

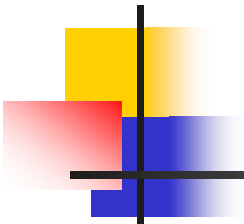
# 嵌入式Linux学习七步曲

## Sailor\_forever(扬帆)

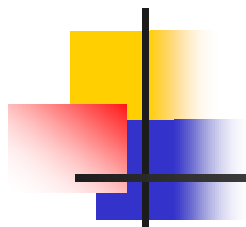
自由传播 版权所有 翻版必究



# 八一卦-我是who

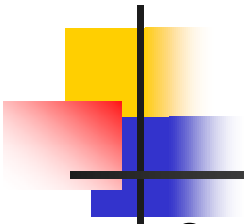
- 
- 
- n 目前就职于通信行业某外企研发中心
  - n 参与校园招聘和社会招聘的技术面试工作
  - n 5年嵌入式软件开发经验，擅长嵌入式Linux开发；
  - n 接触的软硬件平台包括ARM，DSP，PowerPC，uC/OS-II，Linux，VxWorks及OSE

# 八一卦-我是who



- n 嵌入式Linux七步曲 学习群 交流讨论 资源共享
- n 群号 107900817
- n 7steps2linux@gmail.com
- n [http://blog.csdn.net/sailor\\_8318](http://blog.csdn.net/sailor_8318)

# 嵌入式水平小调查



---

n 0—3个月

n 3—6 个月

n 1年左右

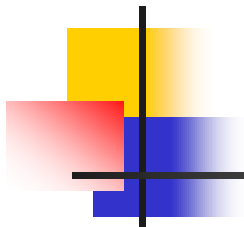
n 2年以上

n 多少人参加过系列交流会？



# 嵌入式Linux学习七步曲

- 
- 1 Linux主机开发环境
  - 2 嵌入式Linux交叉开发环境
  - 3 Linux系统bootloader移植
  - 4 Linux的内核移植
  - 5 **Linux的内核及驱动开发**
  - 6 文件系统制作
  - 7 Linux的高级应用编程



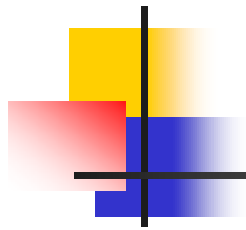
# Volunteer Task

## n 宗旨

- n 鼓励大家实际的参与嵌入式Linux的开发
- n 自己解决动手解决问题
- n 总结记录、分享
- n 形成知识库
- n 采用统一的模板，争取成为系列交流会的特色项目
- n 扩大BUPT BES的影响力，创造品牌

## n 运作

- n 下次交流会之前完成上次的总结文档
- n Share给大家，提建议意见
- n 每次交流会颁奖鼓励
- n 最终将评出 STAR Volunteer



# Volunteer Task

n Logo

《嵌入式 Linux 学习七步曲》  
BUPT BES 系列交流会 Volunteer Task

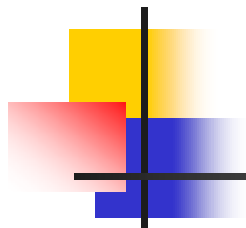


# Key To Success

- n Google、Baidu
- n 理论 + 实践（开发板）
- n 勤于思考，善于总结
- n 多上相关技术论坛，他山之石可以攻玉
- n 良好的文档撰写习惯
- n Passion！








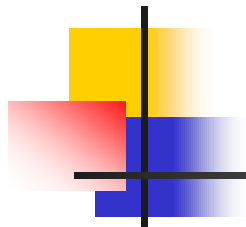
## CHAPTER

# 5 Linux的内核 及驱动开发



# 主要内容

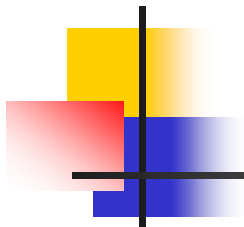
- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制



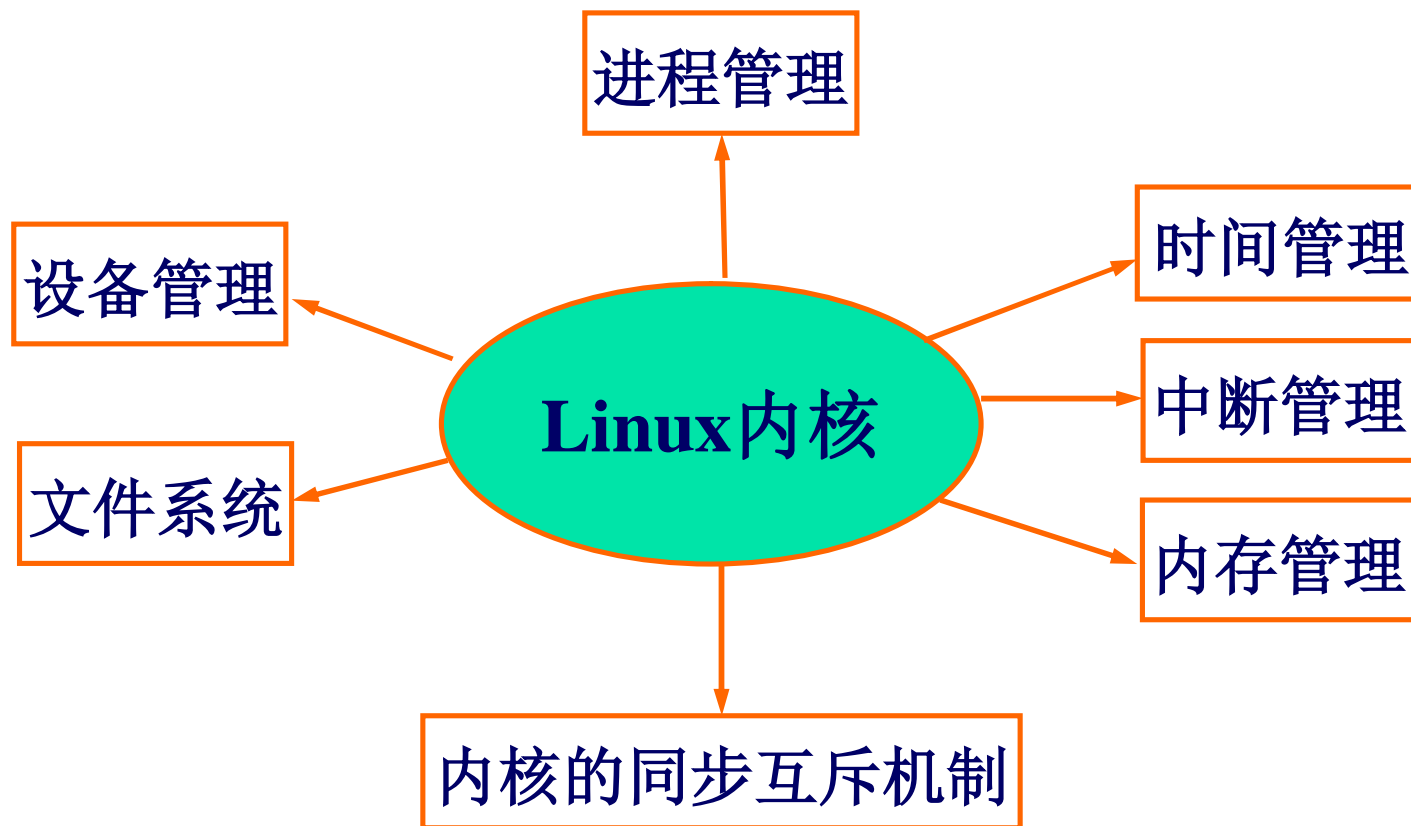
# 内核概述

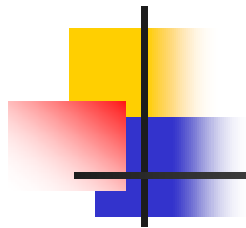
---

- n 内核组成
- n 内核源代码
- n 内核空间 and 用户空间
- n 内核函数和C库



# 内核组成





# 内核源代码

## n 下载

n <http://www.kernel.org/pub/linux/kernel/>

## n 在线浏览

n <http://lxr.linux.no/#linux+v2.6.25/>

## n 安装

### n PC

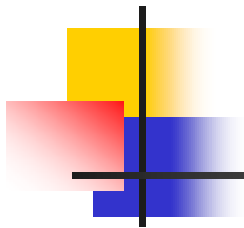
n 必须和发行版的Linux版本相对应

n apt-cache search linux-source

n sudo apt-get install linux-source-x.x.x

### n 嵌入式平台

n 直接下载解压缩即可



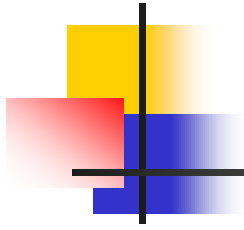
# 内核空间和用户空间

- n 为什么这么划分？
  - n CPU的运行级别
  - n 特权模式和普通模式
  - n 安全，入口唯一
- n 如何划分
  - n 0-3G 用户空间
  - n 3G-4G 内核空间
  - n 此分界点可配置，但一般为3G(0xC000 0000)
- n 如何交互
  - n 系统调用



# 内核函数和C库


- n 内核代码和C库独立，无法相互链接
- n 内核代码作为一个整体，可以相互动态链接
- n 内核负责维护内核的所有外部符号表，包括数据和代码
  - n `cat /proc/kallsyms`
- n 用户程序可以链接C库
- n `printf`和`printk`的区别

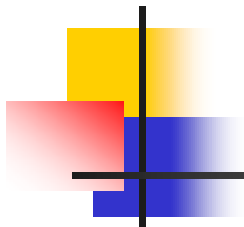






# 主要内容

- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制



# 设备驱动管理

---

- n 何谓驱动程序
- n 驱动程序的类型
- n 字符设备的驱动架构
- n 如何访问驱动程序
- n 驱动的调试
- n 2.6内核的驱动模型



# 何谓驱动程序

- n 以模块的形式位于内核之中
- n 在操作系统和硬件之间建立访问通道
- n 特定的硬件需要特定的驱动，也有所谓的  
万能驱动
- n 用户空间通过设备节点/dev/xxx对设备进行抽象管理与控制
- n 设备节点不是必需的
- n 所有的设备节点作为文件访问
- n 统一的用户接口



# 驱动程序的类型

---

## n 分类

- n 字符设备(char device)

- n 块设备(block device)

- n 网络接口(network device)

## n 如何查看类型

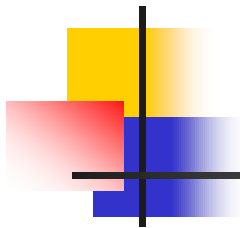
- n `cat /proc/devices`

- n `ls -l /dev`



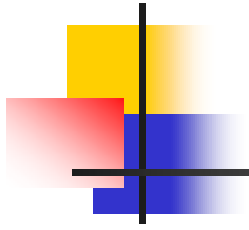
# 字符设备和块设备

- n 字符设备：以字节流的形式被访问的设备。它通过设备文件节点被访问。字符设备与一般文件(regular file)的区别：
  - n 可以在一般文件中前后移动(lseek), 但只能顺序访问字符设备.
  - n 当然, 也有特例
- n 块设备：能支持文件系统的设备
  - n 传统的UNIX: 只能以block(512B)为单位访问块设备
  - n Linux: 能以访问字符设备的方式访问块设备, 即以字节为单位访问块设备.
- n Linux中字符设备与块设备的区别
  - n 内核内部对数据的组织和管理不同, 对驱动开发者来说透明
  - n 接口不同: 使用两套不同的interface

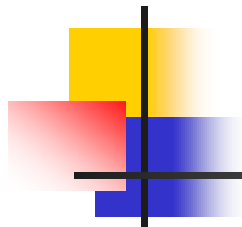


# 网络设备

- n 网络接口：能与其他主机通信的网络设备
  - n 它可以是硬件设备，也可以是软件设备，比如loopback.
  - n 网络接口只管收发数据包，而不管这些数据包被什么协议所使用
  - n 不同于字符设备和块设备，网络接口没有对应的文件系统节点. 虽然可以通过类似eth0这样的"文件名"来访问网络接口，但文件系统节点中却没有针对网络接口的节点
  - n 内核与网络接口之间的通信也不同于内核与字符/块设备之间的通信(read, write)，它们之间使用特定的传输数据包的函数调用



Q & A ?



# 字符设备的驱动架构

- n 字符驱动概念
- n 设备编号的表示形式、分配及释放
- n 模块组成
  - n 模块初始化/卸载
  - n `file_operations`操作方法
  - n 其他信息
- n 字符驱动的注册和释放
- n 模块加载卸载





# 字符驱动概念

- n 字符设备可以通过文件系统来存取
  - n 字符设备一般位于/dev下
  - n 有c标志的是字符设备
  - n 有b标志的是块设备
- n 主设备号决定驱动的种类
- n 次设备号决定使用哪个设备，默认从0开始编码
- n 一个设备节点唯一代表一个设备



# 设备编号的内部表达

- n `dev_t` 类型(在 `<linux/types.h>` 中定义)用来持有设备编号 -- 主次部分都包括
  - n 获得一个 `dev_t` 的主或者次编号, 使用
    - n `MAJOR(dev_t dev);`
    - n `MINOR(dev_t dev);`
  - n 转换为一个 `dev_t`, 使用:
    - n `MKDEV(int major, int minor);`



# 分配和释放设备编号

## n 分配指定的主设备号

n `int register_chrdev_region(dev_t first, unsigned int count, char *name);`

## n 动态分配主设备号

n `int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);`

## n 释放

n `void unregister_chrdev_region(dev_t first, unsigned int count);`



# 模块初始化

```
static int __init initialization_function(void)
{
    /*initialization code here*/
}
module_init(initialization_function);
```

n static

n int

n \_\_init(特定级别)

n module\_init



# 模块卸载

---

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
module_exit(cleanup_function);
```

n static

n void

n \_\_exit

n module\_exit



# file\_operation示例

```
static const struct file_operations i2cdev_fops = {  
    .owner          = THIS_MODULE,  
    .llseek         = no_llseek,  
    .read           = i2cdev_read,  
    .write          = i2cdev_write,  
    .ioctl          = i2cdev_ioctl,  
    .open           = i2cdev_open,  
    .release        = i2cdev_release,  
};  
n THIS_MODULE  
n “ ”  
n .
```



# 其他信息

---

## n 头文件

- n `#include <linux/module.h>`
- n `#include <linux/init.h>`

## n 导出符号表

- n `EXPORT_SYMBOL(i2c_smbus_xfer);`

## n 版权申明

- n `MODULE_LICENSE("GPL");`

## n 注释信息

- n `MODULE_AUTHOR("Simon G. Vogl <simon@tk.unilinz.ac.at>");`
- n `MODULE_DESCRIPTION("I2C-Bus main module");`

## n 查看工具

- n `lsmod`
- n `modinfo xxx`



# 字符驱动的注册和释放

## n 功能

### n 字符设备驱动模块初始化

- n 创建并注册设备结构
- n 创建驱动自身数据结构和相关资源
- n 注册中断服务程序

### n 字符设备驱动模块卸载

- n 关中断
- n 释放中断
- n 释放驱动数据结构和资源
- n 注销并删除设备结构





# 字符驱动的注册和释放

## n API

### n 字符设备驱动模块初始化

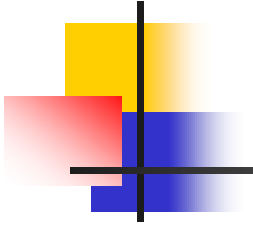
n static inline int register\_chrdev(unsigned int major, const char \*name, const struct file\_operations \*fops)

### n 字符设备驱动模块卸载

n static inline void unregister\_chrdev(unsigned int major, const char \*name)

n major 主设备编号

n name为字符设备在用户空间的名字(/proc/devices)，而非/dev/xx下文件的名称



# 字符驱动的注册和释放

## n 设备节点的创建

### n 静态创建

- n 预先创建，需要主设备号和次设备号
- n 浪费空间，大部分设备节点未使用

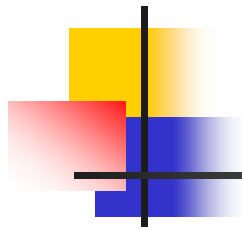
### n 动态创建

- n 2.4 内核，devfs
- n 2.6内核，sysfs和udev



# 注册和释放的错误处理

- n 为什么会出现错误?
  - n 资源已经被占用，如设备号、中断号
  - n 内存不够
- n 基本规则
  - n 申请和释放逆序
- n 策略
  - n goto
  - n 设置成功标志，调用统一的清除函数



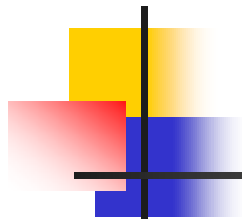
# goto

```
int __init my_init_function(void)
{
    int err;

    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* success */

fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}
```

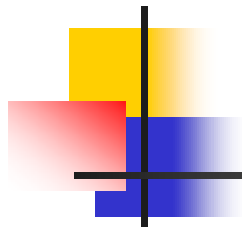


# 标志法

---

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff();
    return;
}
```



# 标志法

```
int __init my_init(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item1 || !item2)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* success */

fail:
    my_cleanup();
    return err;
}
```



# 模块加载卸载

## n 特点

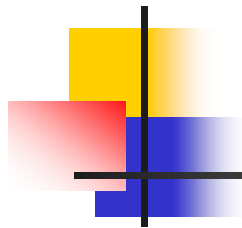
- n 传统的操作系统中OS和应用程序编译链接在一起
- n 为内核提供了动态扩展的能力
- n 尤其便于调试驱动程序

## n API

- n insmod xx.ko
- n lsmod
- n modinfo
- n modprobe
- n rmmod modname


## n 原理

- n 利用内核符号表实现了动态链接
- n system.map 内核的静态符号表
- n cat /proc/kallsyms或者ksyms(2.4)



# 模块的实现细节

## n ELF映像组成

<code>.text</code>	<i>instructions</i>
<code>.fixup</code>	<i>runtime alterations</i>
<code>.init.text</code>	<i>module init instructions</i>
<code>.exit.text</code>	<i>module exit instructions</i>
<code>.rodata.str1.1</code>	<i>read-only strings</i>
<code>.modinfo</code>	<i>module macro text</i>
<code>__versions</code>	<i>module version data</i>
<code>.data</code>	<i>initialized data</i>
<code>.bss</code>	<i>uninitialized data</i>
 <code>other</code>	





# 模块的实现细节

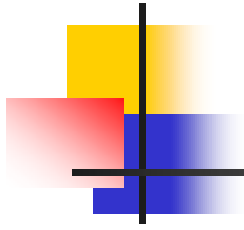
## n 加载过程

- n 分配临时内存
- n 内核版本检查
- n 解析引用的内核符号，重定位
- n 将解析后的映像拷贝到内核空间
- n 获得初始化指针
- n 调用初始化函数



# 模块依赖

- n 建立依赖关系
  - n `depmod -A`
  - n `/lib/modules/version-xxx/modules.dep`
- n 自动根据依赖关系进行加载
  - n `modprobe destmod.ko`
- n 模块依赖的典型问题
  - n `insmod:unresolved symbol`, 引用未定义的符号
  - n `CONFIG_MODVERSIONS` 内核选项
  - n `CFLAGS += -DMODVERSIONS -include /usr/src/linux/include/linux/modversions.h`





# 如何访问驱动程序

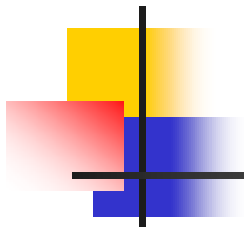
---

## n 前提

- n 存在设备节点
- n 驱动已经加载

## n 工具

- n Open
- n Read
- n Write
- n Io control
- n close



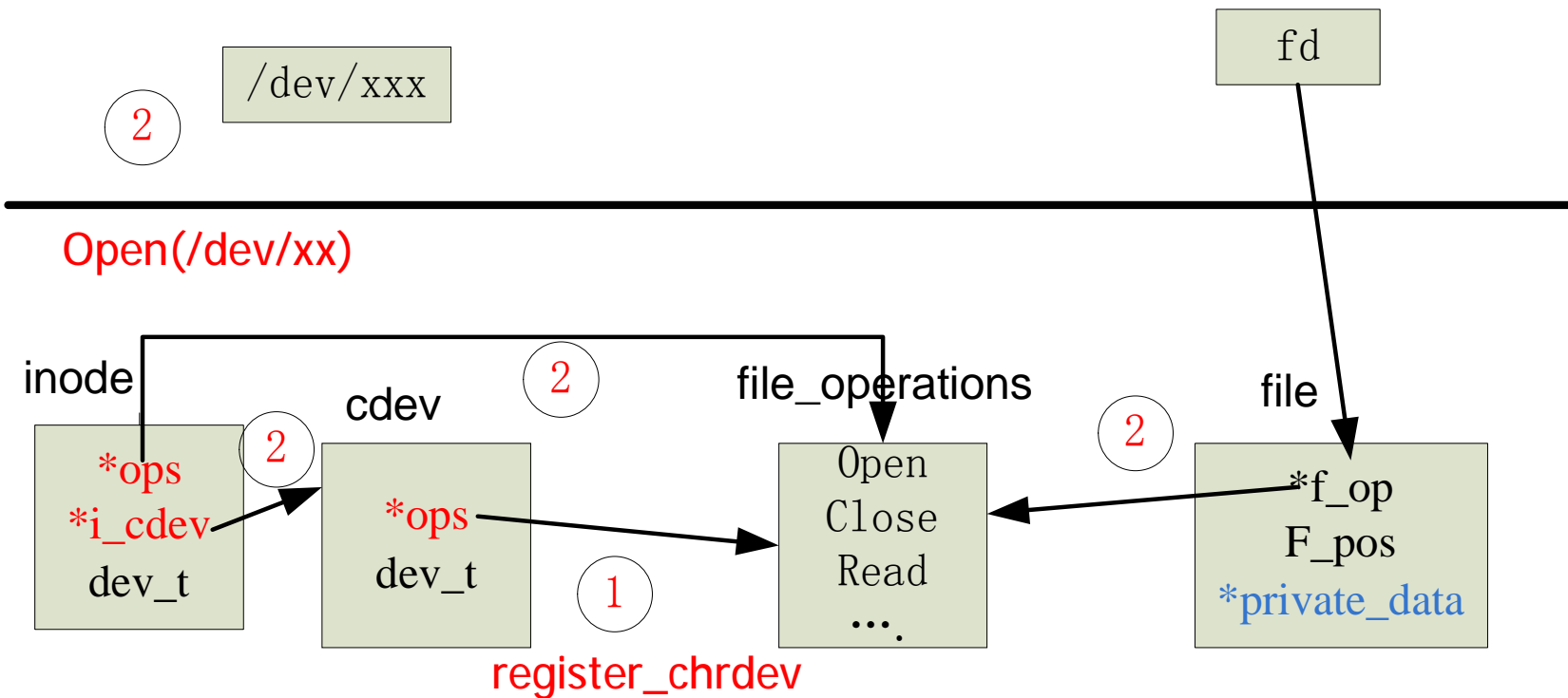
# 如何访问驱动程序

## n 管理

- n struct file\_operations, 驱动操作方法
- n struct cdev, 字符设备
- n struct dev\_t, 主从设备号
- n struct inode, 设备文件实体
- n struct file, 文件打开后临时建立的内核对象

# 如何访问驱动程序

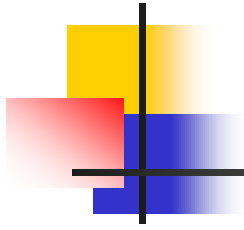
n 管理



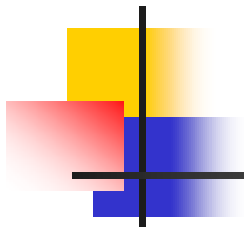


# 如何访问驱动程序

- n 同一个驱动如何区分各个从设备
  - n `/dev/xxx`唯一代表一个从设备
  - n `int (*open) (struct inode *, struct file *)`
  - n Open方法根据inode参数获得从设备信息，并将其保存在file 的private指针中
  - n `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`
  - n Read、write等方法根据file 指针的private参数来得知当前本驱动操作的对象



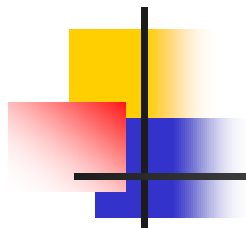




# 驱动的调试

---

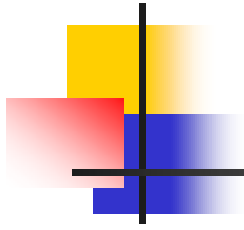
- n 动态加载卸载
- n Printk
- n 模块参数
- n Proc文件系统
- n Io control

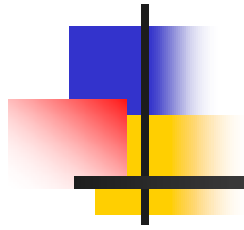


## 2.6内核的驱动模型


---

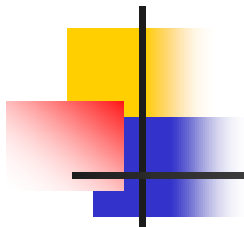
n TBD





# 主要内容

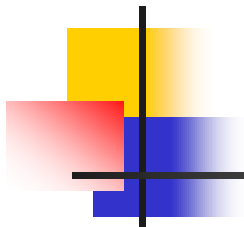
- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制



# 进程管理

---

- n 何谓进程?
- n 进程的表现形式
- n 进程状态
- n 进程调度



# 何谓进程?

---

- n 定义
- n 进程和程序的区别
- n 进程和用户线程的区别
- n 进程和用户及内核线程的区别



# 何谓进程？

---

## n 定义

n 由代码段（text）、用户数据段（user segment）以及系统数据段（system segment）共同组成的一个动态执行环境

n 代码段可共享

n 用户数据段存放全局变量

n 系统数据段为进程的管理控制信息

# 进程和程序的区别

## n 程序

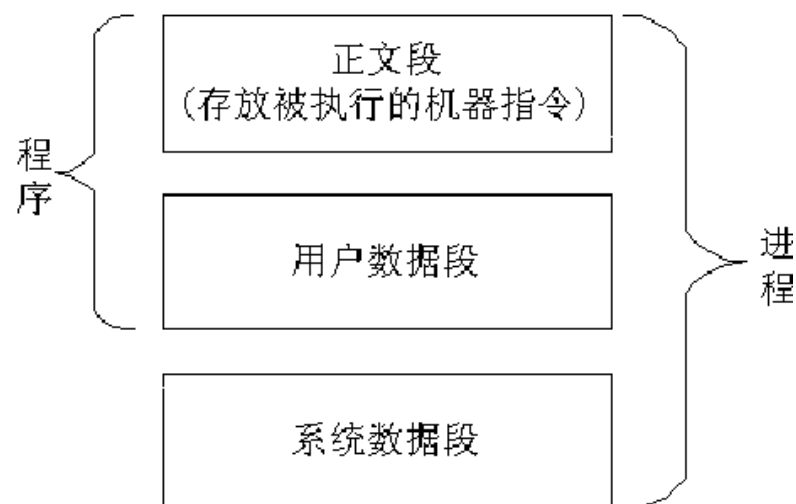
- n 静态对象，纯粹的数据。  
编译后形成的可执行代码

- n 由代码段、数据段、**BSS**段等组成，是进程的一部分

## n 进程

- n 程序的动态实例

- n 同一个程序可以有多个  
**动态实例**，多份拷贝







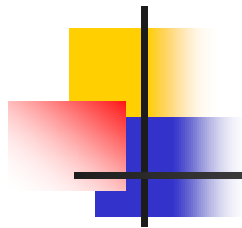
# 进程和用户线程的区别

## n 进程

- n 资源分配和管理的最小单位
- n 具备独立的地址空间
- n 创建销毁慢
- n 内核调度的基本单位
- n 上下文切换开销大

## n 线程

- n 进程的子集，由线程库调度，程序执行的最小单位
- n 多线程模型设计使程序更简洁明了
- n 更好的支持SMP以及减小上下文切换开销



# 进程和用户及内核线程的区别

## n 进程

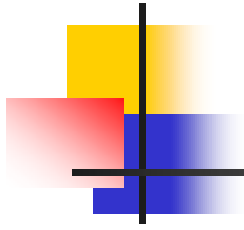
- n 内核调度的基本单位
- n 包括用户空间和内核空间两部分

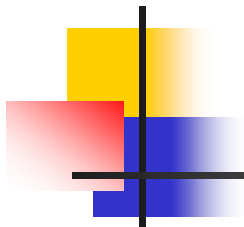
## n 用户线程

- n 线程库调度，只有用户空间

## n 内核线程

- n 内核调度，只有内核空间
- n 其他和进程一样

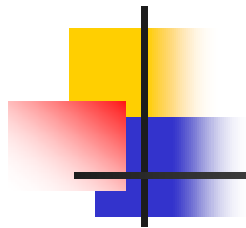




# 进程管理

---

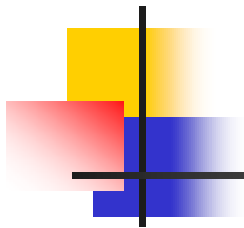
- n 何谓进程?
- n 进程的表现形式
- n 进程状态
- n 进程调度



# 进程的表现形式

---

- n 进程的组成
- n 进程的资源分配
- n 进程的组织形式



# 进程的组成

## n 定义

- n 由task\_struct数据结构来描述，简称PCB(Process Control Block)
- n TCB，大多数OS中所谓的任务控制块

## n 组成

- n 进程状态（State）
- n 进程调度信息
- n 进程通信有关信息
- n 时间和定时器信息（Times and Timers）
- n 文件系统信息（File System）
- n 虚拟内存信息（Virtual Memory）
- n 页面管理信息
- n 和处理器相关的环境（上下文）信息



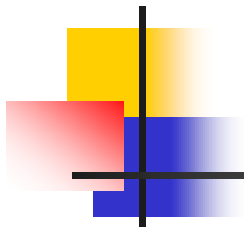
# 进程的资源分配

## n 数据结构

```
n union task_union {  
n     struct task_struct task;  
n     unsigned long stack[2048];  
n };
```

## n 特点

- n 内核栈和PCB共享8K的地址空间
- n 必须起始于 $2 \times 4K$ (PAGE\_SIZE)的页表边界
- n 内核栈起始于末端，并朝这个内存区开始的方向增长
- n 从用户态刚切换到内核态以后，进程的内核栈总是空的



# 进程的资源分配

## n Current宏

- n 当前进程指针
- n 可以看作全局变量来用，无需互斥

## n 原理

- n `task_struct = (struct task_struct *)  
STACK_POINTER & 0xffffe000`
- n 简洁高效
- n 否则查表



# 进程的组织形式

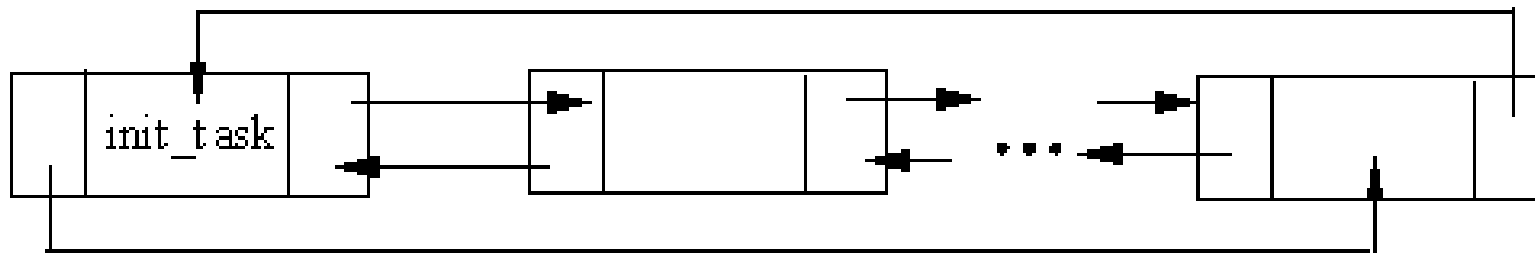
## n Hash表

n 根据pid快速进行查找等操作

## n 双向循环链表

n 可以反映进程间的创建关系和亲属关系

n 链表的头和尾都为init\_task





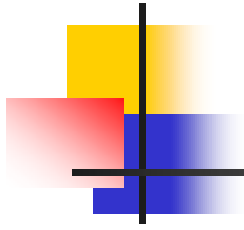
# 进程的组织形式

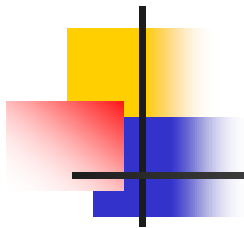
## n 运行队列

- n 所有处于可运行状态或正在运行状态的进程列表，内核根据此列表选择新的进程运行
- n 每个CPU只有一个

## n 等待队列

- n 所有需要等待除CPU以外的资源的进程都处在某个等待队列上
- n 可以有很多个
- n 资源就绪或者被信号唤醒时会从等待队列搬迁到运行队列中

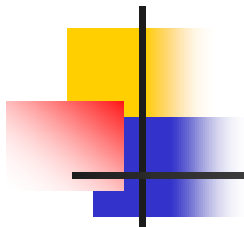




# 进程管理

---

- n 何谓进程?
- n 进程的表现形式
- n 进程状态
- n 进程调度



# 进程状态

- n 功能

- n 进程调度的参考依据

- n 类别

- n Running

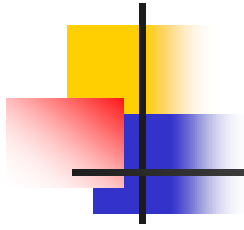
- n Pending

- n Interruptible

- n Uninterruptible

- n Stopped

- n Zombie



# 进程状态

## n Running

- n 进程处于运行（它是系统的当前进程）或者准备运行状态（它在等待系统将CPU分配给它）
- n 包括传统意义的running和ready态

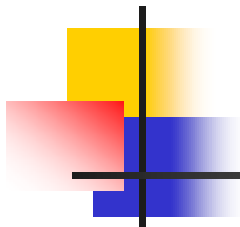
## n Pending

### n Uninterruptible

- n 正在等待系统关键资源，不可被异步信号打断，不能kill掉

### n Interruptible

- n 可被信号打断，大部分进程都处在这种状态。当被唤醒后，需要检查资源是否可用



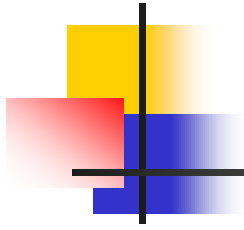
# 进程状态

## n Stopped

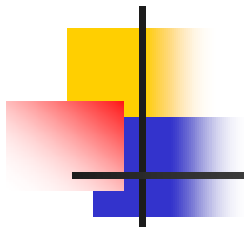
- n SIGSTOP信号使得进程进入stopped状态
- n SIGCONT信号，可以让其从TASK\_STOPPED状态恢复到TASK\_RUNNING状态
- n 主要用于调试

## n Zombie

- n 子进程消亡时，父进程未回收task\_struct资源，只剩下空壳，故为僵尸
- n 父进程可以通过wait系列的系统调用（如wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息
- n 杀死父进程可以让init进程回收僵尸进程的资源



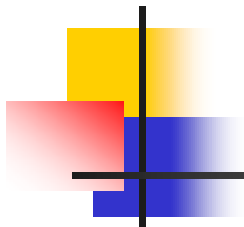




# 进程管理

---

- n 何谓进程?
- n 进程的表现形式
- n 进程状态
- n 进程调度



# 进程调度

---

- n 调度策略
- n 调度时间
- n 内核抢占
- n 优先级反转



# 调度策略

## n 定义

- n 决定什么时候以怎样的方式选择一个新进程运行的这组规则就是所谓的调度策略（scheduling policy），即何时如何选择被调度的进程。

## n 原则

- n 及时响应
- n 公平
- n 整体吞吐量

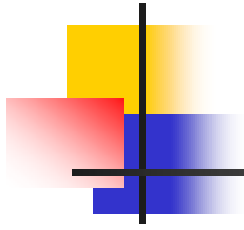
## n 类型

- n 实时进程
  - n SCHED\_RR
  - n SCHED\_FIFO
- n 普通进程
  - n SCHED\_OTHER



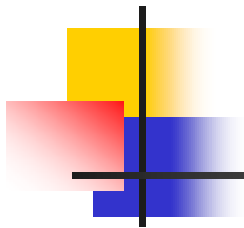
# 普通进程

- n 任何时刻的优先级都比实时进程低
- n 基于时间片来调度
- n 静态优先级
  - n 进程创建时赋予的优先级，决定了可以运行的时间片长短
  - n Nice值，值越大，优先级越低
  - n 进程创建后可以动态更改
- n 动态优先级
  - n 初始值为静态优先级
  - n 运行过程中降低
- n 调度依据
  - n 静态优先级+动态优先级，总体优先级是动态变化的
  - n 保证非实时进程可以交替运行，整体性能提高



# 实时进程

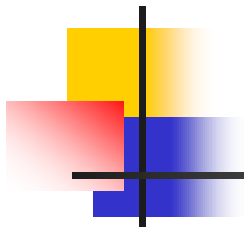
- n 任何时刻的优先级都比普通进程高，保证实时性
- n 优先级在运行过程是固定的
- n 不同优先级
  - n 按照优先级调度
- n 相同优先级
  - n **SCHED\_RR**
    - n 运行特定的时间片后让出CPU
    - n 此时间片取决于其静态优先级
  - n **SCHED\_FIFO**
    - n 在没有更高优先级的进程就绪时，始终占据CPU，除非主动放弃CPU



# 调度时机

---

- n 主动schedule放弃CPU
- n 系统调用完毕返回用户空间时
- n 中断处理完毕后
- n 时钟中断完毕后，检查时间片
- n 释放自旋锁时(2.6抢占内核)



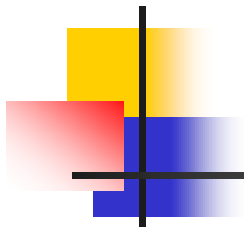
# 内核抢占

## n 不可抢占内核

- n 当进程在内核态运行时，除非主动放弃CPU，在返回用户空间前不能被其他进程抢占
- n 在单CPU上，大大简化了互斥

## n 为什么需要抢占特性

- n 实时性的要求。非抢占内核在内核中的停留时间无法预测，会影响实时性的应用
- n 此特性为2.6内核的可选项



# 内核抢占

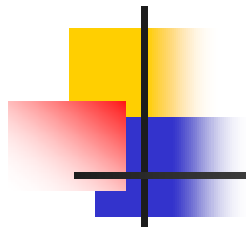
## n 抢占时机

- n 当有高优先级的进程就绪时，当抢占使能时，可立刻调度
- n 重新使能抢占(释放自旋锁)

## n 影响

- n 驱动程序时刻要考虑到可能的并发，对于全局资源需要做互斥

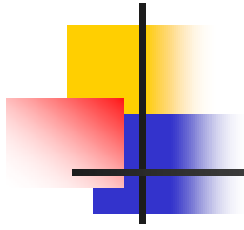




# 优先级反转


---

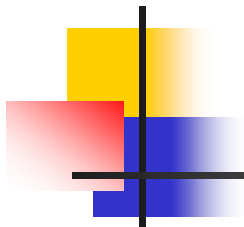
n TBD





# 主要内容

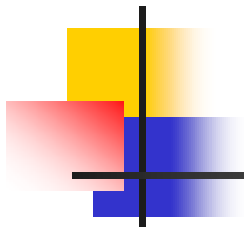
- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制



# 中断管理

---

- n 何谓中断？
- n 中断的处理流程
- n 面向对象的中断设计
- n 中断**API**
- n 中断共享
- n 中断下半部机制



# 何谓中断？

## n 定义

- n 中断现有的执行流程而处理其他事件

## n 作用

- n 响应异步事件，避免polling浪费CPU时间
- n 响应突发事件，提高响应速度

## n 类型

- n 可屏蔽中断
- n 不可屏蔽中断NMI



# 何谓中断？

---

## n 触发形式

- n 电平，高、低

- n 沿触发，上升沿、下降沿

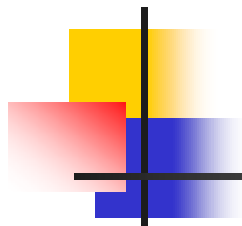
## n 和异常的区别

- n 异常一般为CPU内部异常事件，中断为正常事件

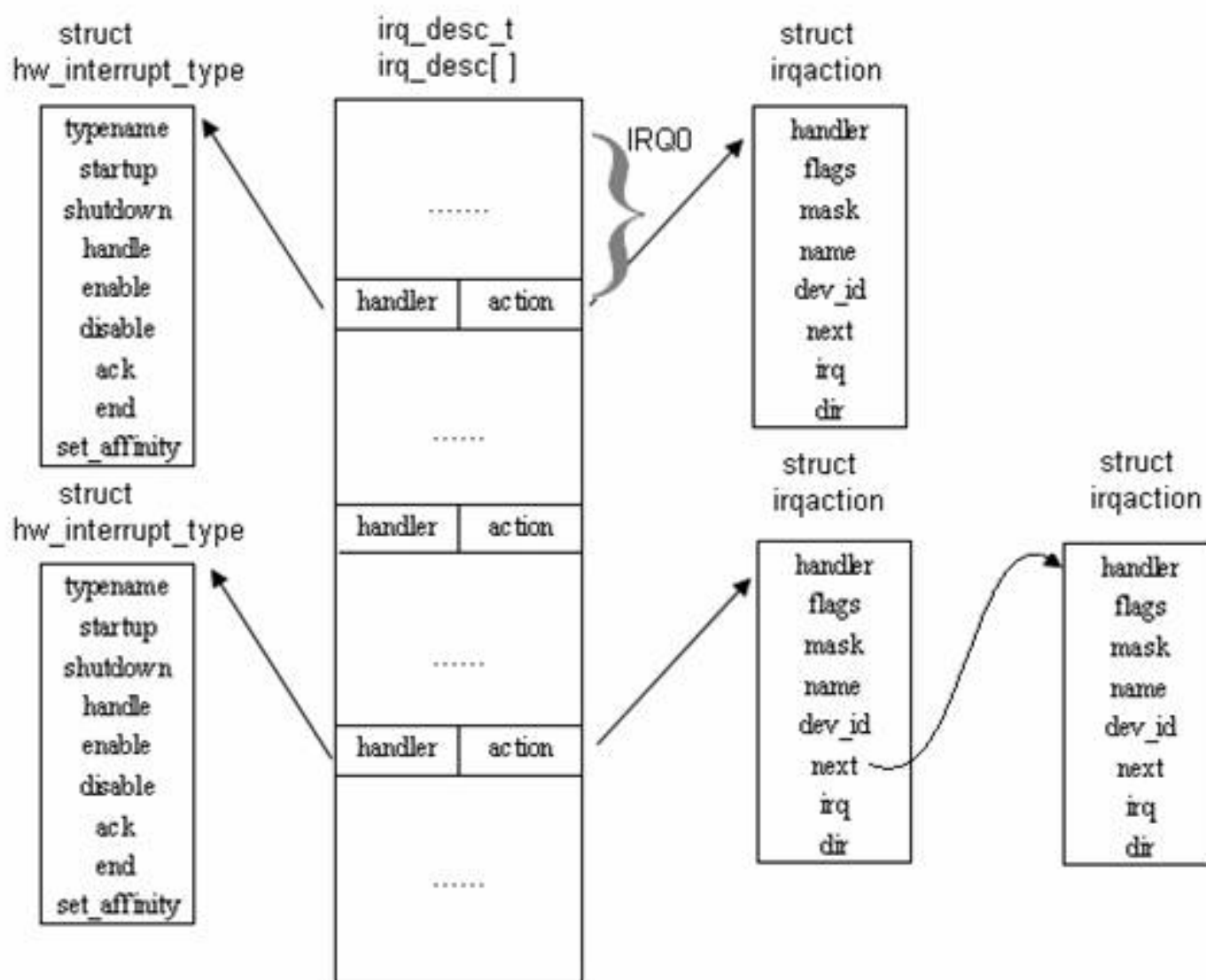


# 中断的处理流程

- n 保存现场
  - n PC指针自动跳转到IRQ异常向量的处理入口
  - n 保存现场至上一个被中断任务的PCB中，包括堆栈指针、PC等CPU的核心资源
- n 中断处理
  - n 查找中断号，检查中断服务程序是否挂接
  - n 执行向应的中断处理程序，中断清除
- n 中断返回
  - n 检查是否有软中断需要执行
  - n 重新进行可能的（取决于是否可抢占，是否内核空间）进程调度，没有更高优先级的进程就绪，则恢复原有进程的运行



# 面向对象的中断设计







# 中断API

## n 中断挂接和释放

n int request\_irq(unsigned int irq,  
irq\_handler\_t handler, unsigned long irqflags,  
const char \*devname, void \*dev\_id)

n Irq, 要分配的中断号

n Handler, 中断处理程序

n Irqflags, 与中断管理相关的选项的位掩码,  
共享、屏蔽之类的

n dev\_name, 中断属主

n dev\_id, 特定标识

n void free\_irq(unsigned int irq, void \*dev\_id)



# 中断API

---

## n 使能禁止

n void local\_irq\_disable(void);

n void local\_irq\_enable(void);

n unsigned long flags;

n void local\_irq\_save(unsigned long flags);

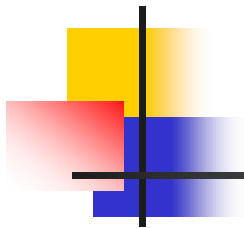
n void local\_irq\_restore(unsigned long flags);

n void disable\_irq(unsigned int irq);

n void enable\_irq(unsigned int irq);

## n 查看中断

n /proc/interrupts



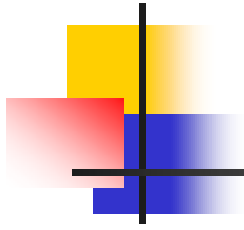
# 中断共享

## n 需求

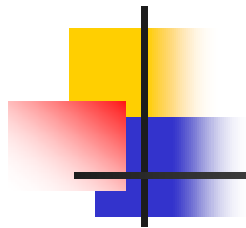
- n 中断线有限
- n 同一个中断线管理多个事件

## n 实现

- n `request_irq()` 的参数 `flags` 必须设置 `SA_SHIRQ` 标志
- n 对每个注册的中断处理程序来说，`dev_id` 参数必须惟一
- n 中断处理程序必须能够区分它的设备是否真的产生了中断



Q & A ?



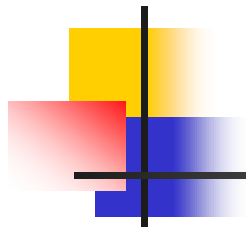
# 中断下半部机制

## n 需求

- n 中断处理时间应该尽量短，避免打断当前程序过久
- n 避免同级别的中断不能及时响应
- n 不能休眠，不能在进程上下文运行，所执行的功能有限

## n 解决方案

- n 分成上下两部分



# 中断下半部机制

## n 上下部分家的原则

### n 上半部

- n 任务对时间非常敏感
- n 任务和硬件相关
- n 任务要保证不被其他中断(特别是相同的中断)打断

### n 下半部

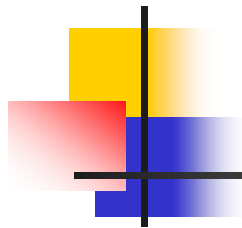
- n 其他所有任务，考虑放置在下半部执行
- n 尤其是可能休眠的工作



# 下半部机制的类型

## n Softirq

- n 类型有限，一般为32个
- n 一般用户不会采用softirq机制，对编程人员要求高
- n enum
- n {
- n   HI\_SOFTIRQ=0,
- n   TIMER\_SOFTIRQ,
- n   NET\_TX\_SOFTIRQ,
- n   NET\_RX\_SOFTIRQ,
- n   BLOCK\_SOFTIRQ,
- n   TASKLET\_SOFTIRQ
- n };
- n 当需要执行的软中断过多时，由ksoftirqd内核线程处理



# 下半部机制的类型

## n Softirq

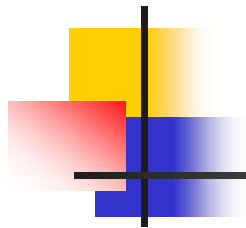
softirq\_action 结构表示，它定义在 <linux/interrupt.h> 中：

```
246 struct softirq_action  
247 {  
248     void    (*action)(struct softirq_action *);  
249     void    *data;  
250 };
```

Action: 待执行的函数;

Data: 传给函数的参数，任意类型的指针，在 action 内部转化





# 下半部机制的类型

## n tasklet

- n 由softirq实现

  - n HI\_SOFTIRQ

  - n TASKLET\_SOFTIRQ

- n 有更好的保护机制

- n 能实现softirq的所有功能

- n **通常**在原子上下文执行，除非软中断过多，被调度到softirqd内核线程上



# 软中断类型

## n tasklet

```
280struct tasklet_struct  
281{  
282    struct tasklet_struct *next;  
283    unsigned long state;  
284    atomic_t count;  
285    void (*func)(unsigned long);  
286    unsigned long data;  
287};
```

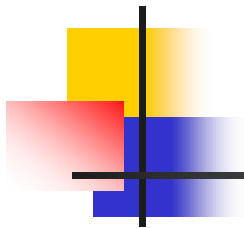
next: 链表中的下一个 tasklet;

state: tasklet 的状态;

count: 引用计数器;

func: tasklet 处理函数;

data: 给 tasklet 处理函数的参数



# 软中断类型

## n 工作队列(work queue)

- n 可运行在进程上下文，调度时机不确定，取决于系统当时的运行状态
- n 可以指定延时多少时间再触发，可利用内核 timer
- n 如果推后执行的任务需要在在一个tick之内处理，则使用软中断或tasklet，因为其可以抢占普通进程和内核线程

# 软中断类型

## n 工作队列(work queue)

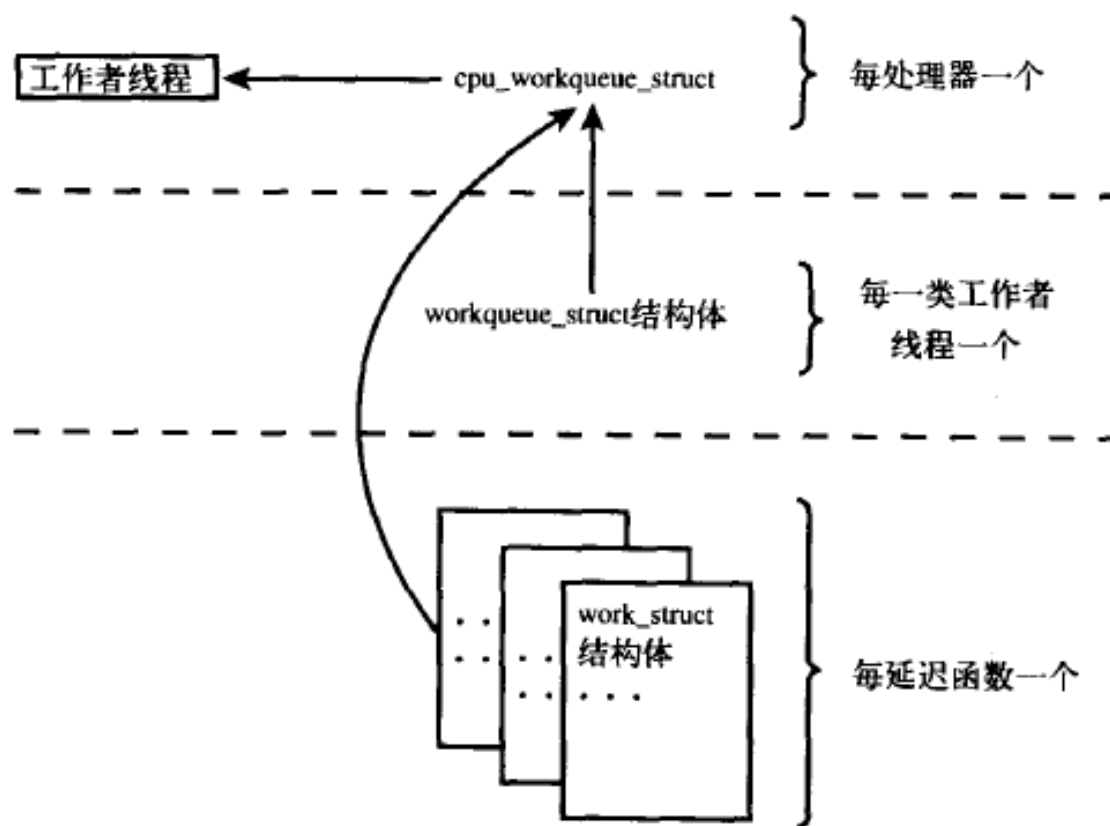
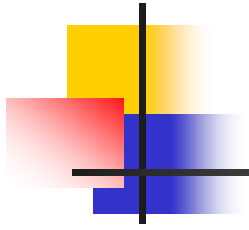



图7-1 工作(work)、工作队列和工作者线程之间的关系

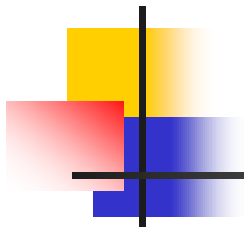


Q & A ?



# 主要内容

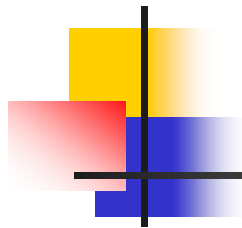
- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制



# 时间管理

---

- n 时间管理的意义
- n 内核定时器
- n 短延时和长延时



# 时间管理的意义

- n 为系统提供定时机制
- n 操作系统的调度需要时间片的概念，tick为操作系统时间的基本单位，一般为1ms(取决于CPU的主频和调度的粒度)
- n 系统时间为软件时间
- n RTC时间为硬件时间
- n 软件时间和硬件时间会阶段性的自动同步





# 内核定时器

## n 数据结构

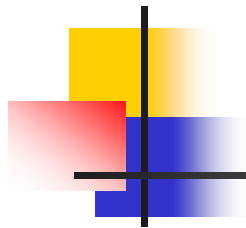
```
10 struct timer_list {  
11     struct list_head entry;  
12     unsigned long expires;  
13  
14     void (*function)(unsigned long);  
15     unsigned long data;  
16  
17     struct tvec_t base_s *base;  
18};
```



# 内核定时器

## n 数据结构

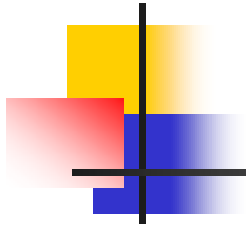
```
__66 struct tvec_t_base_s {  
__67     spinlock_t lock;  
__68     struct timer_list *running_timer;  
__69     unsigned long timer_jiffies;  
__70     tvec_root_t tv1;  
__71     tvec_t tv2;  
__72     tvec_t tv3;  
__73     tvec_t tv4;  
__74     tvec_t tv5;  
__75 } ____cacheline_aligned_in_smp;  
__76  
__77 typedef struct tvec_t_base_s tvec_base_t;
```



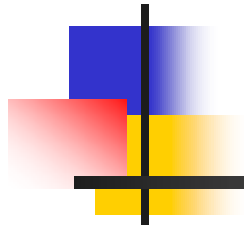
# 短延时和长延时

---


n TBD

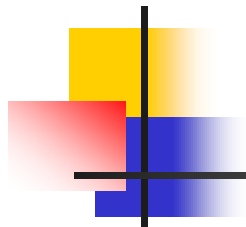


Q & A ?



# 主要内容

- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制




# 内存管理

---

- n 虚拟内存的意义
- n IO内存和普通内存
- n 内存管理算法



# 主要内容

- 
- 1 内核概述
  - 2 设备驱动管理
  - 3 进程管理
  - 4 中断管理
  - 5 时间管理
  - 6 内存管理
  - 7 内核的同步互斥机制

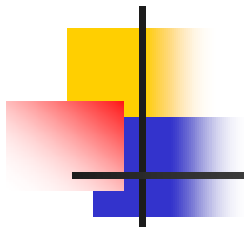


# 内核的同步互斥机制

---

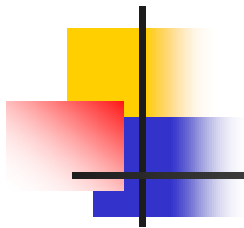
- n 休眠与同步
- n 等待队列
- n 等待event
- n Completion
- n 并发与竞态
- n 原子操作
- n 自选锁
- n 信号量
- n 互斥锁





# 休眠与同步

- n 驱动无法满足用户请求怎么办?
  - n Read调用无法读取到数据
  - n Write调用，缓冲区已满
- n 策略
  - n 非阻塞调用
    - n Open时指定non block
    - n 无法满足需求时立即返回
  - n 阻塞调用
    - n 无法满足需求时休眠
    - n 重新唤醒后，检查资源是否可用



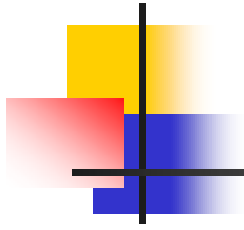
# 休眠与同步

## n 休眠的时机

- n 不能在原子上下文休眠
- n 拥有自选锁时不能休眠
- n 关中断时不能休眠
- n 拥有一个信号量时再休眠要防止自锁

## n 唤醒时机

- n 等待的资源可用
- n Interruptable进程被信号中断



# 休眠与同步

n 机制

n 等待队列

n event

n completion

n 信号量sem



# 等待队列

## n 数据结构

n `\include\linux\wait.h`

n `struct __wait_queue_head {`

n `spinlock_t lock;`

n `struct list_head task_list;`

n `};`

n `typedef struct __wait_queue_head`  
`wait_queue_head_t;`

n Lock为自旋锁，保护此队列

n `task_list`为等待某个资源的双向循环列表，包含了所有等待的任务

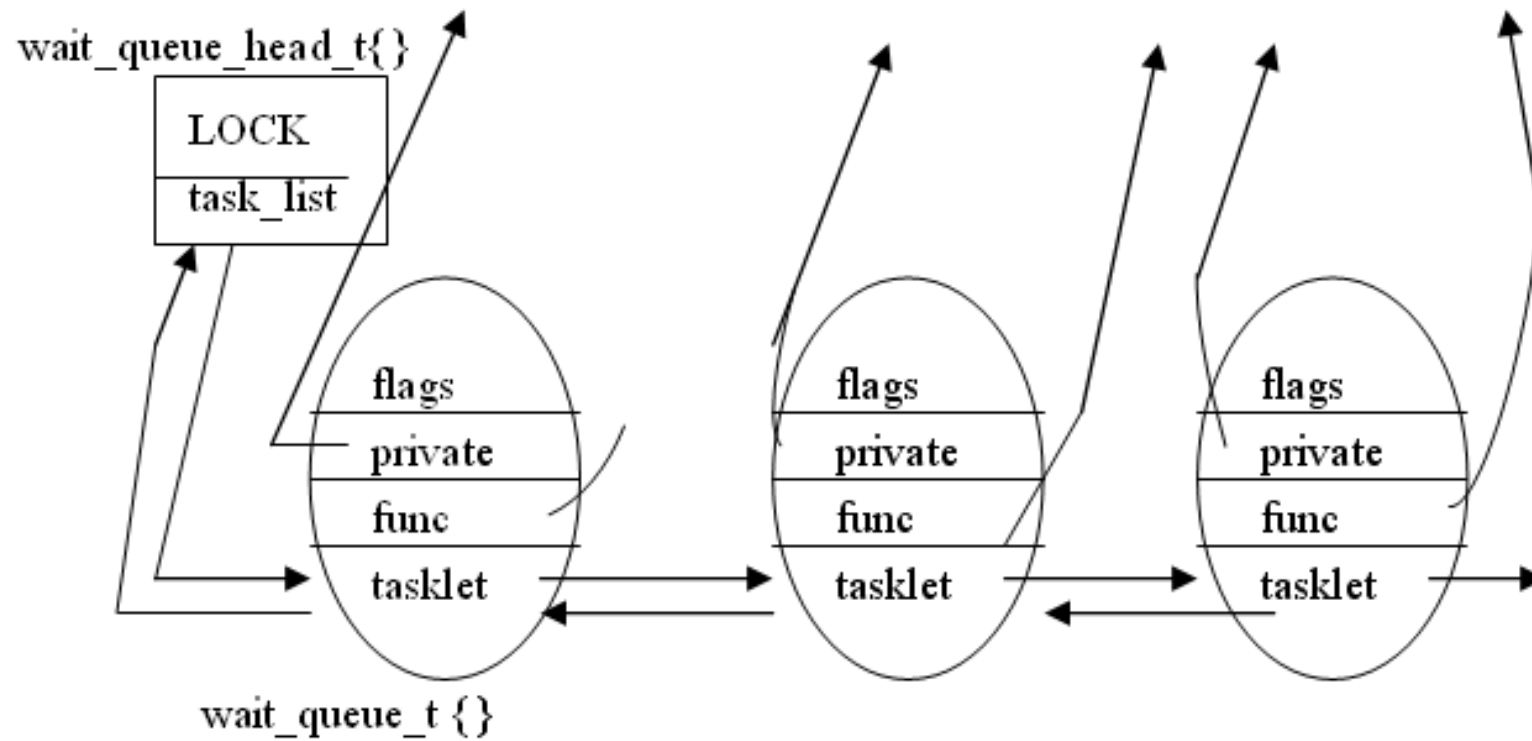


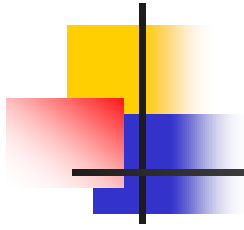
# 等待队列

## n 数据结构

```
n struct __wait_queue {  
n     unsigned int flags; //在等待队列上唤醒时是  
    否具备排他性  
n     #define WQ_FLAG_EXCLUSIVE      0x01  
n     void *private;  
n     wait_queue_func_t func;  
n     struct list_head task_list; //与该等待队列对  
    应的任务链表  
n };
```

# 等待队列

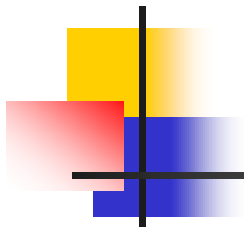




# 等待队列

## n API

- n DECLARE\_WAIT\_QUEUE\_HEAD(name)
- n void init\_waitqueue\_head(wait\_queue\_head\_t \*q)
- n DECLARE\_WAITQUEUE(name, tsk)
- n interruptible\_sleep\_on(wait\_queue\_head\_t \*q)
- n interruptible\_sleep\_on\_timeout (wait\_queue\_head\_t \*q, long timeout)
- n sleep\_on(wait\_queue\_head\_t \*q)
- n sleep\_on\_timeout(wait\_queue\_head\_t \*q, long timeout)



# 等待队列

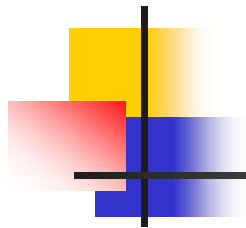
## n 问题

- n 事件就绪后可以唤醒所有在等待队列中挂起的进程，但是资源很可能只有一个，唤醒后必须检查资源是否可用，被唤醒的进程可能再次休眠
- n 频繁唤醒及休眠会浪费CPU的资源
- n 对码农的要求高

## n 解决方案

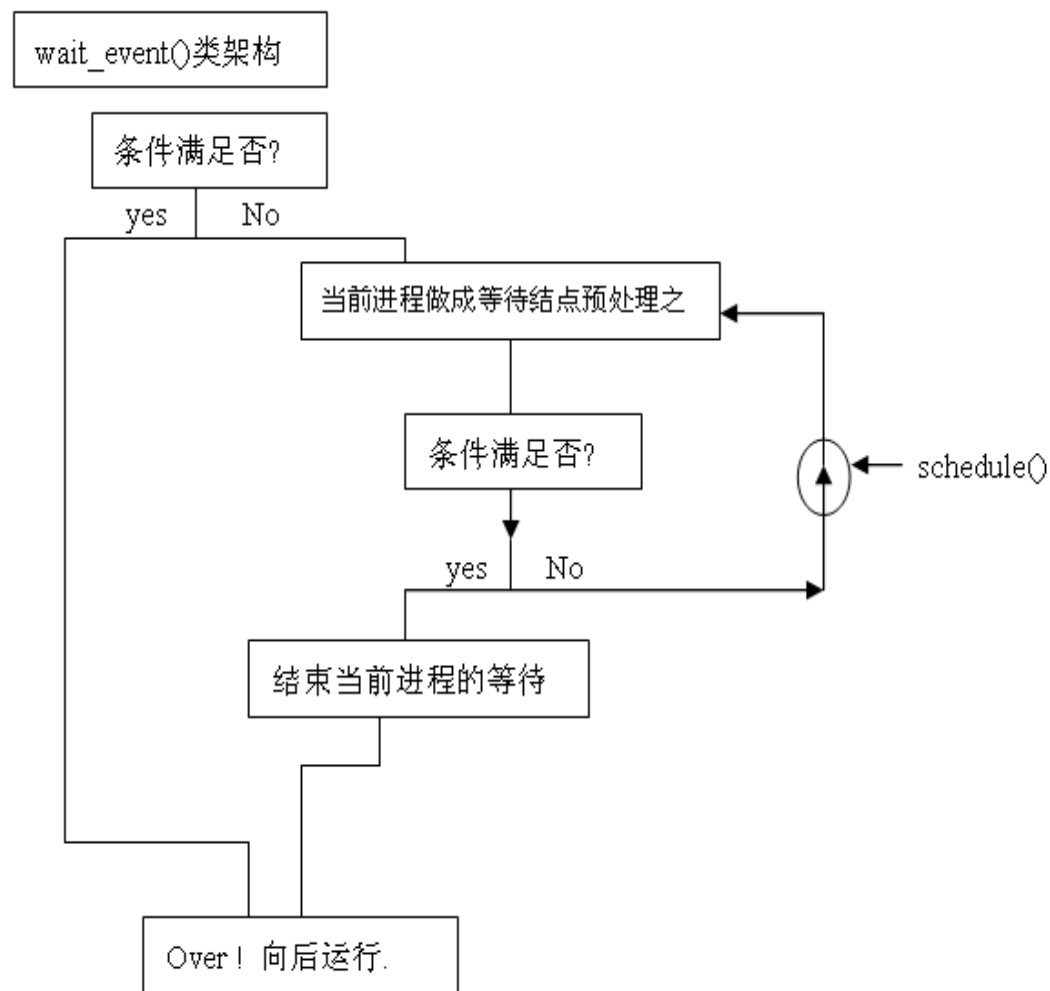
- n 排它等待
- n 封装接口，自动检查等待资源





# 等待event

## n 基本流程



# 等待event

## n 实现

```
#define __wait_event(wq, condition) \
do { \
    DEFINE_WAIT(__wait); \
    \
    for(;;) { \
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
        /// 添加到等待队列中，同时更改进程状态；若已经加入则不会重复添加 \
        if (condition) \
            break; \
        schedule(); \
    } \
    finish_wait(&wq, &__wait); \
} while (0) \
// “__”表示内部函数，默认为 condition 不满足，添加至等待队列，调度
```



# 等待event

## n 实现

```
#define wait_event(wq, condition)
do {
    if(condition)
        break;
    __wait_event(wq, condition);
}while(0)
```

//对外的接口函数，需要判断 `condition`，若假则等待；若真则直接退出

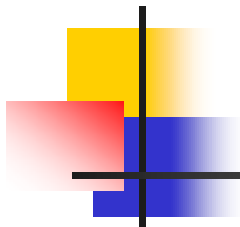


# 等待event

## n 实现

```
#define wait_event(wq, condition)
do {
    if(condition)
        break;
    __wait_event(wq, condition);
}while(0)
```

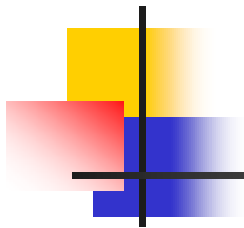
//对外的接口函数，需要判断 `condition`，若假则等待；若真则直接退出



# 等待event

## n API

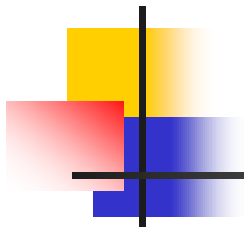
- n `wait_event(queue, condition)`
  - n 不可中断休眠，不推荐
- n `wait_event_interruptible(queue, condition)`
  - n 推荐，返回非零值意味着休眠被中断,且驱动应返回 - ERESTARTSYS
- n `wait_event_timeout(queue, condition, timeout)`  
`wait_event_interruptible_timeout(queue, condition, timeout)`
  - n 有限的时间的休眠；若超时，则不管条件为何值返回 0
- n 上述API为宏定义，`queue`都是“值传递”



# 等待event

## n API

- n `wake_up(wait_queue_head_t *queue)`
- n `wake_up` 唤醒队列中的每个非独占等待进程和一个独占等待进程
- n 和`wait_event` 匹配
- n `wake_up_interruptible(wait_queue_head_t *queue)`
- n `wake_up_interruptible` 同样, 但它跳过处于不可中断休眠的进程
- n 和`wait_event_interruptible(queue, condition)`匹配



# Completion

## n 功能

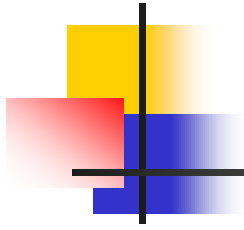
- n 允许一个线程告诉另一个线程某个工作已经完成，便于回收资源

## n 应用场合

- n 用在大型复杂的驱动程序中
- n 主要是驱动模块告诉内核线程释放资源

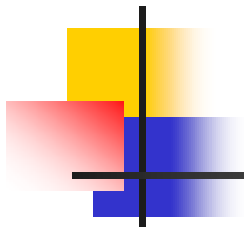
## n API

- n `void wait_for_completion(struct completion *c);`
- n 等待completion
- n `void complete(struct completion *c);`
- n 唤醒一个等待completion的线程



Q & A ?





# 并发与竞态

## n 根本原因

### n 访问共享资源

n 全局变量

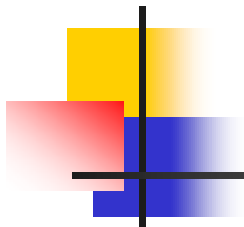
n 硬件资源

n 指针传递

## n 应对策略

n 尽量避免共享

n 互斥访问



# 并发类型

## n 单处理器

### n 2.6以前的内核

- n 无内核抢占
- n 硬件中断
- n 软中断

### n 2.6以后的内核

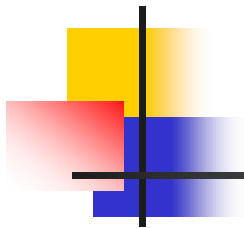
- n 配置了抢占内核的特性后，要注意内核抢占
- n 可以用自选锁来禁止短期抢占

## n SMP

### n 多CPU运行相同的代码

## n 并发严重性

### n SMP 》 2.6内核》 2.4内核



# 并发与竞态

---

- n 如何互斥
  - n 关中断
  - n 原子操作
  - n 自旋锁
  - n 信号量
  - n 互斥锁



# 原子操作

## n 原理

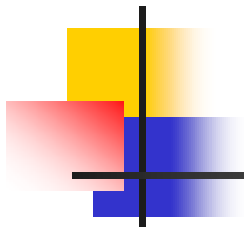
- n 临界段被包含在 **API** 函数中，不需要额外的锁定
- n C无法实现，需要依赖底层汇编

## n API

- n `int atomic_read(atomic_t *v)`
- n `void atomic_add(int i, atomic_t *v)`

## n 典型应用

- n 位操作
- n `void set_bit(nr, void *addr);`
- n `void clear_bit(nr, void *addr);`
- n `void change_bit(nr, void *addr)`



# 自旋锁

## n 特点

- n 自旋期间一直占用CPU(除非被中断)
- n 会使其他CPU忙等

## n 单处理器

- n 2.6以前的内核

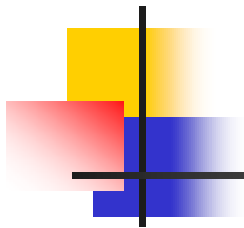
- n 空，无任何作用

- n 2.6以后的内核

- n 配置了抢占内核后，动态设置内核抢占的开关

## n SMP

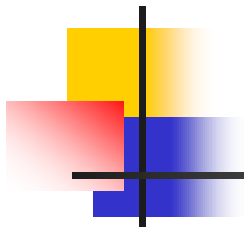
- n 多处理器间的互斥



# 自旋锁

## n 策略

- n 为了防止出现可移植性问题，代码需要按照 **SMP** 的抢占内核来设计
- n 短时间的互斥，忙等的效率比休眠唤醒高
- n 拥有自旋锁的代码必须是原子的，即拥有锁期间不能休眠
- n 必须获取多个锁时，应始终以相同顺序获取，这样可以防止互锁
- n 当我们拥有信号量和自旋锁的组合，必须先获得信号量



# 自旋锁

## n API

### n 初始化

- n `SPIN_LOCK_UNLOCKED (spinlock_t)`

- n `void spin_lock_init(spinlock_t *lock)`

### n 获得锁和解锁

- n `void spin_lock(spinlock_t *lock);/* 获得spinlock*/`

- n `void spin_unlock(spinlock_t *lock);`

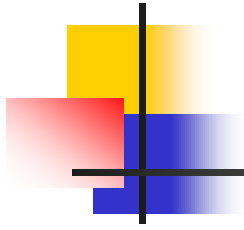
### n 中断互斥锁

- n `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`

- n `void spin_unlock_irqrestore`

### n 探测锁

- n `int spin_trylock(spinlock_t *lock)`



# 信号量sem

## n 特点

- n 用于单处理器中进程上下文对临界资源的保护
- n P、V操作
- n 既可用于同步也可以用于互斥
- n 互斥时初始值为可用状态
- n 当信号量不可用时，进程进入排它性等待队列，信号量再次可用时，只有一个进程被唤醒





# 信号量和自旋锁的区别

## n 粒度

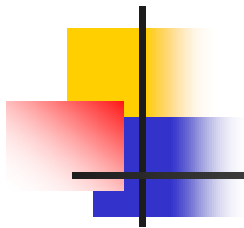
- n 自旋锁短期互斥
- n 信号量较长时间的互斥

## n 可否被抢占

- n 自旋锁会禁止抢占，也可选择性的禁止中断
- n 持有信号量时可被抢占

## n 应用场合

- n 仅进程上下文，信号量
- n 进程上下文，非常短的时间，优选自旋锁
- n 中断上下文，只能自旋锁



# 信号量

## n API

### n 初始化

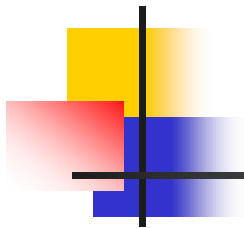
- n DECLARE\_MUTEX(name)
- n DECLARE\_MUTEX\_LOCKED(name);
- n void init\_MUTEX(struct semaphore \*sem);
- n void init\_MUTEX\_LOCKED(struct semaphore \*sem);

### n P、V操作

- n void down(struct semaphore \*sem);
- n int down\_interruptible(struct semaphore \*sem);
- n void up(struct semaphore \*sem); 出错时必须释放信号量, 防止死锁

### n 探测锁

- n int down\_trylock(struct semaphore \*sem);



# 信号量

## n 实现细节

- n `struct semaphore {`
- n   `atomic_t count;`
- n   `int sleepers;`
- n   `wait_queue_head_t wait;`
- n `};`
- n **Count**该信号量表示的可用资源数目
- n **Sleepers**, 睡眠在该信号量上的进程数目, 等于等待队列中的表项数
- n **Wait**内嵌的等待队列头结构
- n 信号量本身的互斥操作是由**wait**中的互斥锁提供的



# 互斥锁

## n 特点

- n 二值信号量的特例
- n 但比二值信号量更高效

## n 实现细节

- n `struct mutex {`
- n `/* 1: unlocked, 0: locked, negative: locked,`  
`possible waiters */`
- n `atomic_t` `count;`
- n `spinlock_t` `wait_lock;`
- n `struct list_head` `wait_list;`
- n `};`
- n 相比semaphere而言, mutex少了sleepers成员

