

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作者线程
      - 工作者线程本质上是一个普通的内核线程，在默认情况下，每个CPU均有一个类型为“events”的工作者线程，当调用schedule_work时，这个工作者线程会被唤醒去执行工作链表上的所有工作
      - 以上介绍操作均是采用缺省工作者线程来实现工作队列，其优点是简单易用，缺点是如果缺省工作队列负载太重，执行效率会很低，这就需要我们创建自己的工作者线程和工作队列

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列
      - 工作队列数据类型定义
        - » 定义在内核源代码kernel/workqueue.c中

```
struct workqueue_struct {
        …
        struct list_head list;
        const char *name;
        …
}
```

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列操作
      - 创建工作队列
        - » struct workqueue_struct *create_workqueue(const char *name);
          - 创建新的工作队列和相应的工作者线程，name用于该内核线程的命名

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列操作
      - 调度工作
        - » int queue_work(struct workqueue_struct *wq, struct work_struct *work);
          - 调度工作，类似于schedule_work()函数；将指定工作work提交给指定工作队列wq

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列操作
      - 调度延迟工作
        - » int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay);
          - 调度工作，类似于schedule_work()函数；将指定延迟工作work提交给指定工作队列wq，并指定延迟时间(同内核定时器延迟处理)

# Linux字符设备驱动

- 中断和定时器
  - 工作队列
    - 工作队列操作
      - 刷新工作队列
        - » void flush_workqueue(struct workqueue_struct *wq);
          - 刷新指定工作队列wq，此函数会一直等待，直到队列中的所有工作都被执行完成
      - 销毁工作队列
        - » void destroy_workqueue(struct workqueue_struct *wq);
          - 销毁指定工作队列wq

# Linux字符设备驱动

- 中断和定时器
  - 软中断
    - 软中断使用软件方式模拟硬件中断，目的是实现异步执行
    - tasklet即基于软中断实现
    - 软中断和tasklet仍然运行与中断上下文，工作队列运行于进程上下文

# Linux字符设备驱动

- 内存与I/O访问
  - I/O空间
    - 在X86处理器中才有I/O空间的概念
  - 内存空间
    - 内存空间可以直接通过地址、指针来访问，程序和程序运行时使用的变量和其它数据都存在于内存空间中
    - 绝大多数嵌入式处理器只有内存空间

# Linux字符设备驱动

- 内存与I/O访问
  - Linux系统中的虚拟地址和物理地址
    - 定义在arch/arm/include/asm/memory.h中
      - unsigned long virt_to_phys(void *x);
        - » 虚拟地址转换为物理地址
      - void *phys_to_virt(unsigned long x);
        - » 物理地址转换为虚拟地址

# Linux字符设备驱动

- 内存与I/O访问
  - 设备I/O内存访问
    - I/O内存申请
    - I/O内存映射
    - I/O内存存取
    - I/O内存释放

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存申请
    - 内核将I/O内存作为资源来进行管理，在使用之前需要request_mem_region()函数来申请
    - 通过request_mem_region()函数申请的I/O内存资源，不再使用时可以用release_mem_region()函数释放

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存申请
    - 定义在<linux/ioport.h>中
      - struct resource *request_mem_region(unsigned long first, unsigned long len, const char *name);
        - » first: 待申请访问的I/O内存起始地址(物理地址)
        - » len: 待申请访问的I/O内存区块长度
        - » name: 该内存区块名称
        - » 返回: 成功返回非NULL，失败返回NULL

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存映射
    - 在内核中访问I/O内存之前，需要首先使用ioremap()函数将设备所处的物理地址映射到内核虚拟地址，该内核虚拟地址可以用来存取特定的物理地址范围；通过ioremap()函数获得的虚拟地址应该被iounmap()释放
    - ioremap()函数类似于vmalloc()功能，也需要建立新的页表，但是它不进行vmalloc()中所执行的内存分配行为

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存映射
    - 定义在arch/arm/include/asm/io.h中
      - void __iomem *ioremap(unsigned long phys_addr, size_t size);
        - » phys_addr: 待映射的I/O内存物理起始地址(物理地址)
        - » size: 待映射的I/O内存区块大小
        - » 返回: 成功返回重映射后的内核I/O内存虚拟地址，失败返回NULL

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存存取
    - 内核为I/O内存的存取提供了一系列访问函数
      - 读取I/O内存
      - 写入I/O内存
      - 连续读取I/O内存
      - 连续写入I/O内存
      - 复制I/O内存
      - 设置I/O内存

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存存取
    - 读取I/O内存
      - unsigned int ioread8(void *addr);
      - unsigned int ioread16(void *addr);
      - unsigned int ioread32(void *addr);
    - 写入I/O内存
      - unsigned int iowrite8(void *addr);
      - unsigned int iowrite16(void *addr);
      - unsigned int iowrite32(void *addr);

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存存取
    - 连续读取I/O内存
      - void ioread8_rep(void *addr, void *buf, unsigned long count);
      - void ioread16_rep(void *addr, void *buf, unsigned long count);
      - void ioread32_rep(void *addr, void *buf, unsigned long count);

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存存取
    - 连续写入I/O内存
      - void iowrite8_rep(void *addr, void *buf, unsigned long count);
      - void iowrite16_rep(void *addr, void *buf, unsigned long count);
      - void iowrite32_rep(void *addr, void *buf, unsigned long count);

# Linux字符设备驱动

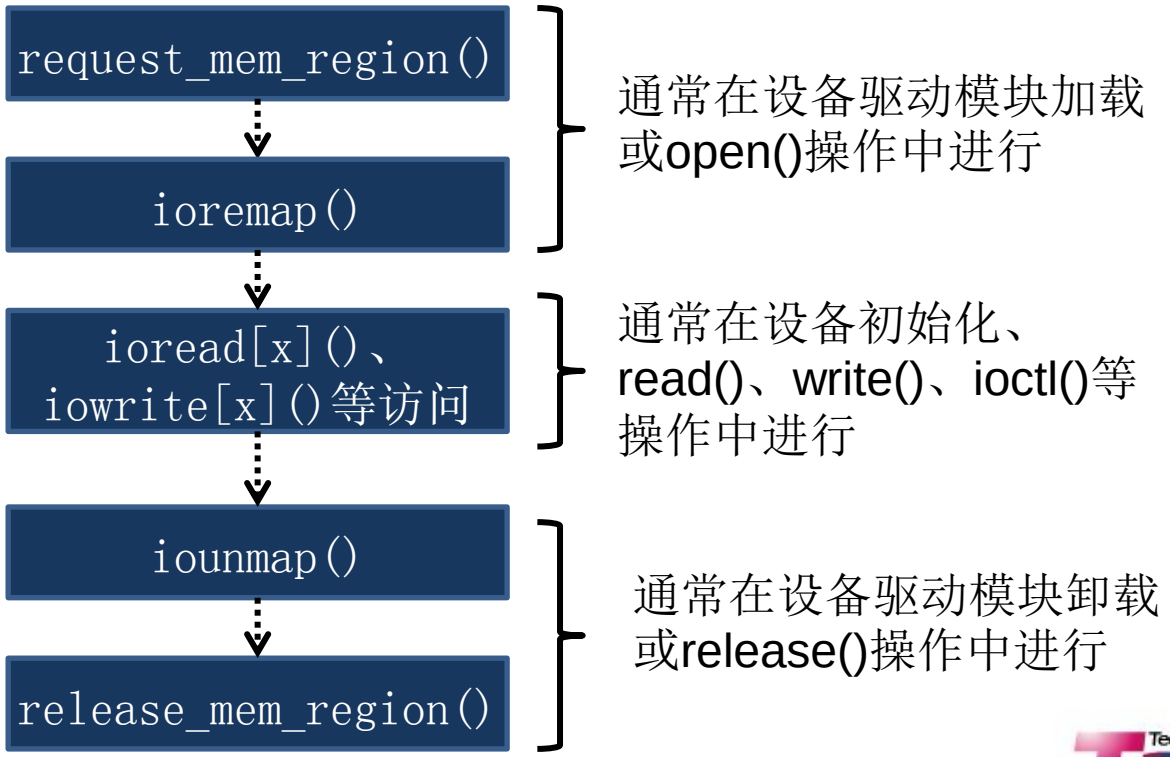
- 内存与I/O访问
  - I/O内存存取
    - 复制I/O内存
      - void memcpy_fromio(void *dest, void *source, unsigned int count);
      - void memcpy_toio(void *dest, void *source, unsigned int count);
    - 设置I/O内存
      - void memset_io(void *addr, u8 value, unsigned int count);

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存释放
    - 当I/O内存不再被使用时应该即时释放
      - void iounmap(void __iomem * io_addr);
        - » 释放由ioremap()映射的I/O内存
      - void release_mem_region(unsigned long start, unsigned long len);
        - » 释放由request_mem_region()申请的I/O内存

# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存访问流程



# Linux字符设备驱动

- 内存与I/O访问
  - I/O内存静态映射
    - 在ARM Linux系统中，通常会在内核初始化代码中建立外设I/O内存物理地址到内核虚拟地址的静态映射，这个映射通过map_desc数据结构来描述
    - 定义在arch/arm/include/asm/mach/map.h中

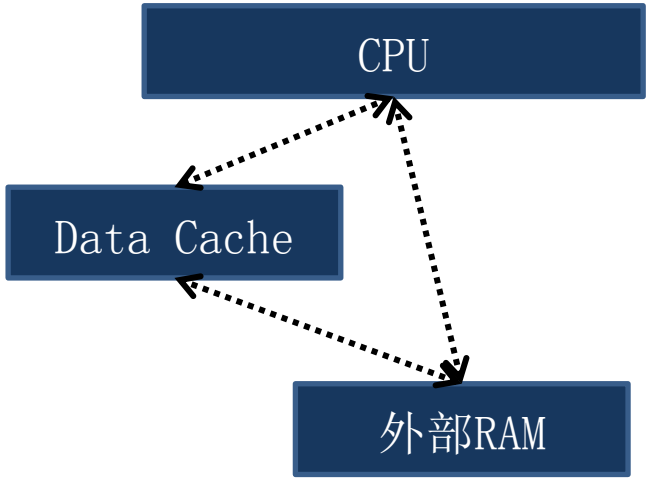
```
struct map_desc {
    unsigned long virtual; // 待映射的内核虚拟地址
    unsigned long pfn; // 从__phys_to_pfn(phy_addr)得到
    unsigned long length; // 待映射的内存区块大小
    unsigned int type; // 待映射的内存类型
};
```

# Linux字符设备驱动

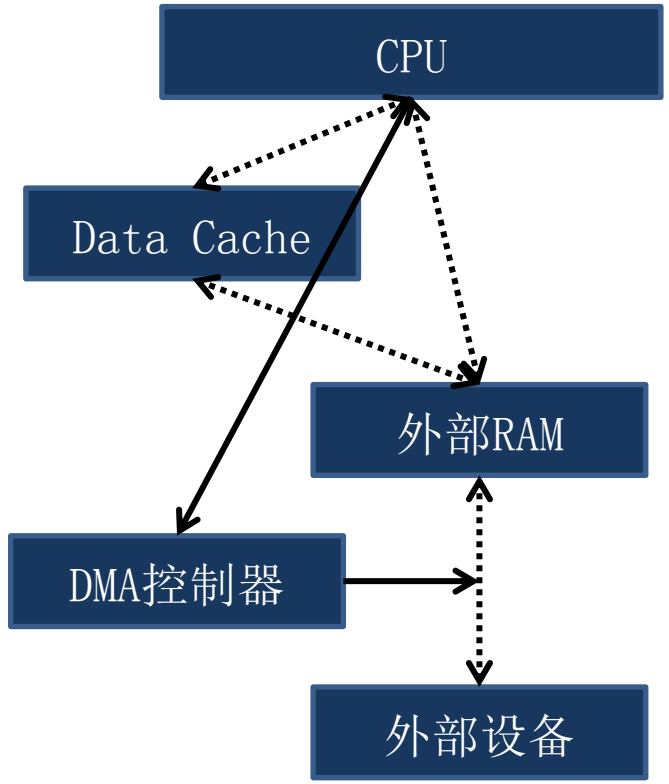
- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA是一种无需CPU的参与就可以在外设与系统内存之间进行双向数据传输的硬件机制
    - DMA本身不属于一种等同于字符设备、块设备和网络设备的外设，它只是外设与内存交互数据的一种方式

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA缓冲区与Cache




无DMA情况




有DMA情况

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA缓冲区
      - 内存中用于与外设交互数据的一块区域被称做DMA缓冲区，在设备不支持scatter/gather CSG(分散/聚集操作)的情况下，DMA缓冲区必须是物理上连续的
      - 对于ISA设备而言，DMA操作只能在16MB以下的内存中进行，在使用kmalloc()或__get_free_pages()等函数申请DMA缓冲区时，应使用GFP_DMA标志

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA映射
      - 基于DMA的硬件使用总线地址而非物理地址，总线地址是从设备角度上看到的内存地址，物理地址则是从CPU角度上看到的未经转换的内存地址（经过转换的为虚拟地址）
      - DMA映射必须考虑Cache一致性问题
      - DMA映射包括两个方面的工作：
        - » 分配一片DMA缓冲区
        - » 为这片缓冲区产生设备可访问的地址

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA缓冲区分配
      - 申请DMA缓冲区的宏__get_dma_pages()
        - » #define __get_dma_pages(gfp_mask, order) __get_free_pages((gfp_mask) | GFP_DMA,(order))

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA缓冲区分配
      - 分配与释放DMA一致性的内存区域
        - » void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flag)
          - dev: 待分配DMA缓冲区的设备
          - size: 待分配DMA缓冲区的大小
          - dma_handle: DMA缓冲区的总线地址
          - gfp: 内存分配标志
          - 返回: 申请到的DMA缓冲区的虚拟地址
        - » void dma_free_coherent(struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle);

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA缓冲区分配
      - 分配与释放写合并的DMA缓冲区
        - » void *dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
          - dev: 待分配DMA缓冲区的设备
          - size: 待分配DMA缓冲区的大小
          - dma_handle: DMA缓冲区的总线地址
          - gfp: 分配标志
          - 返回: 申请到的DMA缓冲区的虚拟地址
        - » #define dma_free_writecombine(dev,size,cpu_addr,handle) dma_free_coherent(dev,size,cpu_addr,handle)

# Linux字符设备驱动

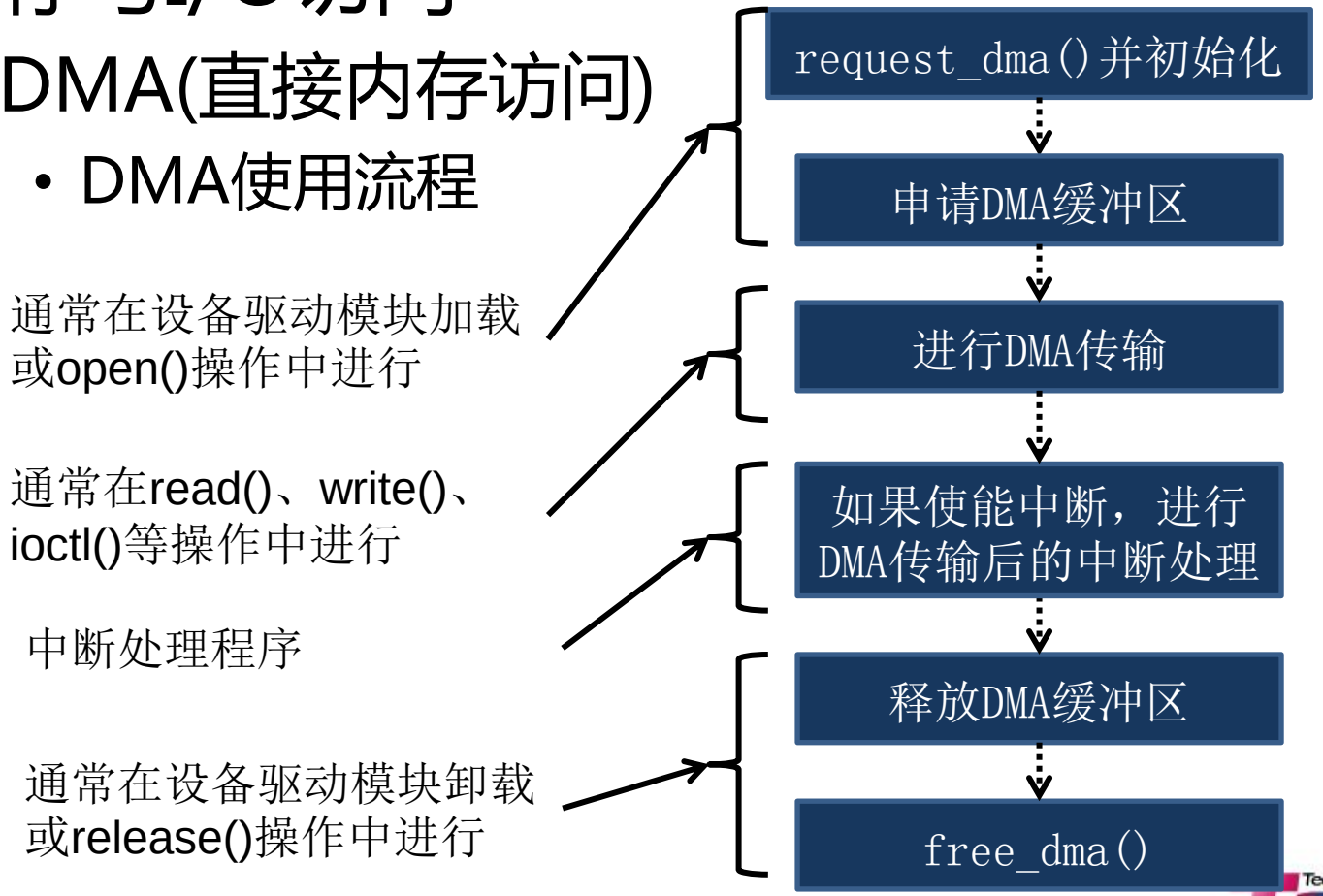
- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA通道
      - 和中断一样，在使用DMA之前，设备驱动程序需要先向内核申请DMA通道

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA通道操作
      - 请求DMA通道
        - » int request_dma(dmach_t channel, const char * device_id);
          - channel: 待申请的DMA通道号
          - device_id: 标识设备的字符串(出现在/proc/dma中)
          - 返回: 成功返回0，失败返回负值
      - 释放DMA通道
        - » void free_dma(dmach_t channel);
          - channel: 待释放的DMA通道号

# Linux字符设备驱动

- 内存与I/O访问
  - DMA(直接内存访问)
    - DMA使用流程



# Linux字符设备驱动

- mmap设备操作
  - mmap设备操作对应mmap系统调用
  - mmap系统调用过程
    - 在进程的虚拟内存空间查找一块VMA
    - 对这块VMA进行映射
    - 如果设备驱动程序或文件系统的file_operations定义了mmap操作，则调用它
    - 将这个VMA插入到进程VMA链表中

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区)
    - 虚拟内存区(VMA)是Linux内核用来管理一个进程的地址空间的特殊区域的内核数据结构
    - 一个VMA代表一个进程的虚拟内存的一个同质区域：一个有相同许可标志和被相同对象支持的连续虚拟地址范围

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区)
    - 一个进程的内存区在/proc/[进程id]/maps描述
      - start-end: 该内存区的开始和结束地址(进程虚拟地址)
      - perm: 该内存区的保护位(p: 私有 s: 共享)
      - offset: 该内存区对应在文件中的偏移位置
      - major:minor: 该内存区映射的文件所在设备主次设备号
      - inode: 该内存区映射文件的inode号
      - image: 该内存区映射文件的文件名(含完整路径)

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区域)
    - vm_area_struct数据类型定义
      - 定义在<linux/mm_types.h>中

```
struct vm_area_struct {
    struct mm_struct *vm_mm; // 所处的地址空间
    unsigned long vm_start; // 虚拟内存区起始地址
    unsigned long vm_end; //虚拟内存区结束地址
    pgprot_t vm_page_prot; //访问权限
    unsigned long vm_flags; //虚拟内存区域标志
    ...
    struct vm_operations_struct *vm_ops; //操作VMA的函数集指针
    unsigned long vm_pgoff; // 偏移(页帧号)
    ...
};
```

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区域)
    - 虚拟内存区域标志
      - 设备I/O内存映射用标志:
        - » VM_IO: 标志该内存区域为内存映射的I/O内存区域，该标志会阻止系统将该区域包含在进程的存放转存(core dump)中
        - » VM_RESERVED: 标志该内存区域不能被换出

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区域)
    - vm_operations_struct数据类型定义
      - 定义在<linux/mm.h>中

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct *area); // 打开vma时被执行的函数
    void (*close)(struct vm_area_struct *area); // 关闭vma时被执行的函数
    ...
};
```

# Linux字符设备驱动

- mmap设备操作
  - VMA(虚拟内存区域)
    - 当用户进行mmap()系统调用后，尽管VMA在设备驱动文件操作结构体的mmap()设备操作被调用前就已经产生，内核不会调用VMA的open()函数，通常需要在驱动的mmap()设备操作中显式调用vma->vm_ops->open()

# Linux字符设备驱动

- mmap设备操作
  - 设备驱动中的mmap
    - 该操作将用户空间的一段地址关联到设备内存上；当用户读写这段用户空间地址时，实际上是在访问设备
    - mmap操作的任务就是建立虚拟地址到物理地址的页表
    - 大多数驱动程序都不需要提供设备内存到用户空间的映射能力，因为，对于串口等面向流的设备而言，实现这种映射毫无意义

# Linux字符设备驱动

- mmap设备操作
  - 设备驱动中的mmap
    - int (*mmap)(struct file *filp, struct vm_area_struct *area);
      - filp: 文件指针
      - area: 待映射的进程vma指针

# Linux字符设备驱动

- mmap设备操作
  - 设备驱动中建立页表的方法
    - 使用remap_pfn_range()函数一次建立所有页表
    - 使用nopage()每次为VMA建立一个页表

# Linux字符设备驱动

- mmap设备操作
  - 使用remap_pfn_range()函数一次建立所有页表
    - 实现在内核源代码mm/memory.c中
      - int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, unsigned size, pgprot_t prot);
        - » vma: 虚拟内存区域指针
        - » addr: 虚拟内存起始地址
        - » pfn: 要映射的物理地址所在页帧号，可以通过物理地址>>PAGE_SHIFT得到
        - » size: 待映射的内存区域的大小
        - » prot: vma的保护属性

# Linux字符设备驱动

- 混杂设备
  - 混杂设备是Linux中的一类特殊字符设备，内核用miscdevice数据结构描述
  - 所有混杂设备共享一个主设备号(10)，但是它们的次设备号不同
  - 内核维护一个所有混杂设备的链表，对设备进行访问时内核依据次设备号进行查找

# Linux字符设备驱动

- 混杂设备
  - miscdevice数据类型定义
    - 定义在<linux/miscdevice.h>中

```
struct miscdevice {
    int minor; // 次设备号
    const char *name; // 设备名称
    const struct file_operations *fops; // 设备操作集合
    struct list_head list; // 混杂设备数据链节点
    struct device *parent; // 可能存在的父设备指针
    struct device *this_device; // 设备的device结构体指针
};
```

# Linux字符设备驱动

- 混杂设备
  - 混杂设备操作
    - 实现在内核源代码driver/char/misc.c中
      - int misc_register(struct miscdevice *misc);
        - » 注册一个混杂设备
      - int misc_deregister(struct miscdevice *misc);
        - » 注销一个混杂设备

# Linux设备模型

- sysfs文件系统
  - sysfs自Linux2.6内核开始引入
  - sysfs是从ramfs发展而来的一种基于内存的文件系统；它提供一种导出内核数据结构及其属性，以及它们之间联系到用户空间的手段(参考documentation/filesystems/sysfs.txt)
  - sysfs把连接在系统上的设备、总线和类组织成分级的文件，使其从用户空间可以访问到

# Linux设备模型

- sysfs文件系统
  - sysfs文件系统目录
    - block
      - 系统中注册的所有块设备，每个子目录对应一个块设备；每个子目录下又包含一些属性文件，它们描述该块设备的各种属性，比如设备大小等；其中的loop块设备是使用文件来模拟的
    - bus
      - 系统中注册的所有总线，每个子目录对应一条总线；每条总线子目录下又包含名为devices和drivers的目录，前者包含系统中发现的属于该总线类型的设备，后者包含系统中发现的注册到该总线的所有驱动

# Linux设备模型

- sysfs文件系统
  - sysfs文件系统目录
    - class
      - 系统中注册的所有设备功能类，每个目录对应一个功能类
    - devices
      - 系统中注册的所有设备
    - firmware
      - 系统中的固件
    - fs
      - 系统中的文件系统

# Linux设备模型

- sysfs文件系统
  - sysfs文件系统目录
    - kernel
      - 内核中的配置参数
    - module
      - 系统中所有模块信息
    - power
      - 系统中的电源选项

# Linux设备模型

- kobject结构体
  - 该数据结构实现了基本的面向对象管理机制，它是构成Linux 2.6设备模型的核心数据结构
  - 该数据结构与sysfs文件系统联系紧密，在内核中注册的每个kobject对象对应sysfs文件系统中的一个目录

# Linux设备模型

- kobject结构体
  - 该数据结构类似于C++中的基类，所以，kobject经常被嵌入到其它数据结构(也就是容器)中；比如：device，bus，device_driver ，class等数据结构都是典型的容器；这些容器数据结构通过kobject连接起来，形成了一个树状结构

# Linux设备模型

- kobject结构体
  - kobject数据类型定义
    - 定义在<linux/kobject.h>中

```
struct kobject {
    const char *name; // 对象名称
    struct list_head entry; // 用于维护所有同类型的kobject对象的链表
    struct kobject *parent; // 指向可能存在的父对象的指针
    struct kset *kset; // 对象所属的集合
    struct kobj_type *ktype; // 对象类型
    struct sysfs_dirent *sd; // sysfs文件系统目录入口
    struct kref kref; // 对象引用计数
};
```

# Linux设备模型

- kobject结构体
  - kobject数据类型定义
    - 该结构体最后还包括几个位段
      - unsigned int state_initialized:1; // 对象初始化标志
      - unsigned int state_in_sysfs:1; // 对象在sysfs存在标志
      - unsigned int state_add_uevent_sent:1; // 对象添加“热插拔”事件发送标志
      - unsigned int state_remove_uevent_sent:1; // 对象删除“热插拔”事件发送标志

# Linux设备模型

- kobject结构体
  - kobj_type结构体
    - 该结构体记录了kobject对象的一些属性
    - 定义在<linux/kobject.h>中

```
struct kobj_type {
    void (*release)(struct kobject *kobj); // 用于释放kobject占用的资源，当kobject的引用计数为0时被调用
    struct sysfs_ops *sysfs_ops; // sysfs文件系统中属性文件操作方法集合
    struct attribute **default_attrs; // kobject对象属性指针数组
};
```

# Linux设备模型

- kobject结构体
  - sysfs_ops结构体
    - 定义在<linux/sysfs.h> 中

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *kobj, struct attribute
    *attr, char *buf); // 当用户读属性文件时该函数被调用，
    该函数将属性值存入buf中返回给用户
    ssize_t (*store)(struct kobject *kobj, struct attribute
    *attr, const char *buf, size_t count); // 当用户写属性文
    件时该函数被调用，用户存储用户传入的属性值
};
```

# Linux设备模型

- kobject结构体
  - attribute结构体
    - 定义在<linux/sysfs.h> 中

```
struct attribute {
    const char *name; // 该属性文件名称
    struct module *owner; // 该属性文件拥有者模块
    mode_t mode; // 该属性文件的文件保护位
};
```

# Linux设备模型

- kobject结构体
  - kobject操作
    - 实现在内核源代码lib/kobject.c中
    - 初始化对象
      - void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
        - » kobj: 待初始化的kobject对象
        - » ktype: 待初始化kobject对象的对象类型
    - 注册对象
      - int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, ...);
        - » kobj: 待注册的kobject对象
        - » parent: 待注册对象的父对象
        - » fmt: 格式化参数，常用于传递对象名称
        - » 返回: 成功返回0，失败返回负值

# Linux设备模型

- kobject结构体
  - kobject操作
    - 初始化并注册对象
      - int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype, struct kobject *parent, const char *fmt, ...);
        - » kobj: 待初始化的kobject对象
        - » ktype: 待初始化kobject对象的对象类型
        - » parent: 待注册对象的父对象
        - » fmt: 格式化参数，常用于传递对象名称
        - » 返回: 成功返回0，失败返回负值
    - 删除对象
      - void kobject_del(struct kobject *kobj);
        - » kobj: 待删除的kobject对象

# Linux设备模型

- kobject结构体
  - kobject操作
    - 增加引用计数
      - struct kobject *kobject_get(struct kobject *kobj);
        - » 将kobject对象的引用计数加1，同时返回该对象指针
        - » kobject: 待操作的kobject对象
        - » 返回: 返回操作后的kobject对象指针
    - 减少引用计数
      - void kobject_put(struct kobject *kobj);
        - » 将kobject对象的引用计数减1，如果引用计数降为0，则调用release()函数释放该对象
        - » kobject: 待操作的kobject对象

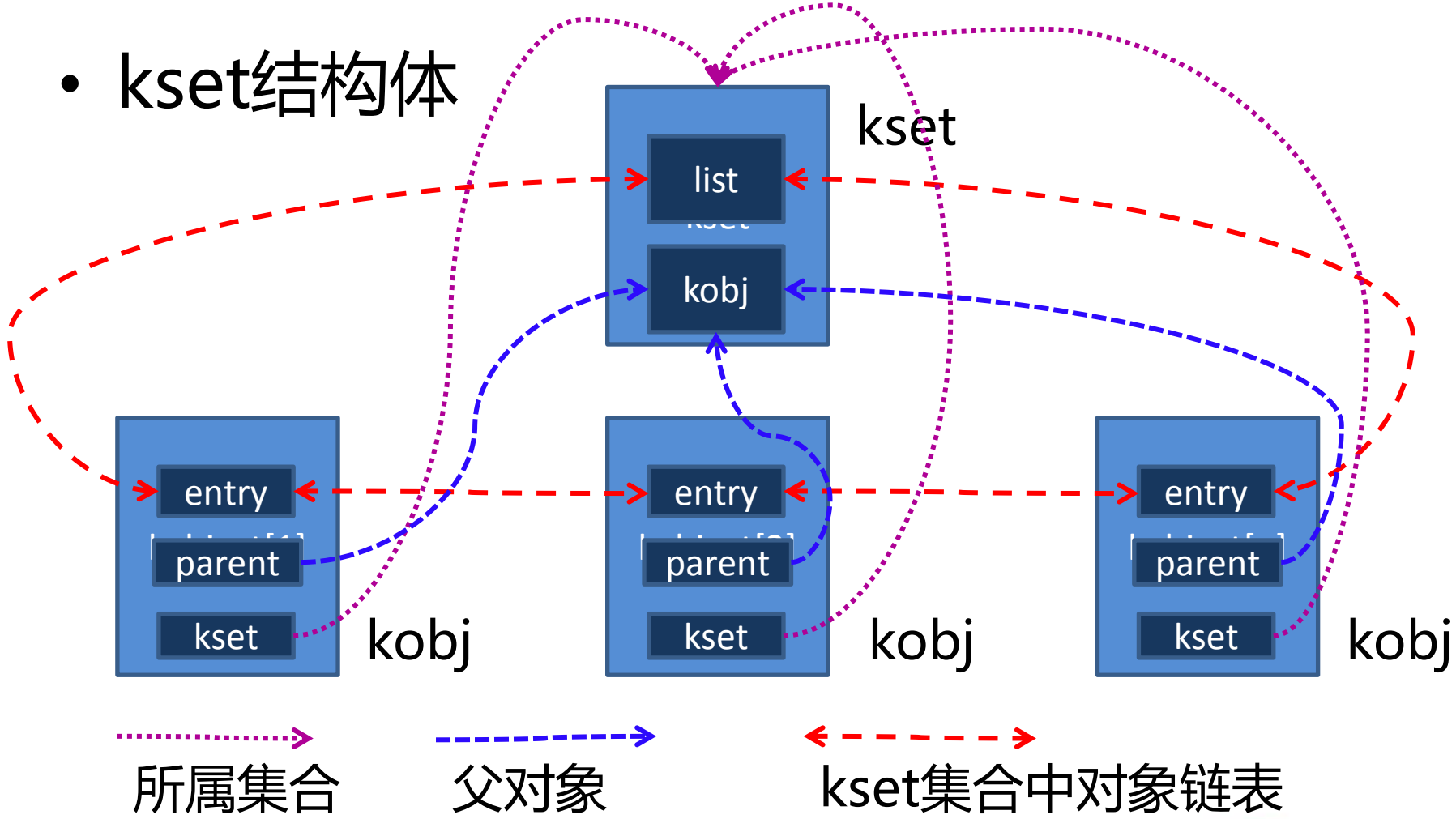
# Linux设备模型

- kset结构体
  - kset是具有相同类型的kobject的集合，在sysfs中体现为一个目录
  - kset数据类型定义
    - 定义在<linux/kobject.h>中

```
struct kset {
    struct list_head list; // 维护该集合中kobject对象链表头
    spinlock_t list_lock; // 操作该集合中的链表的保护锁
    struct kobject kobj; // 内嵌的kobject对象，表示集合本身
    struct kset_uevent_ops *uevent_ops; //处理热插拔事件的操作集合
};
```

# Linux设备模型

- kset结构体



# Linux设备模型

- kset结构体
  - kset_uevent_ops结构体
    - 定义在<linux/kobject.h>中

```
struct kset_uevent_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj); // 该操作决定是否将事件传递到用户空间；如果操作返回0，将不传递事件
    const char *(*name)(struct kset *kset, struct kobject *kobj); // 该操作将字符串传递给用户空间的热插拔处理程序
    int (*uevent)(struct kset *kset, struct kobject *kobj, struct kobj_uevent_env *env); // 该操作将用户空间需要的参数添加到环境变量中
};
```

# Linux设备模型

- kset结构体
  - kset操作
    - 实现在内核源代码lib/kobject.c中
    - 初始化集合
      - void kset_init(struct kset *kset);
        - » kset: 待初始化的kset集合
    - 注册集合
      - int kset_register(struct kset *kset);
        - » kset: 待注册到内核的kset集合
        - » 返回: 成功返回0，失败返回负值

# Linux设备模型

- kset结构体
  - kset操作
    - 创建并注册集合
      - struct kset * kset_create_and_add(const char *name, struct kset_uevent_ops *u, struct kobject *parent_kobj);
        - » name: kset集合名称
        - » u: kset集合中处理热插拔事件的操作集合
        - » parent_kobj: 可能存在的父对象
        - » 返回: 成功创建的kset集合指针，失败返回NULL
    - 注销集合
      - void kset_unregister(struct kset *kset);
        - » kset: 待注销的kset对象

# Linux设备模型

- Linux设备模型元素
  - 总线
  - 设备
  - 驱动

# Linux设备模型

- 总线
  - 总线是处理器与设备之间的数据传输通道
  - 在设备模型中，所有的设备都通过总线相连，甚至是内部的虚拟“platform”总线
  - 在Linux设备模型中，总线由bus_type结构体表示

# Linux设备模型

- 总线
  - bus_type结构体
    - 定义在<linux/device.h>中

```
struct bus_type {
    const char *name; // 总线名称
    struct bus_attribute *bus_attrs; // 总线属性
    struct device_attribute *dev_attrs; // 设备属性
    struct driver_attribute *drv_attrs; // 驱动属性
    int (*match)(struct device *dev, struct device_driver *drv);
    int (uevent)(struct device *dev, struct kobj_uevent_env *env);
    ...
    struct pm_ext_ops *pm; // 电源管理操作集合
    struct bus_type_private *p;
};
```

# Linux设备模型

- 总线
  - bus_type结构体
    - match成员
      - 该函数用于判断指定驱动程序是否能驱动指定的设备；成功返回非0值
      - 当一个新设备或者新驱动添加到这条总线时，该函数会被调用
    - uevent成员
      - 该函数允许总线在内核为用户空间产生热插拔事件之前为系统添加环境变量

# Linux设备模型

- 总线
  - 总线操作
    - 实现在内核源代码drivers/base/bus.c中
    - 总线注册
      - int bus_register(struct bus_type *bus);
        - » 该操作将新的系统总线添加进内核，成功之后，在sysfs文件系统的/sys/bus下能看到新注册的总线
        - » bus: 待注册的总线
        - » 返回值: 成功返回0，失败返回负值
    - 总线注销
      - void bus_unregister(struct bus_type *bus);
        - » 该操作从内核中注销一条已经注册的系统总线
        - » bus: 待注销的总线

# Linux设备模型

- 总线
  - bus_attribute结构体
    - 定义在<linux/device.h>中

```
struct bus_attribute {
    struct attribute attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char *buf,
    size_t count);
};
```

# Linux设备模型

- 总线
  - bus_attribute操作
    - 创建总线属性
      - int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
        - » bus: 待创建属性所属总线
        - » attr: 待创建总线属性
        - » 返回: 成功返回0，失败返回负值
    - 移除总线属性
      - void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
        - » bus: 待移除属性所属总线
        - » attr: 待移除总线属性

# Linux设备模型

- 总线
  - device_attribute结构体
    - 定义在<linux/device.h>中

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct
    device_attribute *attr, char *buf);
    ssize_t (*store)(struct device *dev, struct
    device_attribute *attr, const char *buf, size_t count);
};
```

# Linux设备模型

- 总线
  - device_attribute操作
    - 创建设备属性
      - int device_create_file(struct device *device, struct device_attribute *entry);
        - » device: 待创建属性所属设备
        - » entry: 待创建设备属性
        - » 返回: 成功返回0，失败返回负值
    - 移除设备属性
      - void device_remove_file(struct device *dev, struct device_attribute *attr);
        - » dev: 待移除属性所属设备
        - » attr: 待移除设备属性

# Linux设备模型

- 总线
  - driver_attribute结构体
    - 定义在<linux/device.h>中

```
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *driver, char *buf);
    ssize_t (*store)(struct device_driver *driver, const char
    *buf, size_t count);
};
```

# Linux设备模型

- 总线
  - driver_attribute操作
    - 创建驱动属性
      - int driver_create_file(struct device_driver *driver, struct driver_attribute *attr);
        - » driver: 待创建属性所属驱动
        - » attr: 待创建驱动属性
        - » 返回: 成功返回0，失败返回负值
    - 移除驱动属性
      - void driver_remove_file(struct device_driver *driver, struct driver_attribute *attr);
        - » driver: 待移除属性所属驱动
        - » attr: 待移除驱动属性

# Linux设备模型

- 总线
  - bus_type_private结构体
    - 定义在内核源代码drivers/base/base.h中

```
struct bus_type_private {
    struct kset subsys;  // 总线子系统
    struct kset *drivers_kset; // 总线包含的驱动对象集合
    struct kset *devices_kset; // 总线包含的设备对象集合
    struct klist klist_devices; // 总线包含的设备数据链
    struct klist klist_drivers; // 总线包含的驱动数据链
    struct blocking_notifier_head bus_notifier; // 总线通知链
    unsigned int drivers_autoprobe:1; // 驱动自动侦测标志
    struct bus_type *bus; // 指向包含该总线结构体的指针
};
```

# Linux设备模型

- 总线
  - 属性相关宏
    - 定义在<linux/sysfs.h>中

```
#define __ATTR(_name, _mode, _show, _store) {
    .attr = {.name = __stringify(_name), .mode = _mode},
    .show = _show,
    .store = _store,
}
```

# Linux设备模型

- 总线
  - 属性相关宏
    - 定义在<linux/device.h>中
      - #define BUS_ATTR(_name, _mode, _show, _store)
        struct bus_attribute bus_attr_##_name =
        __ATTR(_name, _mode, _show, _store)
      - #define DEVICE_ATTR(_name, _mode, _show, _store)
        struct device_attribute dev_attr_##_name =
        __ATTR(_name, _mode, _show, _store)
      - #define DRIVER_ATTR(_name, _mode, _show, _store)

      struct driver_attribute driver_attr_##_name =
        __ATTR(_name, _mode, _show, _store)

# Linux设备模型

- 设备
  - 总线也是设备，必须按设备注册
  - 在Linux设备模型中，设备由device结构体表示

# Linux设备模型

- 设备
  - device结构体
    - 定义在<linux/device.h>中

```
struct device {
    struct klist klist_children; // 子设备数据链
    struct klist_node knode_parent; // 对应设备链表节点
    struct klist_node knode_driver; // 对应驱动链表节点
    struct klist_node knode_bus; // 对应总线链表节点
    struct device *parent; // 指向父设备指针
```

# Linux设备模型

- 设备
  - device结构体

```
struct kobject kobj; // 设备对象
char bus_id[BUS_ID_SIZE]; // 总线上唯一标识该设备的字符串
const char *init_name; // 原始初始名称
struct device_type *type; // 设备类型
unsigned uevent_suppress:1; // uevent禁止位

struct semaphore sem; // 同步驱动调用的信号量

struct bus_type *bus; // 设备所在总线
struct device_driver *driver; // 管理该设备的驱动
void *driver_data; // 该设备的驱动使用的私有数据
void *platform_data; // 平台私有数据
```

# Linux设备模型

- 设备
  - device结构体

```
    ...

    // 体系结构相关内容
    struct dev_archdata archdata; // DMA反弹信息
    spinlock_t devres_lock; // 设备资源操作保护锁
    struct list_head devres_head; // 设备资源数据链
    struct klist_node knode_class; // 对应设备类节点
    struct class * class; // 设备所属类
    dev_t devt; // 设备号
    struct attribute_group **groups; // 可选的设备属性组
    void  (*release)(struct device *dev); // 释放设备操作
};
```

# Linux设备模型

- 设备
  - device操作
    - 实现在内核源代码drivers/base/core.c中
    - 设备初始化
      - void device_initialize(struct device *dev);
        - » dev: 待初始化的设备
    - 设备注册
      - int device_register(struct device *dev);
        - » dev: 待注册的设备
        - » 返回: 成功返回0，失败返回负值
    - 设备注销
      - void device_unregister(struct device *dev)
        - » dev: 待注销的设备

# Linux设备模型

- 设备
  - device_type结构体
    - 定义在<linux/device.h>中

```
struct device_type {
    const char *name; // 设备名称
    struct attribute_group **groups; // 可选的设备属性组
    int (*uevent)(struct device *dev, struct kobj_uevent_env
    *env);
    void (*release)(struct device *dev);
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);
    struct pm_ops *pm;
};
```

# Linux设备模型

- 驱动
  - 在Linux设备模型中，驱动由device_driver结构体表示

# Linux设备模型

- 驱动
  - device_driver结构体
    - 定义在<linux/device.h>中

```
struct device_drvier {
    const char *name; // 驱动名称
    struct bus_type *bus; // 驱动所属总线

    struct module *owner; // 驱动模块拥有者
    const char *mod_name; // 通常不用
```

# Linux设备模型

- 驱动
  - device_driver结构体
    - 定义在<linux/device.h>中

```
int (*probe)(struct device *dev); // 设备侦测函数
int (*remove)(struct device *dev); // 设备移除函数
int (*shutdown)(struct device *dev); // 设备关闭函数
int (*suspend)(struct device *dev, pm_message_t state); // 设备挂起函数
int (*resume)(struct device *dev); // 设备恢复函数

struct attribute_group **groups; // 可选的驱动属性组

struct pm_ops *pm;
struct driver_private *p;
};
```

# Linux设备模型

- 驱动
  - device_driver操作
    - 实现在内核源代码drivers/base/driver.c中
    - 驱动注册
      - int driver_register(struct device_driver *drv);
        - » drv: 待注册的驱动
        - » 返回: 成功返回0，失败返回负值
    - 驱动注销
      - void driver_unregister(struct device_driver *drv);
        - » drv: 待注销的驱动

# Linux设备模型

- 驱动
  - driver_private结构体
    - 定义在内核源代码drivers/base/base.h中

```
struct driver_private {
    struct kobject kobj; // 驱动对象
    struct klist klist_devices; // 驱动管理的设备数据链
    struct klist_node knode_bus; // 驱动对应总线中的节点
    struct module_kobject *mkobj; // 模块对象
    struct device_driver *driver; // 驱动指针
};
```

# Linux设备模型

- 类
  - 类是一个设备的高层视图，它抽象出了底层的实现细节，从而允许用户空间使用设备所提供的功能，而不用关心设备是如何连接和工作的
  - 类成员通常由上层代码所控制，而无需驱动明确支持，但有些情况下驱动也需要直接处理类

# Linux设备模型

- 类
  - 几乎所有的类都显示在/sys/class目录中；出于历史的原因，有一个例外：块设备显示在/sys/block目录中
  - 在许多情况，类子系统是向用户空间导出信息的最好方法
  - 当类子系统创建一个类时，它将完全拥有这个类，根本不用担心哪个模块拥有那些属性，而且信息的表示也比较友好

# Linux设备模型

- 类
  - 为了管理类，驱动程序核心导出了一些接口，其目的之一是提供包含设备号的属性以便自动创建设备节点，所以udev的使用离不开类
  - 类函数和结构与设备模型的其他部分遵循相同的模式，所以真正崭新的概念是很少的
  - 在Linux系统中，类由class结构体表示

# Linux设备模型

- 类
  - class结构体定义
    - 定义在<linux/device.h>中

```
struct class {
    const char *name; // 类名称
    struct module *owner; // 类模块拥有者

    struct class_attribute *class_attrs; // 类属性
    struct device_attribute *dev_attrs; // 类设备属性
    struct kobject *dev_kobj; // 类对象
```

# Linux设备模型

- 类
  - class结构体定义
    - 定义在<linux/device.h>中

```
    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env
    *env); // 类中设备热插拔产生时添加环境变量的函数

    void (*class_release)(struct class *class); // 类释放函数
    void (*dev_release)(struct device *dev); // 类设备释放函数

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    struct pm_ops *pm;
    struct class_private *p;
};
```

# Linux设备模型

- 类
  - class操作
    - 类创建并注册
      - 定义在<linux/device.h>中
      - struct class *class_create(struct module *owner, const char *name);
        - » owner: 待创建类所属模块
        - » name: 待创建类名称
        - » 返回: 成功返回创建的类指针，失败返回NULL
    - 类销毁
      - 实现在内核源代码drivers/base/class.c中
      - void class_destroy(struct class *cls);
        - » cls: 待销毁类指针

# Linux设备模型

- 类
  - class_attribute结构体
    - 定义在<linux/device.h>中

```
struct class_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class *class, char *buf);
    ssize_t (*store)(struct class *class, const char *buf,
    size_t count);
};
```

# Linux设备模型

- 类
  - class_attribute操作
    - 创建类属性
      - int class_create_file(struct class *class, const struct class_attribute *attr);
        - » class: 待创建属性所属类
        - » attr: 待创建类属性
        - » 返回: 成功返回0，失败返回负值
    - 移除类属性
      - void class_remove_file(struct class *class, const struct class_attribute *attr);
        - » class: 待移除属性所属类
        - » attr: 待移除类属性

# Linux设备模型

- 类
  - 属性相关宏
    - 定义在<linux/device.h>中
      - #define CLASS_ATTR(_name, _mode, _show, _store)
        struct class_attribute class_attr_##_name =
        __ATTR(_name, _mode, _show, _store)

# Linux设备模型

- 类
  - class_private结构体
    - 定义在内核源代码drivers/base/base.h中

```
struct class_private {
    struct kset class_subsys; // 类子系统
    struct klist class_devices; // 类设备数据链
    struct list_head class_interfaces; // 类接口数据链
    struct kset class_dirs; // 设备类目录
    struct mutex class_mutex; // 类操作互斥体
    struct class *class; // 类指针
};
```

# Linux设备模型

- 类
  - 类接口
    - 它是类子系统中特有的， Linux 设备模型的其它部分找不到的附加概念，称为“接口”，可将它理解为一种设备加入或离开类时获得信息的触发机制
    - 在Linux系统中，类接口由class_interface结构体表示

# Linux设备模型

- 类
  - 类接口
    - class_interface结构体
      - 定义在<linux/device.h>中

```
struct class_interface {
    struct list_head node; // 类子系统接口数据链节点
    struct class *class; // 接口所属类

    int (*add_dev) (struct device *, struct class_interface
    *);
    void (*remove_dev) (struct device *, struct
    class_interface *);
};
```

# Linux设备模型

- 类
  - 类接口
    - class_interface结构体
      - add_dev成员
        - » 当一个类设备被加入到在class_interface结构中指定的类时，将调用接口的add_dev()函数，进行一些设备需要的额外设置，通常是添加更多属性或其它的一些工作
      - remove_dev成员
        - » 当设备从类中删除时，将调用remove_dev()函数来进行必要的清理

# Linux设备模型

- 类
  - 类接口
    - class_interface操作
      - 实现在内核源代码drivers/base/class.c中
      - 类接口注册
        - » int class_interface_register(struct class_interface *class_intf);
          - class_intf: 待注册的类接口
      - 类接口注销
        - » void class_interface_unregister(struct class_interface *class_intf);
          - class_intf: 待注销的类接口

# Linux设备模型

- platform总线
  - platform总线是Linux 2.6中引入的一种虚拟总线，主要用来管理CPU的片上资源，具有更好的移植性，因此在Linux 2.6中，很多驱动都用platform重新改写

# Linux设备模型

- platform总线
  - platform总线为Linux 2.6内核引入了一套新的驱动管理和注册机制: platform_device和platform_driver
  - Linux中大部分的设备驱动，都可以使用这套机制，设备用platform_device表示，驱动用platform_driver进行注册

# Linux设备模型

- platform总线
  - 和传统的设备驱动机制（通过driver_register函数进行注册）相比，一个十分明显的优势在于platform机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过platform_device提供的标准接口进行申请并使用；这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性（这些标准接口是安全的）

# Linux设备模型

- platform总线
  - 基本上，只要和内核本身运行依赖性不大的外围设备（换句话说，只要不在内核运行所需的一个最小系统之内的设备），相对独立的，拥有各自独自的资源（内存地址、中断等），都可以用platform_driver实现（如：lcd、usb、uart等驱动）；而timer、irq等最小系统之内的设备则最好不用platfrom_driver机制，实际上内核实现就是这样做的

# Linux设备模型

- platform总线
  - platform总线定义
    - 定义在内核源代码drivers/base/platform.c中
      - platform总线

```
struct bus_type platform_bus_type = {
    .name           = "platform", // 总线名称
    .dev_attrs      = platform_dev_attrs, // 总线属性
    .match          = platform_match, // 驱动配备函数
    .uevent         = platform_uevent, // 热插拔事件发出
    前的环境变量处理函数
    .pm    = PLATFORM_PM_OPS_PTR, // 电源管理函数集
};
```

# Linux设备模型

- platform总线
  - platform总线定义
    - 定义在内核源代码drivers/base/platform.c中
      - platform总线设备
        - » 表示platform总线本身的设备

```
struct device platform_bus = {
    .bus_id                 = "platform", // 总线上唯一
    标识该设备的字符串
};
```

# Linux块设备驱动

- 块设备
  - 块设备是Linux设备驱动一个重要的部分，需要和字符型加以区分，字符型设备不带有缓冲，操作直接与实际的设备相连，直接进行操作，而对于常见的块设备，通常都带有一个缓冲区，也就是数据块
  - 块设备中数据的操作单元是块，数据的输入输出都是通过这些缓冲区来完成的

# Linux块设备驱动

- 块设备
  - 通常意义的块设备主要指的是硬盘、SD卡、flash等设备， 早期的块设备主要指的是硬盘，后来很多的块设备的驱动都是从早期硬盘驱动衍化而来的

# Linux块设备驱动

- 块设备注册与注销
  - 块设备驱动的第一个任务就是将自己注册到内核中去
  - 定义在<linux/fs.h>中
  - 实现在内核源代码block/genhd.c中

# Linux块设备驱动

- 块设备注册与注销
  - 注册
    - int register_blkdev(unsigned int major, const char *name);
      - major: 块设备希望使用的主设备号，如果为0，内核会自动分配一个新的主设备号
      - name: 块设备名称(出现在/proc/devices中)
      - 返回: 如果major不为0，成功返回0，失败返回负值；如果major为0，返回内核为设备自动分配的主设备号

# Linux块设备驱动

- 块设备注册与注销
  - 注销
    - void unregister_blkdev(unsigned int major, const char *name);
      - major: 待注销块设备的主设备号
      - name: 待注销块设备的设备名称

# Linux块设备驱动

- 块设备相关结构体
  - block_device_operations结构体
    - 类似于字符设备驱动中的file_operations结构体
    - file_operations结构体是联系虚拟文件系统的文件操作和具体文件系统的文件操作的桥梁
    - block_device_operations结构体是联系抽象的块设备操作与具体块设备操作之间的桥梁
    - 具体的块设备是由主设备号唯一确定的，主设备号也唯一确定了一个具体的block_device_operations结构体
    - 该结构体描述对块设备操作的集合
    - 定义在<linux/blkdev.h>中

# Linux块设备驱动

- 块设备相关结构体
  - block_device_operations结构体

```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t); // 打开
    int (*release) (struct gendisk *, fmode_t); // 关闭
    int (*locked_ioctl)(...);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned
    long); // ioctl控制
    int (*compat_ioctl)(...);
    int (*direct_access)(...);
    int (*media_changed) (struct gendisk *); // 介质改变
    int (*revalidate_disk) (struct gendisk *); // 使介质有效
    int (*getgeo)(struct block_device *, struct hd_geometry *); //
    填充驱动器信息
    struct module *owner; // 模块拥有者
};
```

# Linux块设备驱动

- 块设备相关结构体
  - block_device结构体
    - 类似于字符设备驱动中的cdev结构体
    - 该结构体是内核中描述块设备的结构体
    - inode结构体中包含一个该结构体指针
    - 该结构体中包含一个gendisk结构体指针
    - 定义在<linux/fs.h>中

# Linux块设备驱动

- 块设备相关结构体
  - gendisk结构体
    - 在Linux内核中，使用该结构体来表示一个独立的磁盘设备（或分区）
    - 定义在<linux/genhd.h>中

# Linux块设备驱动

- 块设备相关结构体
  - gendisk结构体

```
struct gendisk {
    int major; // 主设备号
    int first_minor; // 起始次设备号
    int minors; // 次设备数目
    char disk_name[DISK_NAME_LEN]; // 磁盘的标准名称
    struct disk_part_table *part_tbl; // 整个磁盘分区表
    struct hd_struct part0; // 第一个分区信息

    struct block_device_operations *fops; // 块设备操作集合
    struct request_queue *queue; // 块设备请求队列
    void *private_data; // 私有数据

    ...
};
```

# Linux块设备驱动

- 磁盘操作
  - 定义在<linux/genhd.h>中
  - 实现在内核源代码block/genhd.c中
  - 分配磁盘
    - 该函数分配一个gendisk结构体，并返回指针
    - struct gendisk *alloc_disk(int minors);
      - minors: 该磁盘使用的次设备的数目，一般就是指分区数量；该值指定后不能修改
      - 返回: 如果成功，返回分配到的gendisk结构体指针，失败返回NULL

# Linux块设备驱动

- 磁盘操作
  - 添加磁盘
    - 该函数将一个磁盘添加到内核，应该在磁盘准备好可以被操作之后再调用
    - void add_disk(struct gendisk *disk);
      - disk: 待添加到系统的磁盘
  - 删除磁盘
    - 该函数将一个注册到内核的磁盘删除掉
    - void del_gendisk(struct gendisk *disk);
      - disk: 待从系统删除的磁盘

# Linux块设备驱动

- 磁盘操作
  - 磁盘容量操作
    - 获取磁盘容量
      - sector_t get_capacity(struct gendisk *disk);
        - » disk: 待获取容量的磁盘
        - » 返回: 磁盘容量(实际为扇区数)
    - 设置磁盘容量
      - void set_capacity(struct gendisk *disk, sector_t size);
        - » disk: 待设置容量的磁盘
        - » size: 待设置磁盘的容量(实际为扇区数)

# Linux块设备驱动

- 磁盘操作
  - 磁盘引用计数操作
    - 这两个操作用来操作引用计数，驱动应该是从来不需要调用它们
    - 因为，通常对于del_gendisk操作的调用去掉了最后一个gendisk的最终引用，但是不保证这样
    - 增加引用计数
      - struct kobject *get_disk(struct gendisk *disk);
        - » disk: 待增加引用计数的磁盘
        - » 返回: 磁盘的kobject对象指针
    - 减少引用计数
      - void put_disk(struct gendisk *disk);
        - » disk: 待减少引用计数的磁盘

# Linux块设备驱动

- 块请求队列
  - 在Linux块设备驱动中，使用request结构体来描述等待进行的块I/O请求，并使用request_queue结构体来描述一个块I/O请求队列
  - 请求处理函数是块设备驱动程序中应该完成的最重要的函数

# Linux块设备驱动

- 块请求队列
  - 请求处理函数
    - 该函数不能由驱动自己调用
    - 只有当内核认为是时候让驱动处理对设备的读/写等操作时，它才会调用这个函数
    - 该函数可以在没有完成请求队列中的所有请求的情况下返回，甚至它一个请求不完成都可以返回
    - 对大部分设备而言，一般会在请求处理函数中处理完所有请求后才返回

# Linux块设备驱动

- 块请求队列
  - 请求处理函数的主要任务
    - 提取请求
    - 检查请求的有效性
    - 执行请求
    - 清除已经处理过的请求
  - 整个过程循环执行，一般会在处理完整个块请求队列中的所有请求后返回
  - 函数原型：
    - typedef void (request_fn_proc) (struct request_queue *q);

# Linux块设备驱动

- 块请求队列
  - request结构体
    - 定义在<linux/blkdev.h>中

```
struct request {
    struct list_head queuelist; // 请求队列数据链
    ...
    struct request_queue *q; // 请求所属的请求队列
    ...
    sector_t sector; // 待传输的起始扇区
    sector_t hard_sector; // 待完成的起始扇区
    unsigned long nr_sectors; // 待传输的扇区数
    unsigned long hard_nr_sectors; // 待完成的扇区数
    unsigned int current_nr_sectors; // 当前I/O操作中待传输扇区数
    unsigned int hard_cur_sectors; // 当前I/O操作中待完成的扇区数
```

# Linux块设备驱动

- 块请求队列
  - request结构体
    - 定义在<linux/blkdev.h>中

```
struct bio *bio; // 请求的bio结构体指针
struct bio *biotail; // 请求的bio结构体指针(尾部)
...
unsigned short nr_phys_segments; // 请求在物理内存中占据的不连续的段的数目
...
char *buffer; // 传输数据缓冲区指针(内核虚拟地址)
...
int ref_count; // 请求引用计数
...
};
```

# Linux块设备驱动

- 块请求队列
  - request_queue结构体
    - 定义在<linux/blkdev.h>中

```
struct request_queue {
    struct list_head queue_head; // 请求数据链
    ...
    request_fn_proc *request_fn; // 请求处理函数
    make_request_fn *make_request_fn; //  “制造请求”函数
    ...
    unsigned long bounce_pfn; // DMA反弹物理页框号
    gfp_t bounce_gfp; // DMA反弹标志
    ...
    unsigned long nr_request; // 最大请求数量
    ...
```

# Linux块设备驱动

- 块请求队列
  - request_queue结构体
    - 定义在<linux/blkdev.h>中

```
unsigned int max_sectors; // 最大扇区数
unsigned int max_hw_sectors; // 最大设备扇区数
unsigned short max_phys_segments; // 最大驱动内存段数
unsigned short max_hw_segment; // 最大设备内存段数
unsigned short hardsect_size; // 硬件扇区尺寸
unsigned int max_segment_size; // 最大段尺寸

unsigned long seg_boundary_mask; // 段边界掩码
...
unsigned int dma_alignment; // DMA传送的内存对齐限制
...
};
```

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 定义在<linux/blkdev.h>中
    - 实现分别在内核源代码block目录下的blk-core.c、elevator.c和blk-settings.c中
    - 内核源代码block目录中包含电梯调度算法相关队列处理函数

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 创建并初始化请求队列
      - struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
        - » rfn: 队列请求处理函数
        - » lock: 队列操作的保护锁
        - » 返回: 返回创建并初始化后的请求队列，失败返回NULL

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 提取请求
      - 该函数使用指定的电梯调度算法提取请求队列中的下一个要处理的请求
      - 该操作不会从请求队列中清除请求
      - struct request *elv_next_request(struct request_queue *q);
        - » q: 待提取请求的请求队列
        - » 返回: 从队列中提取的下一个请求，如果请求队列已经为空则返回NULL

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 完成请求
      - 该函数应该在一个请求被处理完之后调用，向上层报告请求完成状态，会导致指定的请求被从请求队列中剥离
      - void end_request(struct request *req, int uptodate);
        - » req: 已经处理过的请求
        - » uptodate: 向上层报告请求完成情况，1表示成功完成请求处理，0表示请求处理失败

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 清除请求队列
      - 该函数会清除请求队列中的所有请求
      - void blk_cleanup_queue(struct request_queue *q);
        - » q: 待清除的请求队列

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 请求出列
      - 该函数从请求队列中出列一个请求
      - void blkdev_dequeue_request(struct request *req);
        - » req: 待从请求队列中移除的请求
    - 请求入列
      - 该函数使用电梯调度算法归还一个请求到请求队列
      - void elv_requeue_request(struct request_queue *q, struct request *req);
        - » q: 待归还请求的请求队列
        - » req: 待归还的请求

# Linux块设备驱动

- 块请求队列
  - request_queue操作
    - 如果块设备处于暂时不能处理等候的请求的状态，应该通知上层停止调用块设备驱动的请求处理函数，并在块设备恢复到可处理等候的请求的状态时重新通知上层可以调用请求处理函数
    - 启动请求队列
      - void blk_start_queue(struct request_queue *q);
        - » q: 待启动的请求队列
    - 停止请求队列
      - void blk_stop_queue(struct request_queue *q);
        - » q: 待停止的请求队列

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - DMA操作设置
      - 通知内核该设备可进行DMA操作的最高物理地址；如果一个请求超出限制，一个反弹缓冲区将被用来给该操作，当然，应该尽可能避免使用
      - void blk_queue_bounce_limit(struct request_queue *q, u64 dma_addr);
        - » q: 待设置的请求队列
        - » dma_addr: 可以是任何可能的值，也可使用预定义符号：

          BLK_BOUNCE_HIGH: 使用反弹缓冲给高内存页，该项为默认值

          BLK_BOUNCE_ISA: 只可DMA到16MB的ISA区域

          BLK_BOUNCE_ANY: 可以进行DMA到任何地址

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 最大扇区数设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - void blk_queue_max_sectors(struct request_queue *q, unsigned int max_sectors);
        - » q: 待设置的请求队列
        - » max_sectors: 设备能满足的以扇区方式表示的任一请求的最大大小，缺省255

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 最大驱动段数设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - void blk_queue_max_phys_segments(struct request_queue *q, unsigned short max_segments);
        - » q: 待设置的请求队列
        - » max_segments: 在一个请求中最多可以包含多少物理段(系统内存中不相邻的区)，实际是驱动最大能处理多少段，缺省128

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 最大设备段数设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - void blk_queue_max_hw_segments(struct request_queue *q, unsigned short max_segments);
        - » q: 待设置的请求队列
        - » max_segments: 在一个请求中最多可以包含多少物理段(系统内存中不相邻的区)，实际是设备最大能处理多少段，缺省128

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 硬件扇区大小设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - 该函数告知内核设备的硬件扇区大小，所有内核产生的请求是这个大小的倍数并且被正确对齐；但是，所有块层和块设备驱动之间的通讯仍然以512字节扇区方式来处理
      - void blk_queue_hardsect_size(struct request_queue *q, unsigned short size);
        - » q: 待设置的请求队列
        - » size: 硬件扇区字节数

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 最大段尺寸设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - void blk_queue_max_segment_size(struct request_queue *q, unsigned int max_size);
        - » q: 待设置的请求队列
        - » max_size: 设置任一请求的段最大可能是多大字节，缺省是65536字节

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - 段边界设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - 该函数告诉内核设备无法处理跨越一个特殊大小内存边界的请求
      - void blk_queue_segment_boundary(struct request_queue *q, unsigned long mask);
        - » q: 待设置的请求队列
        - » mask: 边界掩码，比如处理跨4M边界有困难，传递0x003FFFFF，缺省为0xFFFFFFFF

# Linux块设备驱动

- 块请求队列
  - 请求队列参数设置
    - DMA对齐设置
      - 该函数用于设置参数，描述可被设备满足的请求
      - 该函数告诉内核设备施加于DMA传送的内存对齐的限制，当请求被创建时，有给定的对齐，这个对齐匹配请求长度
      - void blk_queue_dma_alignment(struct request_queue *q, int mask);
        - » q: 待设置的请求队列
        - » mask: 缺省掩码为0x01FF，它导致所有的请求被对齐到512字节边界

# Linux块设备驱动

- 块设备驱动模块加载
  - 块设备驱动模块加载通常应该完成如下任务：
    - 注册块设备驱动
    - 分配、初始化请求队列，绑定请求队列和请求函数
    - 分配、初始化gendisk，给gendisk的major、fops、queue等成员赋值，最后添加gendisk

# Linux块设备驱动

- 块设备驱动模块卸载
  - 块设备驱动模块卸载通常应该完成如下任务：
    - 清除请求队列
    - 删除gendisk和对gendisk的引用
    - 删除对块设备的引用，注销块设备驱动

# Linux块设备驱动

- 块设备操作
  - 块设备打开与释放
    - 块设备的open()与release()函数并非必须，一个简单的块设备驱动可以不提供这两个函数
    - open()函数和字符设备驱动的open()类似，都有inode和file结构体作为参数；当一个节点引用一个块设备时，inode->i_bdev->bd_disk包含一个指向关联的gendisk结构体的指针
    - 在一个真实的块设备硬件的驱动中，open()和release()函数还应该设置驱动和硬件的状态，这些工作可能包含启停磁盘、加锁一个可移除设备和分配DMA缓冲等

# Linux块设备驱动

- 块设备操作
  - 块设备ioctl控制命令操作
    - 通常，高层的块设备驱动代码处理了绝大多数ioctl控制命令
    - 因此，具体的块设备驱动代码中通常不再需要实现很多ioctl命令

# Linux块设备驱动

- 块设备操作
  - 块设备可移动介质操作
    - media_changed()操作
      - 该函数用于查看是否有介质已经改变；如果改变已经发生，它应该返回一个非0值
    - revalidate_disk()操作
      - 该函数在介质被改变后调用，它的任务是做任何可能需要的事情来准备驱动对新介质的操作；该函数被调用之后，内核试图重新读分区表并且启动这个磁盘设备

# Linux块设备驱动

- 块设备操作
  - 块设备getgeo()操作
    - 该函数根据磁盘驱动器的几何信息填充一个hd_geometry结构体
    - hd_geometry结构体包含磁盘驱动器的磁头、扇区、柱面等信息

# Linux块设备驱动

- 块设备操作
  - 块设备getgeo()操作
    - hd_geometry结构体
      - 定义在<linux/hdreg.h>中

```
strcut hd_geometry {
    unsigned char heads; // 磁头数
    unsigned char sectors; // 扇区数
    unsigned short cylinders; // 柱面数
    unsigned long start; // 起始扇区号
};
```

# Linux块设备驱动

- 块设备操作
  - 块设备getgeo()操作
    - 硬磁盘结构
      - 磁头
        - » 硬盘由很多盘片组成，每个盘片的每个面都有一个读写磁头；如果有n个盘片，对应有2n个磁头(heads)
      - 扇区
        - » 每个盘片上的磁道被划分成几十个扇区(sectors)，每个扇区通常的容量是512字节
      - 柱面
        - » 每个盘片被划分成若干个同心圆磁道(逻辑上的)，所有盘片的磁道形成以主轴为轴心的柱面(cylinders)

# Linux块设备驱动

- 块设备操作
  - 块设备getgeo()操作
    - 磁盘容量
      - 磁盘容量 = 磁头数 * 扇区数 * 柱面数 * 512字节
      - 例如: 50GB硬盘容量
        - » 53.6GB = 53687091200 bytes = heads(255) * sectors(63) * cylinders(6527) * 512 bytes

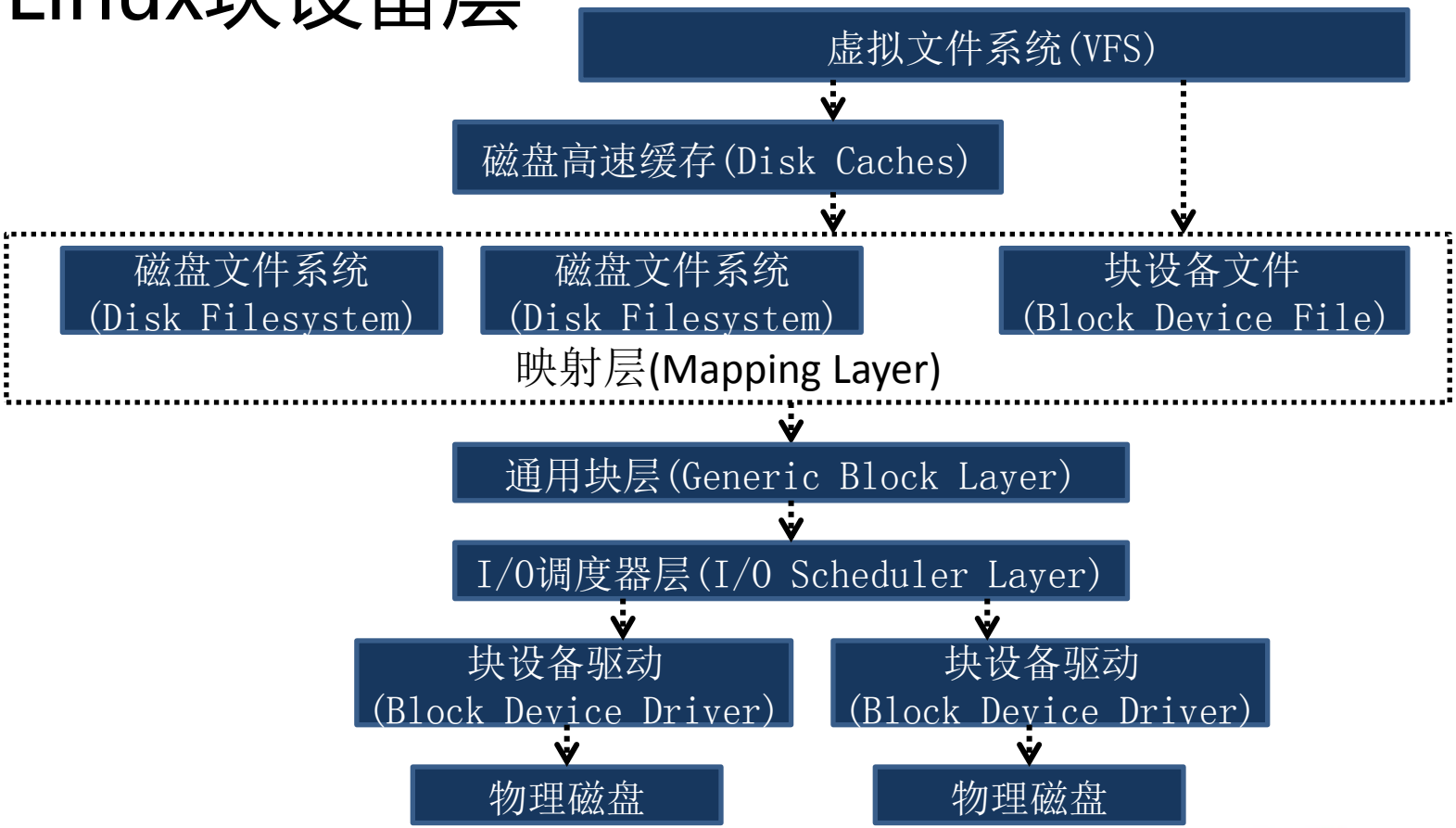
# Linux块设备驱动

- 块设备操作
  - 块设备getgeo()操作
    - 磁盘容量与磁头数和扇区数对应关系

| 磁盘容量 | 磁头数 | 扇区数 |
| --- | --- | --- |
| 容量 < 16MB | 1 | 1 |
| 16MB ≤ 容量 < 512MB | 1 | 32 |
| 512MB ≤ 容量 < 16GB | 32 | 32 |
| 16GB ≤ 容量 | 255 | 63 |

# Linux块设备驱动

- Linux块设备层



# Linux块设备驱动

- 非机械块设备
  - 对于机械的磁盘设备来说，请求队列明显有助于提高系统性能
  - 但是，对于如：Flash、SD卡、RAM Disk等可随机访问的块设备，请求队列毫无意义
  - 对于非机械设备，通用块层支持“无队列”的操作模式，驱动必须提供一个“制造请求”函数(通常，无需完成真正的请求创建，而是在此直接完成设备访问)

# Linux块设备驱动

- 非机械块设备
  - request_queue操作
    - 分配请求队列
      - 对应Flash、RAM Disk等完全随机访问的非机械设备，并不需要进行复杂的I/O调度；此时，可以使用该函数分配一个请求队列，并使用下面的blk_queue_make_request()函数来绑定请求队列和“制造请求”函数
      - struct request_queue *blk_alloc_queue(gfp_t gfp_mask);
        - » gfp_mask: 分配请求队列结构体内存空间的内核内存分配标志
        - » 返回: 分配的请求队列指针，失败返回NULL

# Linux块设备驱动

- 非机械块设备
  - request_queue操作
    - 绑定请求队列
      - 绑定请求队列和“制造请求”函数
      - void blk_queue_make_request(struct request_queue *q, make_request_fn *mfn);
        - » q: 待绑定的请求队列
        - » mfn: 待绑定的“制造请求”函数

# Linux块设备驱动

- 非机械块设备
  - “制造请求”函数原型：
    - typedef int (make_request_fn) (struct request_queue *q, struct bio *bio);
      - q: 待添加请求的请求队列
      - bio: 待处理的块I/O
      - 返回: 成功返回0，失败返回负值

# Linux块设备驱动

- 非机械块设备
  - 块I/O
    - 内核中描述块I/O的结构体是bio结构体
    - 通常一个bio对应一个块I/O请求
    - I/O调度算法可将连续的bio合并成一个请求
    - 所以，通常一个request包含多个bio

# Linux块设备驱动

- 非机械块设备
  - bio结构体
    - 定义在<linux/bio.h>中

```
struct bio {
    sector_t bi_sector; // 该bio操作的起始扇区
    struct bio *bi_next; // 下一个bio
    struct block_device *bi_bdev; // 该bio对应的块设备
    unsigned long bi_flags; // 描述bio的标志，最低位为读写标志，最低位被置位表示写请求；可以使用bio_data_dir(bio)宏来获取读写方向
    unsigned long bi_rw; // 低位表示读/写方向，高位表示优先级
```

# Linux块设备驱动

- 非机械块设备
  - bio结构体
    - 定义在<linux/bio.h>中

      unsigned short bi_vcnt; // 该bio中bio_vec的数量

      unsigned short bi_idx; // 当前bio在bi_io_vec中的索引

      unsigned int bi_phys_segments; // 不相邻的物理段的数目

      unsigned int bi_size; // 被传送的数据大小(按字节计)，可以用bi_sector(bio)宏获取以扇区为单位的大小

      unsigned int bi_seg_front_size; // 第一个可合并段尺寸

      unsigned int bi_seg_back_size; // 最后一个可合并段尺寸

# Linux块设备驱动

- 非机械块设备
  - bio结构体
    - 定义在<linux/bio.h>中

```
unsigned int bi_max_vecs; // 该bio能持有的最大bvl_vecs数目
unsigned int bi_comp_cpu; // 完成的CPU
struct bio_vec *bi_io_vec; // 该bio处理的真实向量列表
bio_end_io_t *bi_end_io; // bio完成处理函数
atomic_t bi_cnt; // 使用计数
void *bi_private; // 私有数据
bio_destructor_t *bi_destructor; // 解除函数
};
```

# Linux块设备驱动

- 非机械块设备
  - bio_vec结构体
    - 定义在<linux/bio.h>中

```
struct bio_vec {
    struct page *bv_pdate; // 指向当前数据缓冲区所驻留的物理页指针
    unsigned int bv_len; // 当前数据缓冲区的大小（按字节计数）
    unsigned int bv_offset; // 当前数据缓冲区在物理页中的偏移量（按字节计数）
};
```

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - 定义在<linux/bio.h>中
    - 获取数据方向
      - 它定义在<linux/fs.h>中
      - unsigned long bio_data_dir(struct bio *bio);
        - » bio: 待处理的bio
        - » 返回: 0代表读，1代表写
    - 获取当前页指针
      - struct page *bio_page(struct bio *bio);
        - » bio: 待处理的bio
        - » 返回: 当前数据缓冲区所在的物理页指针

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - 获取当前偏移
      - unsigned int bio_offset(struct bio *bio);
        - » bio: 待处理的bio
        - » 返回: 当前数据缓冲区在位于所在物理页中的偏移；通常块I/O操作本身是页对齐的
    - 获取当前扇区数
      - unsigned int bio_cur_sectors(struct bio *bio);
        - » bio: 待处理的bio
        - » 返回: 当前bio_vec要处理的扇区数

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - bio_data操作
      - 该函数仅在请求的页不在高端内存中时可用，其它情况使用会产生错误；也就是说，如果已经使用blk_queue_bounce_limit()改变设置，不能调用该函数
      - void *bio_data(struct bio *bio);
        - » bio: 待处理的bio
        - » 返回: 数据缓冲区指针（内核虚拟地址），失败返回NULL

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - bio_kmap_irq和bio_kunmap_irq操作
      - bio_kmap_irq()函数用于获取任何情况下的数据缓冲区内核虚拟地址；不管该数据是否位于高端或低端内存区；它必须和bio_kunmap_irq() 成对使用
      - 使用限制：
        - » 由于使用原子kmap，在操作期间不能睡眠
        - » 同理，不能一次映射多于一个段

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - bio_kmap_irq和bio_kunmap_irq操作
      - char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
        - » bio: 待处理的bio
        - » flags: 保存标志
        - » 返回: 数据缓冲区指针（内核虚拟地址），失败返回NULL
      - void bio_kunmap_irq(struct bio *bio, unsigned long *flags);
        - » bio: 待处理的bio
        - » flags: bio_kmap_irq()中保存的标志

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - bio_for_each_segment操作
      - 该操作遍历bi_io_vec中未被处理的项
      - #define bio_for_each_segment(bvl, bio, i) __bio_for_each_segment(bvl, bio, i, (bio)->bi_idx)
        - » bvl: 获取的bio_vec指针
        - » bio: 待处理的bio
        - » i: 循环变量

# Linux块设备驱动

- 非机械块设备
  - bio操作
    - bio完成
      - 该函数负责通知bio已经处理完成
      - void bio_endio(struct bio *bio, int error);
        - » bio: 待处理的bio
        - » error: 0表示正确处理bio，负值表示该bio处理出错

# Linux网络设备驱动

- Linux网络功能
  - Linux操作系统最为突出的特点之一就是其内置的网络支持，其网络功能几乎是包罗万象
  - Linux不仅支持当前的TCP/IP协议，也是最早支持下一代Internet协议IPv6的操作系统之一
  - Linux内核还包括了IP防火墙代码、IP防伪、IP服务质量控制及许多完全特性

# Linux网络设备驱动

- 网络协议栈

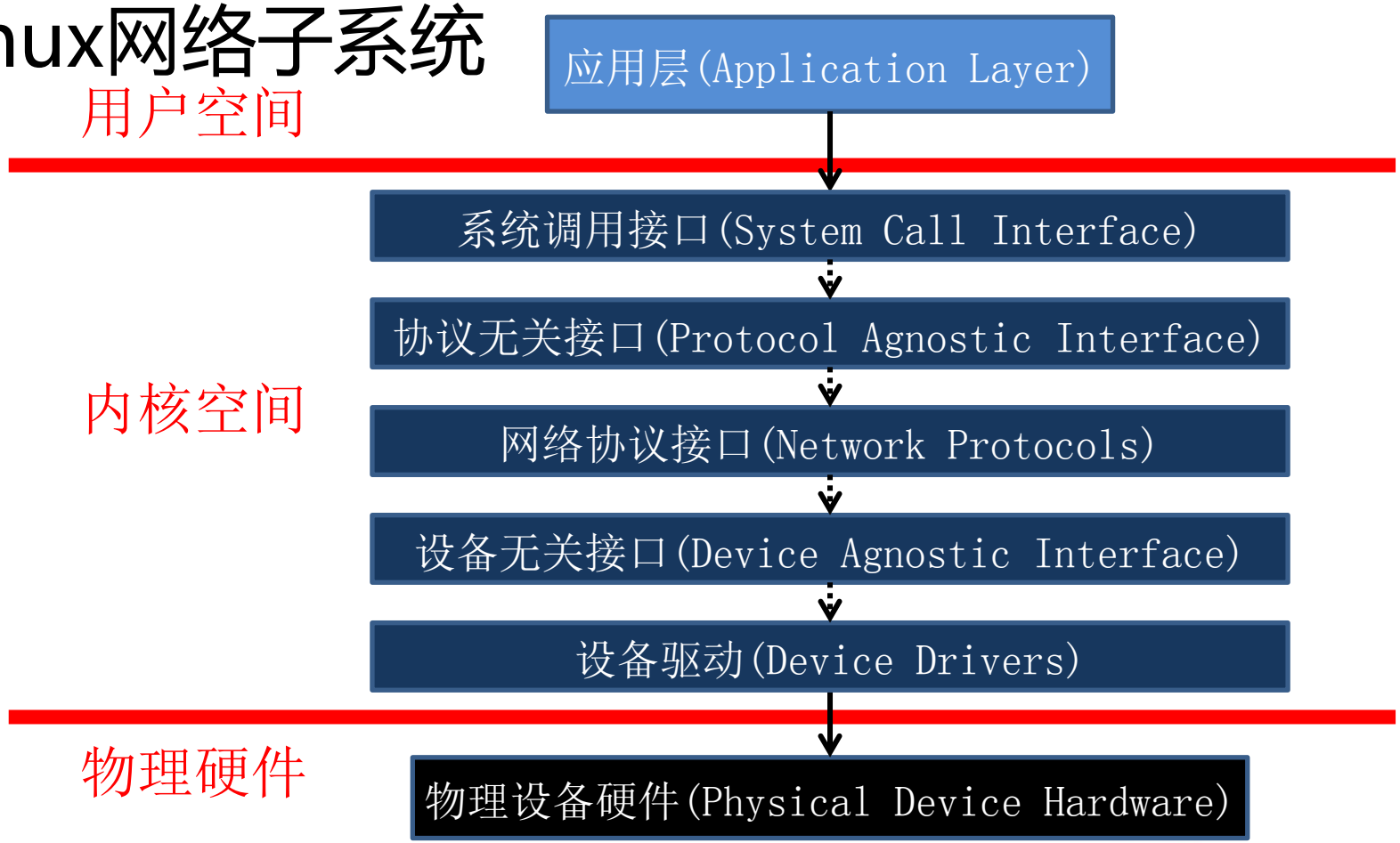
| OSI七层网络模型 | Linux四层概念模型 | 对应网络协议 |
|---|---|---|
| 应用层 | 应用层 | TFTP, FTP, NFS, WAIS |
| 表示层 | | Telnet, Rlogin, SNMP, Gopher |
| 会话层 | | SMTP, DNS |
| 传输层 | 传输层 | TCP, UDP |
| 网络层 | 网际层 | IP, ICMP, ARP, RARP, AKP, UUCP |
| 数据链路层 | 网络接口 | FDDI, Ethernet, Arpanet, PDN, SLIP, PPP |
| 物理层 | | IEEE 802.1A/802.2 |

# Linux网络设备驱动

- 网络接口层
  - Linux系统的网络接口层把数据链路层和物理层合并在一起
  - 该层主要负责从物理介质接收和发送数据
  - 该层也提供访问物理设备的驱动程序，对应的协议主要是以太网协议

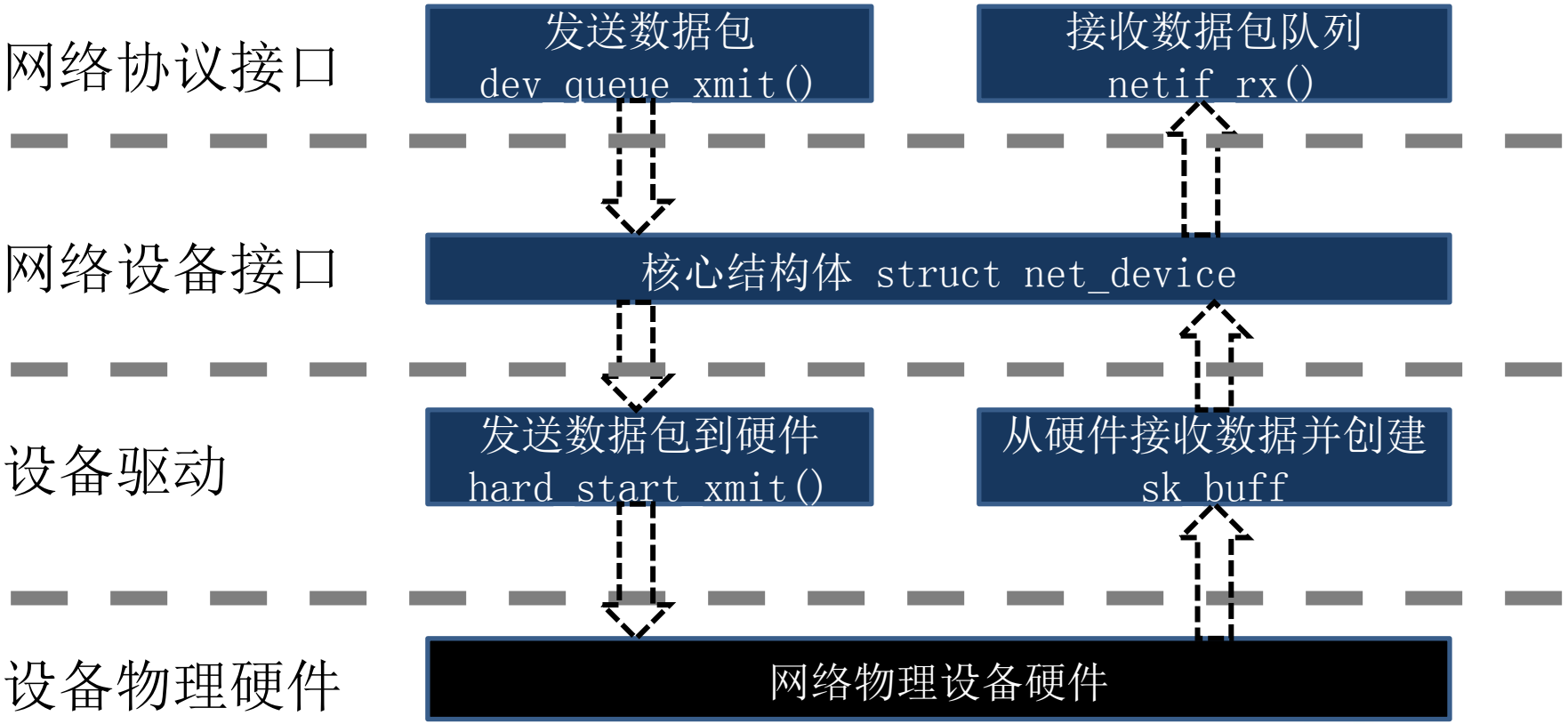
# Linux网络设备驱动

- Linux网络子系统



# Linux网络设备驱动

- Linux网络设备驱动框架



# Linux网络设备驱动

- 网络设备驱动
  - net_device结构体
    - 定义在<linux/netdevice.h>中

```
struct net_device {
    char name[IFNAMSIZ]; // 网络设备名称
    ...
    unsinged long state; // 当前设备状态
    ...
    unsigned long base_addr; // 设备I/O基地址
    ...
    unsigned int irq; // 设备占用中断号
    ...
    int (*init)(struct net_device *dev); // 设备初始化
    int (*open)(struct net_device *dev); // 设备打开
    int (*stop)(struct net_device *dev); // 设备关闭
    int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev); // 数据发送
    ...
};
```

# Linux网络设备驱动

- 网络设备驱动
  - net_device结构体
    - init函数
      - 设备初始化函数，在调用register_netdev()时，由内核调用来完成对net_device结构的初始化
    - open函数
      - 打开设备
    - stop函数
      - 关闭设备
    - hard_start_xmit函数
      - 数据发送

# Linux网络设备驱动

- 网络设备驱动
  - net_device操作
    - alloc_netdev宏
      - 定义在<linux/netdevice.h>中
      - struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct net_device *));
        - » sizeof_priv: 驱动私有数据区大小
        - » name: 设备名称
        - » setup: 初始化函数
        - » 返回: 分配到的net_device结构体指针，失败返回NULL
    - alloc_etherdev宏
      - 定义在<linux/etherdevice.h>中
      - struct net_device *alloc_etherdev(int sizeof_priv);

# Linux网络设备驱动

- 网络设备驱动
  - net_device操作
    - 定义在<linux/netdevice.h>中
    - 实现在内核源代码net/core/dev.c中
    - 网络设备注册
      - int register_netdev(struct net_device *dev);
        - » dev: 待注册的网络设备
        - » 返回: 成功返回0，失败返回负值
    - 网络设备注销
      - void unregister_netdev(struct net_device *dev);
        - » dev: 待注销的网络设备

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff结构体
    - 定义在<linux/skbuff.h>中

```
struct sk_buff {
    ...
    struct net_device *dev; // 发送或接收该报文数据的设备
    ...
    sk_buff_data_t transport_header; // 传输层头部指针
    sk_buff_data_t network_header; // 网络层头部指针
    sk_buff_data_t mac_header; // 链路层头部指针
    ...
    sk_buff_data_t tail ; // 有效数据的结束
    sk_buff_data_t end; // 分配空间的结束
    unsigned char *head, *data; // 分配空间和有效数据的开始
    unsigned int truesize; // 此报文存储区的长度，该长度是16字节对齐的，一般比报文的实际长度大
    ...
};
```

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff操作
    - 定义在<linux/skbuff.h>中
    - 实现在内核源代码net/core/skbuff.c中

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff操作
    - 分配sk_buff
      - struct sk_buff *alloc_skb(unsigned int size, gfp_t priority);
        - » 协议栈使用的函数
        - » size: 分配用于存储网络报文空间的大小
        - » priority: 内核内存分配标志
        - » 返回: 分配到的sk_buff指针，失败返回NULL
      - struct sk_buff *dev_alloc_skb(unsigned int length);
        - » 驱动程序使用的函数
        - » length: 分配用于存储网络报文空间的大小
        - » 返回: 分配到的sk_buff指针，失败返回NULL

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff操作
    - 移动指针操作
      - unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
        - » skb: 待处理的skb
        - » len: 将tail指针下移长度
        - » 返回: 移动前的tail指针
      - unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
        - » skb: 待处理的skb
        - » len: 将data指针上移长度
        - » 返回: 移动前的data指针

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff操作
    - 移动指针操作
      - unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);
        - » skb: 待处理的skb
        - » len: 将data指针下移长度
        - » 返回: 移动前的data指针

# Linux网络设备驱动

- 网络设备驱动
  - sk_buff操作
    - 释放skb_buff
      - void kfree_skb(struct sk_buff *skb);
        - » 协议栈使用的函数
        - » skb: 待释放的skb_buff
      - void dev_kfree_skb(struct sk_buff *skb);
        - » 驱动程序使用的函数
        - » skb: 待释放的skb_buff

# Linux网络设备驱动

- 网络设备驱动
  - 提交报文操作
    - 定义在<linux/netdevice.h>中
    - 实现在内核源代码net/core/dev.c中
      - int netif_rx(struct sk_buff *skb);
        - » skb: 待提交的sk_buff
        - » 返回: 通常返回:
          - NET_RX_SUCCESS: 报文被成功接收
          - NET_RX_DROP: 报文被丢弃

# Linux设备驱动实例

- input输入子系统
  - 输入设备（如按键、键盘、触摸屏、鼠标等）是典型的字符设备，一般的工作机制是底层在按键、触摸等动作发生时产生一个中断（或驱动通过timer定时查询），然后CPU通过特定的接口（如SPI、I2C、USB等总线）读取键值、坐标等数据，放入一个由字符设备驱动管理的缓冲区，驱动的read()接口让用户可以读取键值、坐标等数据

# Linux设备驱动实例

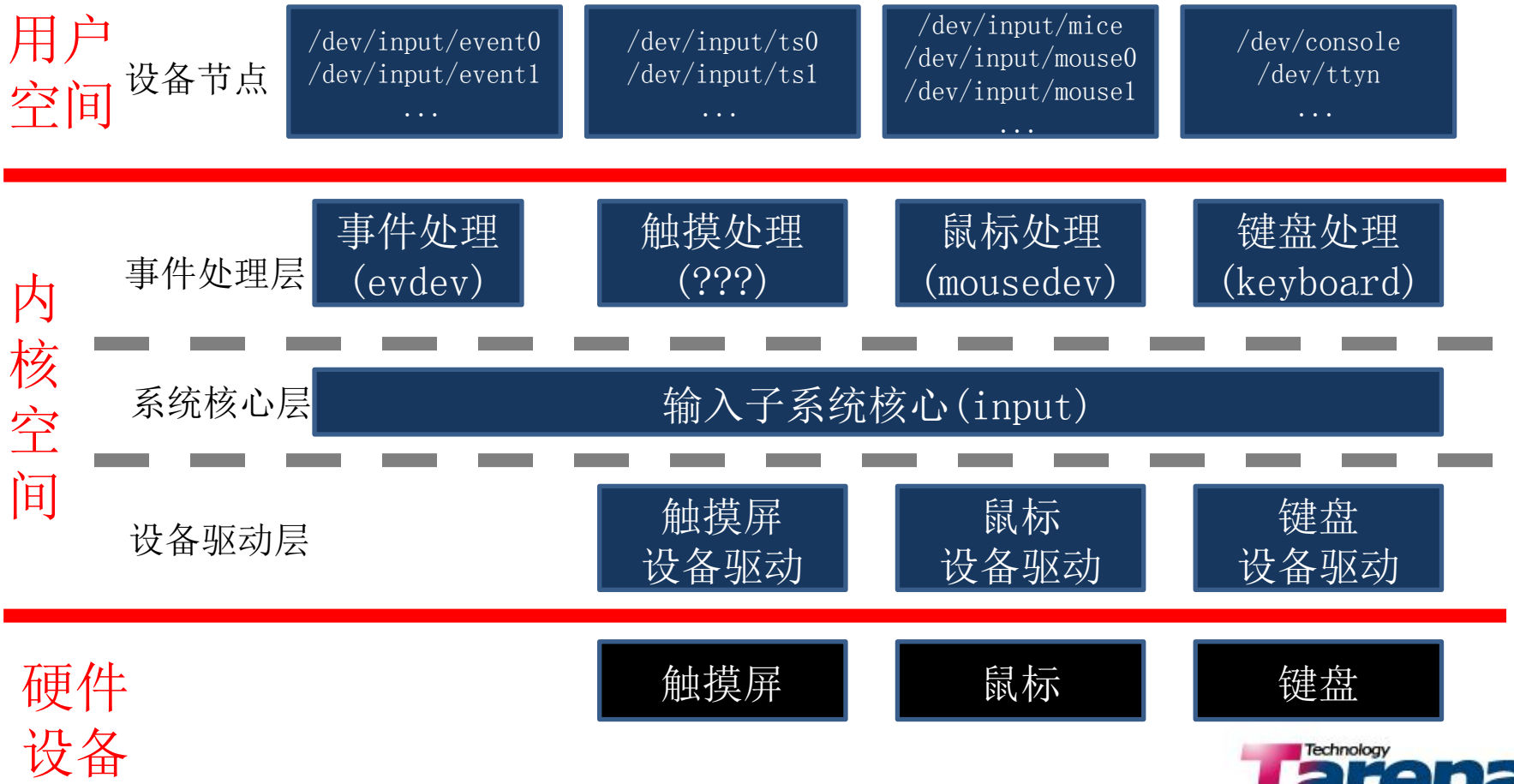
- input输入子系统
  - 在Linux系统中，input输入子系统由输入子系统设备驱动层(Device Driver)、输入子系统核心层(Input Core)、输入子系统事件处理层(Event Handler)组成

# Linux设备驱动实例

- Input输入子系统
  - 设备驱动层
    - 该层提供对硬件的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层提交给事件处理层
  - 系统核心层
    - 该层对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口
  - 事件处理层
    - 该层为用户空间的应用程序提供了统一访问设备的接口和完成驱动层提交上来的事件处理

# Linux设备驱动实例

- Input输入子系统框架

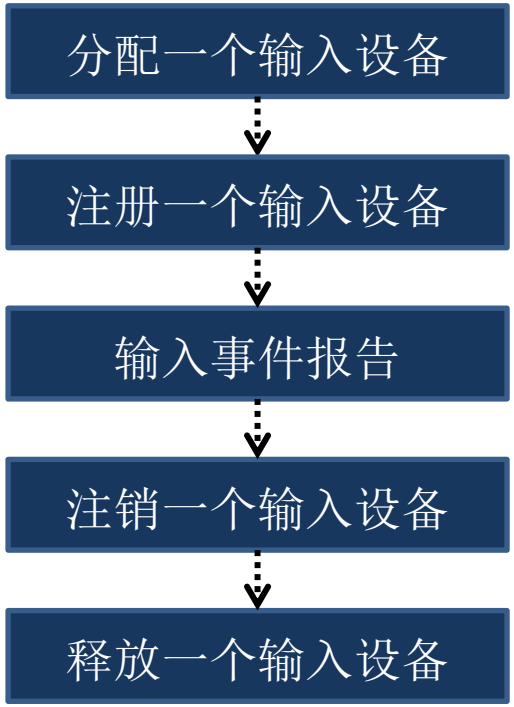


# Linux设备驱动实例

- input输入子系统
  - 设备驱动任务
    - 在模块加载函数中，设置input输入设备支持input子系统定义的哪些事件
    - 将input输入设备注册到input输入子系统中
    - 在input输入设备发生输入操作时（如：键盘按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时等），提交所发生的事件及对应的键值/坐标等状态

# Linux设备驱动实例

- input输入子系统
  - 设备驱动流程



# Linux设备驱动实例

- input输入子系统
  - 输入事件类型

```
#define EV_SYN          0x00 // 同步事件
#define EV_KEY          0x01 // 按键（键盘或按钮）
#define EV_REL          0x02 // 相对坐标（鼠标）
#define EV_ABS          0x03 // 绝对坐标（触摸屏等）
#define EV_MSC          0x04 // 其它
#define EV_SW           0x05
#define EV_LED          0x11 // LED等指示设备
#define EV_SND          0x12 // 声音（如：蜂鸣器）
#define EV_REP          0x14 // 重复
#define EV_FF           0x15 // 力反馈
#define EV_PWR          0x16
#define EV_FF_STATUS    0x17 // 力反馈状态
#define EV_MAX          0x1f
#define EV_CNT          (EV_MAX+1)
```

# Linux设备驱动实例

- input输入子系统
  - input_dev结构体
    - 定义在<linux/input.h>中

```
struct input_dev {
    const char *name; // 设备名称
    const char *phys; // 设备在系统中的路径
    ...
    struct input_id id; // 设备识别标志

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; // 支持的事件类型
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; // 支持的按键类型
    ...
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; // 支持的绝对坐标系统
```

# Linux设备驱动实例

- input输入子系统
  - input_dev结构体
    - 定义在<linux/input.h>中

```
...
// 绝对坐标范围
int absmax[ABS_MAX + 1];
int absmin[ABS_MAX + 1];
int absfuzz[ABS_MAX + 1];
int absflat[ABS_MAX + 1];
...
};
```

# Linux设备驱动实例

- input输入子系统
  - input_dev操作
    - 分配input_dev
      - 分配并初始化input_dev结构体
      - struct input_dev *input_allocate_device(void);
        - » 返回: 分配到的input_dev结构体指针，失败返回NULL
    - 注册input_dev
      - 注册输入设备到输入子系统核心
      - int input_register_device(struct input_dev *dev);
        - » dev: 待注册的输入设备
        - » 返回: 成功返回0，失败返回负值

# Linux设备驱动实例

- input输入子系统
  - 提交事件报告
    - 提交键值报告
      - void input_report_key(struct input_dev *dev, unsigned int code, int value);
        - » dev: 提交报告的输入设备
        - » code: 报告的键值
        - » value: 报告键值的对应数值
    - 提交绝对坐标报告
      - void input_report_abs(struct input_dev *dev, unsigned int code, int value);
        - » dev: 提交报告的输入设备
        - » code: 报告绝对坐标的键值
        - » value: 报告绝对坐标键值的对应数值

# Linux设备驱动实例

- input输入子系统
  - 提交事件报告
    - 提交同步事件报告
      - void input_sync(struct input_dev *dev);
        - » dev: 待提交同步事件报告的输入设备
    - 提交事件
      - 上述三个报告函数均调用该函数实现
      - void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);
        - » dev: 提交报告的输入设备
        - » type: 提交报告的事件类型
        - » code: 提交报告的键值
        - » value: 提交报告键值的对应数值

# Linux设备驱动实例

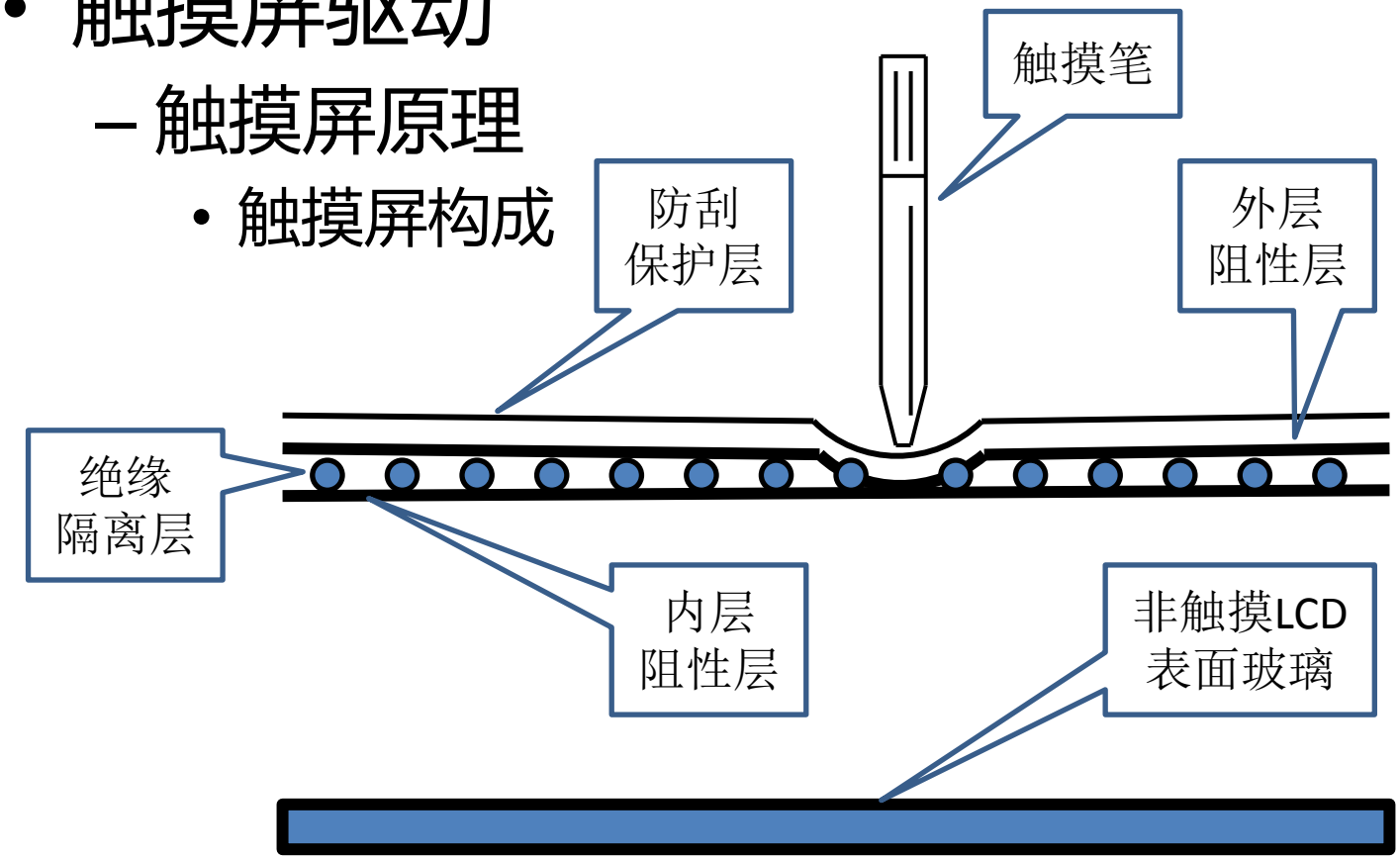
- 触摸屏驱动
  - 触摸屏原理
    - 根据触摸屏的工作原理和传输信息的介质不同，触摸屏分为四种：电阻式、电容式、红外线式和表面声波式
    - 电阻式触摸屏根据检测方式不同又可分为：四线触摸屏、五线触摸屏、八线触摸屏等
    - 因为成本及技术成熟等因素，四线式触摸屏技术被大量采用

# Linux设备驱动实例

- 触摸屏驱动
  - 触摸屏原理
    - 触摸屏构成
      - 触摸屏由上下两层透明的阻性导体层和两层导体之间的隔离层，以及电极构成
      - 阻性导体层选用阻性材料，如：ITO（铟锡氧化物），涂在衬底上构成
      - 隔离层通常为透明绝缘性液体材料，如：聚酯薄膜
      - 电极通常选用导电性能极好的材料，如：银粉墨，其导电性能大约为 ITO 的100倍

# Linux设备驱动实例

- 触摸屏驱动
  - 触摸屏原理
    - 触摸屏构成

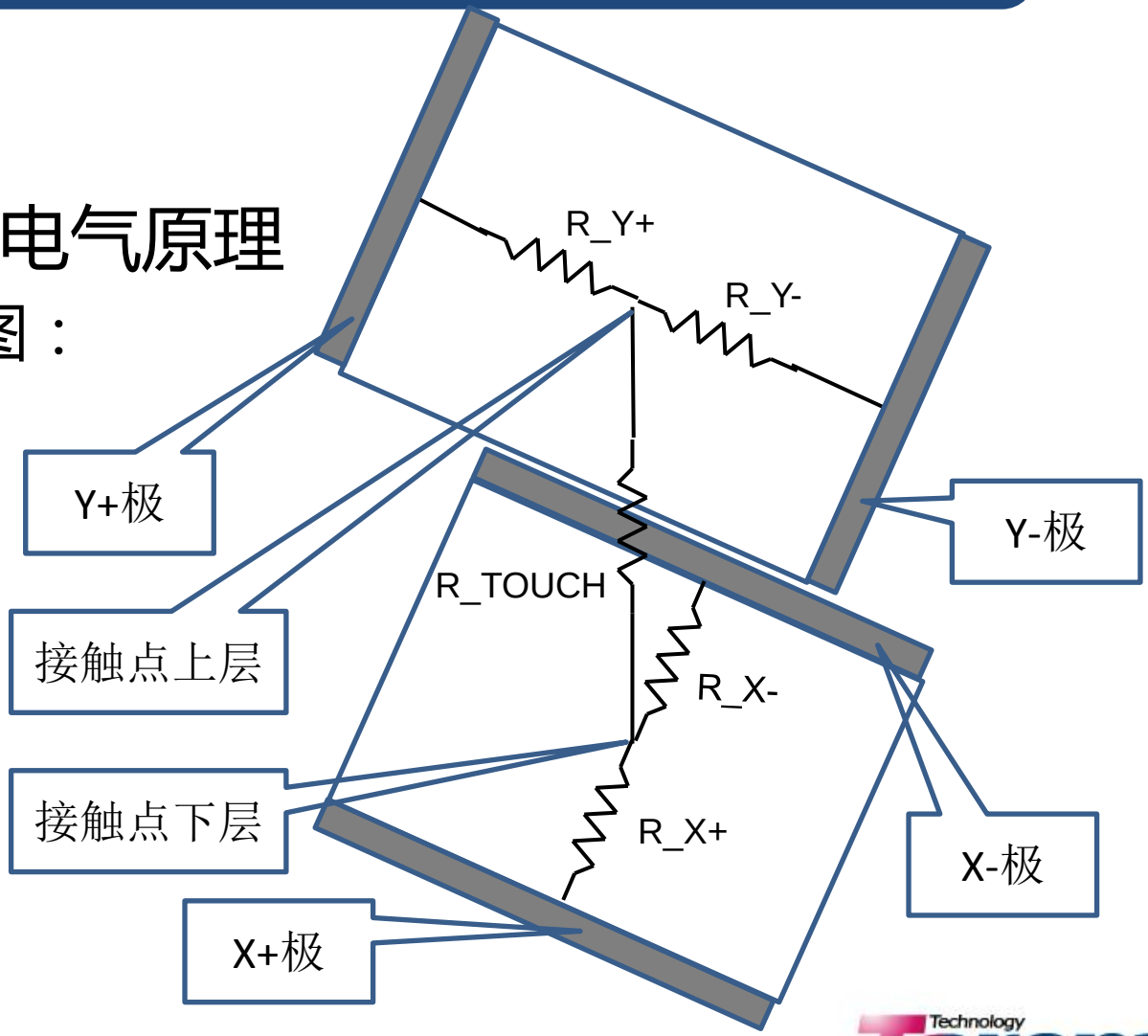


# Linux设备驱动实例

- 触摸屏驱动
  - 四线触摸屏电气原理
    - 当某一阻性层的某个电极加上电压时，会在该网络上形成电压梯度
    - 当有物体接触触摸屏表面并施加一定的压力时，上层的 ITO 导电层发生形变与下层 ITO 发生接触，该结构可以等效为相应的电路

# Linux设备驱动实例

- 触摸屏驱动
  - 四线触摸屏电气原理
    - 等效电路图：

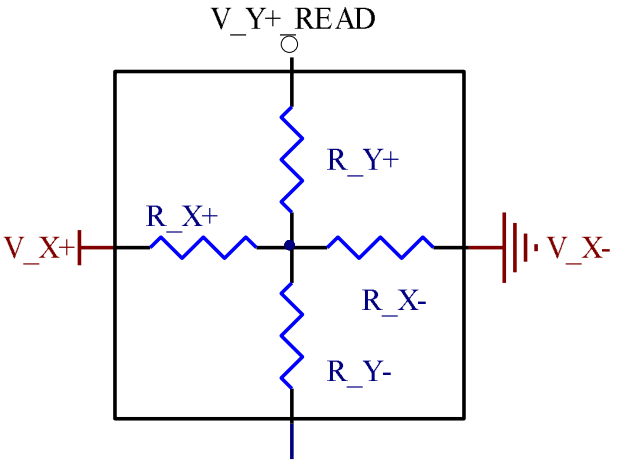


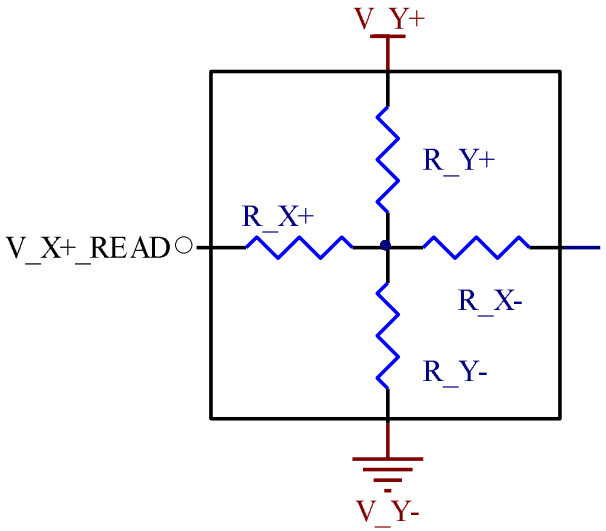
# Linux设备驱动实例

- 触摸屏驱动
  - 四线触摸屏检测方法
    - 坐标检测（触点X/Y坐标）
      - 计算X坐标
        - » 在X+电极上施加电压V_X+，X-电极接地；Y+作为引出端测量得到接触点的电压V_Y+_READ，由于 ITO 层均匀导电，触点引出端电压与施加电压V_X+之比等于触点X坐标与屏幕宽度之比
      - 计算Y坐标
        - » 在Y+电极上施加电压V_Y+，Y-电极接地；X+作为引出端测量得到接触点的电压V_X+_READ，由于 ITO 层均匀导电，触点引出端电压与施加电压V_Y+之比等于触点Y坐标与屏幕高度之比

# Linux设备驱动实例

- 触摸屏驱动
  - 四线触摸屏检测方法
    - 坐标检测（触点X/Y坐标）




$$X\_COORD = WIDTH \frac{V\_Y+\_READ}{V\_X+}$$

$$Y\_COORD = HEIGHT \frac{V\_X+\_READ}{V\_Y+}$$

# Linux设备驱动实例

- 触摸屏驱动
  - ADC转换器
    - S3C6410芯片ADC控制器
  - 触摸屏驱动
    - 触摸屏驱动代码
  - 触摸屏驱动测试
    - 编写应用程序测试触摸屏

# Linux设备驱动实例

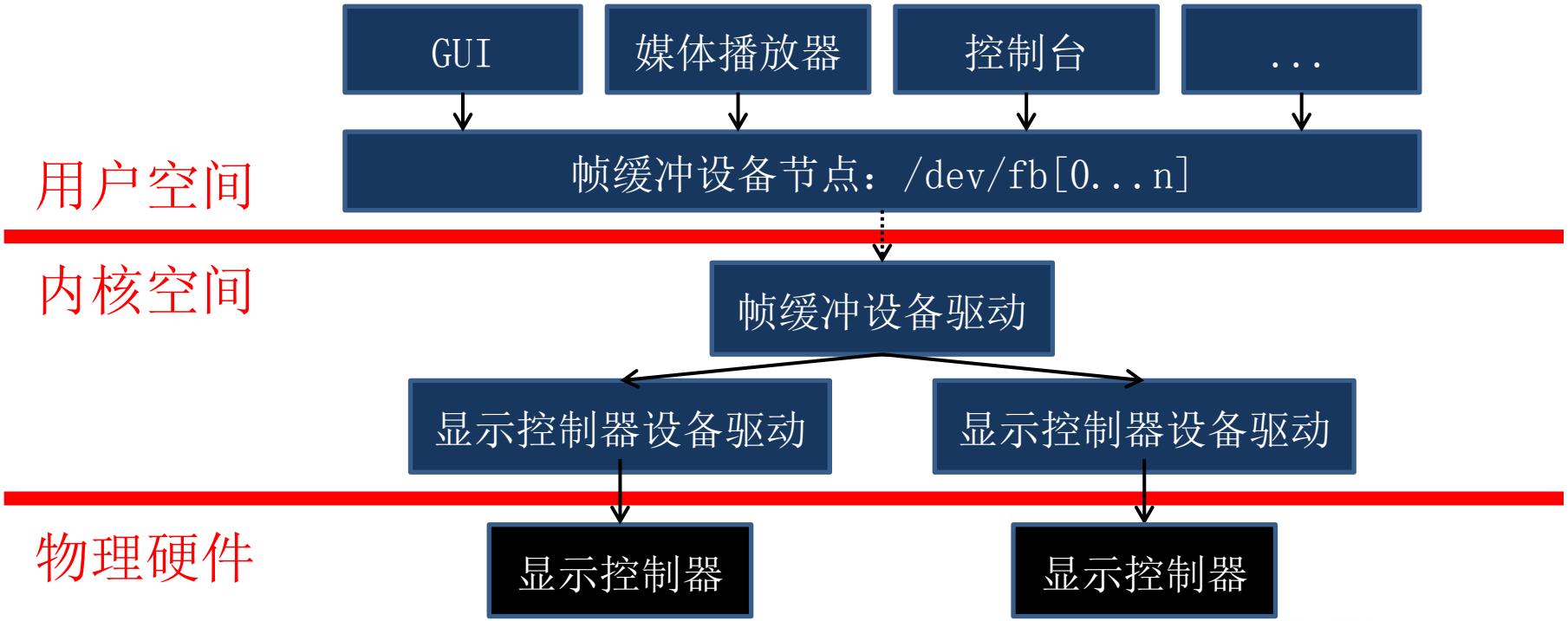
- 帧缓冲(Frame Buffer)
  - 帧缓冲(framebuffer)是Linux系统为图形设备提供的一个抽象接口，它允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作；这种操作是抽象的、统一的；应用程序不必关心物理显存的位置、换页机制等等具体细节
  - 帧缓冲(framebuffer)是嵌入式Linux系统中采用的主流显示技术

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - 帧缓冲设备属于字符设备，在Linux体系中居于上层应用和底层显示设备之间；它的作用是对上层应用屏蔽掉底层不同硬件的操作细节；为上层提供统一接口，管理底层设备驱动

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - Linux显示体系框图



# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - 帧缓冲相关数据结构
    - 对用户而言，帧缓冲就是内存中的一块区域，可以对它进行读、写、映射等操作
    - 只要将显示设备映射到用户进程空间，可以理解为将屏幕上的每一个点和帧缓冲的代表一个点的内存字节一一对应起来

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - 帧缓冲相关数据结构
    - 定义在<linux/fb.h>中
      - fb_fix_screeninfo结构体
      - fb_var_screeninfo结构体
      - fb_info结构体
      - fb_ops结构体

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_fix_screeninfo结构体
    - fb_fix_screeninfo结构体描述帧缓冲设备中设备无关的固定参数信息，这些特性不能通过ioctl()直接修改，只能通过修改fb_var_screeninfo结构体中的某些参数间接调整fb_fix_screeninfo中的特性
    - 其中主要特性包括: smem_start、smem_len、line_length等
    - 应用程序可以通过ioctl()系统调用并使用控制命令FBIOGET_FSCREENINFO获取当前帧缓冲设备的固定参数信息

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_fix_screeninfo结构体

```
struct fb_fix_screeninfo {
    char id[16]; // 字符串形式的标识符（设备名称）
    unsigned long smem_start; // 帧缓冲区内存的起始地址（物理地址）
    __u32 smem_len; //帧缓冲区的长度
    __u32 type; // 设备类型（如：TFT或STN等）
    __u32 type_aux; // 隔行方式
    __u32 visual;  // 色彩类型（如：真彩色、假彩色或单色）
    __u16 xpanstep; // 如果没有硬件panning，赋0
    __u16 ypanstep;
    __u16 ywrapstep;
    __u32 line_length; // 屏幕上每行字节数
    unsigned long mmio_start; // 内存I/O映射的起始地址（物理地址）
    __u32 mmio_len; // 内存I/O映射的长度
    __u32 accel; // 指示特定显示驱动
    __u16 reserved[3]; // 保留字节
};
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_var_screeninfo结构体
    - fb_var_screeninfo结构体描述帧缓冲设备中设备无关的可变参数信息和特定的显示模式
    - 其中主要参数包括：xres、yres、height、width等
    - 应用程序可以通过ioctl()系统调用并使用控制命令FBIOGET_VSCREENINFO或FBIOPUT_VSCREENINFO获取或设置当前帧缓冲设备的可变参数信息
    - 应用程序可以通过ioctl()系统调用并使用控制命令FBIOPAN_DISPLAY实现屏的方向和分辨率改变

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_var_screeninfo结构体

```
struct fb_var_screeninfo {
    __u32 xres; // 可视x分辨率
    __u32 yres; // 可视y分辨率
    __u32 xres_virtual; // 虚拟x分辨率
    __u32 yres_virtual; // 虚拟y分辨率
    __u32 xoffset; // 可视区域在虚拟区域中的x偏移
    __u32 yoffset ; // 可视区域在虚拟区域中的y偏移

    __u32 bits_per_pixel; // 即BPP，每个像素的位数
    __u32 grayscale; // 如果不为0，将采用指定灰度等级显示

    // 当为真彩色时，表示R/G/B位域；否则仅仅指示长度
    struct fb_bitfield red;
    struct fb_bitfield green;
    struct fb_bitfield blue;
    struct fb_bitfield transp; // 透明度
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_var_screeninfo结构体

```
__u32 nonstd; // 非0表示非标准像素格式

__u32 activate; // 当前fb活动

__u32 height; // 图像高度
__u32 width; // 图像宽度

__u32 accel_flags; // 标志(fb_info.flags)
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_var_screeninfo结构体

```
// 除pixclock本身外，其余信号时序均已pixcloclk为基准
__u32 pixclock; // 像素时钟(ps为单位)
__u32 left_margin; // 从行同步信号到数据有效间的延迟时间
__u32 right_margin ; // 从数据有效到下一个行同步信号间的延迟时间
__u32 upper_margin; // 从帧同步信号到数据有效间的延迟时间
__u32 lower_margin; // 从数据有效到下一个帧同步信号间的延迟时间
__u32 hsync_len; // 行同步信号同步时钟
__u32 vsync_len; // 帧同步信号同步行数
__u32 sync; // 同步标志
__u32 vmode; // video模式标志
__u32 rotate; // 顺时针旋转的角度
__u32 reserved[5]; // 保留
};
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_var_screeninfo结构体
    - fb_bitfield结构体

```
struct fb_bitfield {
    __u32 offset; // 位域的偏移
    __u32 length; // 位域的长度
    __u32 msb_right; // 如果不为0，表示最高有效位在右边
};
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - 帧缓冲用户程序
    - 操作帧缓冲的用户程序操作步骤
      - 打开帧缓冲设备文件
      - 获取fb_info的固定参数和可变参数
      - 计算帧缓冲区大小
      - 调用mmap将缓冲区映射到进程的地址空间
      - 读/写缓冲区（读/写屏幕像素数据）

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info结构体
    - fb_info结构体描述控制器设备的综合信息，包括驱动API以及其它底层信息（如：内存地址等）
    - 其中包括fb_fix_screeninfo、fb_var_screeninfo、fb_cmap三个结构体的实例

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info结构体

```
struct fb_info {
    int node; // fb在registered_fb[]中的索引
    int flags; // fb标志
    struct mutex lock; // 用于open/release/ioctl的保护锁
    struct fb_var_screeninfo var; // 当前可变参数
    struct fb_fix_screeninfo fix; // 当前固定参数
    struct fb_monspecs monspecs; // 显示器标准
    struct work_struct queue; // 帧缓冲事件队列
    struct fb_pixmap pixmap; // 图像硬件映射
    struct fb_pixmap sprite; // 光标硬件映射
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info结构体

```
struct fb_cmap cmap; // 当前颜色表
struct list_head modelist; // 模式数据链
struct fb_videomode *mode; // 当前video模式

#ifdef CONFIG_FB_BACKLIGHT
  // 分配背光设备，应该在帧缓冲注册前设置，在注销后再移除
  struct backlight_device *bl_dev; // 对应的背光设备

  // 背光曲线
  struct mutex bl_curve_mutex; // 背光曲线互斥锁
  u8 bl_curve[FB_BACKLIGHT_LEVELS]; // 背光曲线数组
#endif
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info结构体

```
#ifdef CONFIG_FB_DEFERRED_IO // 帧缓冲延迟I/O mmap
   struct delayed_work deferred_work; // 延迟I/O任务
   struct fb_deferred_io *fbdefio; // 延迟I/O结构体
#endif

   struct fb_ops *fbops; // 帧缓冲操作
   struct device *device; // 父设备
   struct device *dev; // 当前设备本身
   int class_flag; // 私有的sysfs标志
#ifdef CONFIG_FB_TILEBLITTING
   struct fb_tile_ops *tileops; // 图块操作
#endif
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info结构体

```
char __iomem *screen_base; //内核虚拟基地址
unsigned long screen_size; // ioremap的虚拟内存大小或0
void *pseudo_palette; // 伪16色颜色模板
#define FBINFO_STATE_RUNNING        0
#define FBINFO_STATE_SUSPENDED      1
u32 state; // 硬件状态，比如：挂起
void *fbcon_par; // 私有数据
// 设备无关数据
void *par;
};
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_ops结构体
    - fb_ops结构体描述可以在控制器设备驱动中执行的操作集合
    - 应用程序可以使用ioctl()系统调用来操作显示设备

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_ops结构体

```
struct fb_ops {
    // 打开/释放
    struct module *owner; // 模块拥有者
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);

    // 对于非线性布局的，常规内存映射方法无法工作的帧缓冲设备需要
    ssize_t (*fb_read)(struct fb_info *info, char __user *buf, size_t count, loff_t
    *ppos);
    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf, size_t count,
    loff_t *ppos);

    // 检查可变参数，并调整到支持的值，不修改par
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);

    // 根据info->var设置video模式
    int (*fb_set_par)(struct fb_info *info);
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_ops结构体

```
// 设置颜色寄存器
int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
unsigned blue, unsigned transp, struct fb_info *info);

// 批量设置颜色寄存器，设置颜色表
int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

// 显示空白
int (*fb_blank)(int blank, struct fb_info *info);

// pan显示
int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);

// 画矩形
void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect *rect);
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_ops结构体

```
// 拷贝区域
void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea *region);
// 图形填充
void (*fb_imageblit) (struct fb_info *info, const struct fb_image *image);

// 绘制光标
int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);

// 旋转显示
void (*fb_rotate)(struct fb_info *info, int angle);

// 等待blit空闲，可选
int (*fb_sync)(struct fb_info *info);

// 控制器特定的ioctl
int (*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned long arg);
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_ops结构体

```
// 处理32位的ioctl（可选）
int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
                       unsigned long arg);

// 控制器特定的mmap
int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);

// 保存当前的硬件状态
void (*fb_save_state)(struct fb_info *info);

// 恢复被保存的硬件状态
void (*fb_restore_state)(struct fb_info *info);

// 获得给定的可变参数的能力
void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps, struct
fb_var_screeninfo *var);
};
```

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info操作
    - 定义在<linux/fb.h>中
    - 实现在内核源代码drivers/video/fbmem.c中
    - 帧缓冲设备驱动为上层提供系统调用，也为下一层的特定硬件驱动提供接口
    - 底层的硬件驱动需要用到这些接口来向系统内核注册它们自己

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info操作
    - 分配fb_info结构体
      - struct fb_info *framebuffer_alloc(size_t size, struct device *dev);
        - » size: 待和fb_info一起分配的私有数据大小
        - » dev: 指向设备的指针
        - » 返回: 分配到的fb_info结构体指针，失败返回NULL
    - 释放fb_info结构体
      - void framebuffer_release(struct fb_info *info);
        - » info: 待释放的fb_info结构体指针

# Linux设备驱动实例

- 帧缓冲(Frame Buffer)
  - fb_info操作
    - 注册帧缓冲控制器驱动
      - int register_framebuffer(struct fb_info *fb_info);
        - » fb_info: 待注册的帧缓冲控制器的fb_info结构体指针
    - 注销帧缓冲控制器驱动
      - int unregister_framebuffer(struct fb_info *fb_info);
        - » fb_info: 待注销的帧缓冲控制器的fb_info结构体指针

# Linux设备驱动实例

- LCD驱动
  - LCD显示屏
    - LCD(Liquid Crystal Display)是笔记型计算机和掌上设备等嵌入式系统的主要显示设备
    - 液晶得名于其物理特性：它的分子晶体以液态存在而非固态；大多数液晶都属于有机复合物

# Linux设备驱动实例

- LCD驱动器时序要求
  - LCD模组数据手册
- LCD控制器RGB接口输出时序
  - S3C6410芯片数据手册

# Linux设备驱动实例

- LCD驱动
  - LCD控制器
    - S3C6410芯片LCD控制器分析
  - LCD驱动
    - LCD驱动代码
  - LCD驱动代码测试
    - 编写应用程序测试LCD显示

# 谢谢！



Technology Tarena 达内科技