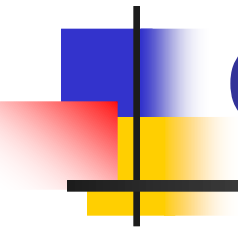
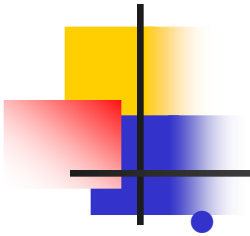


# Ch4 内核与驱动简介

Enyi Tang

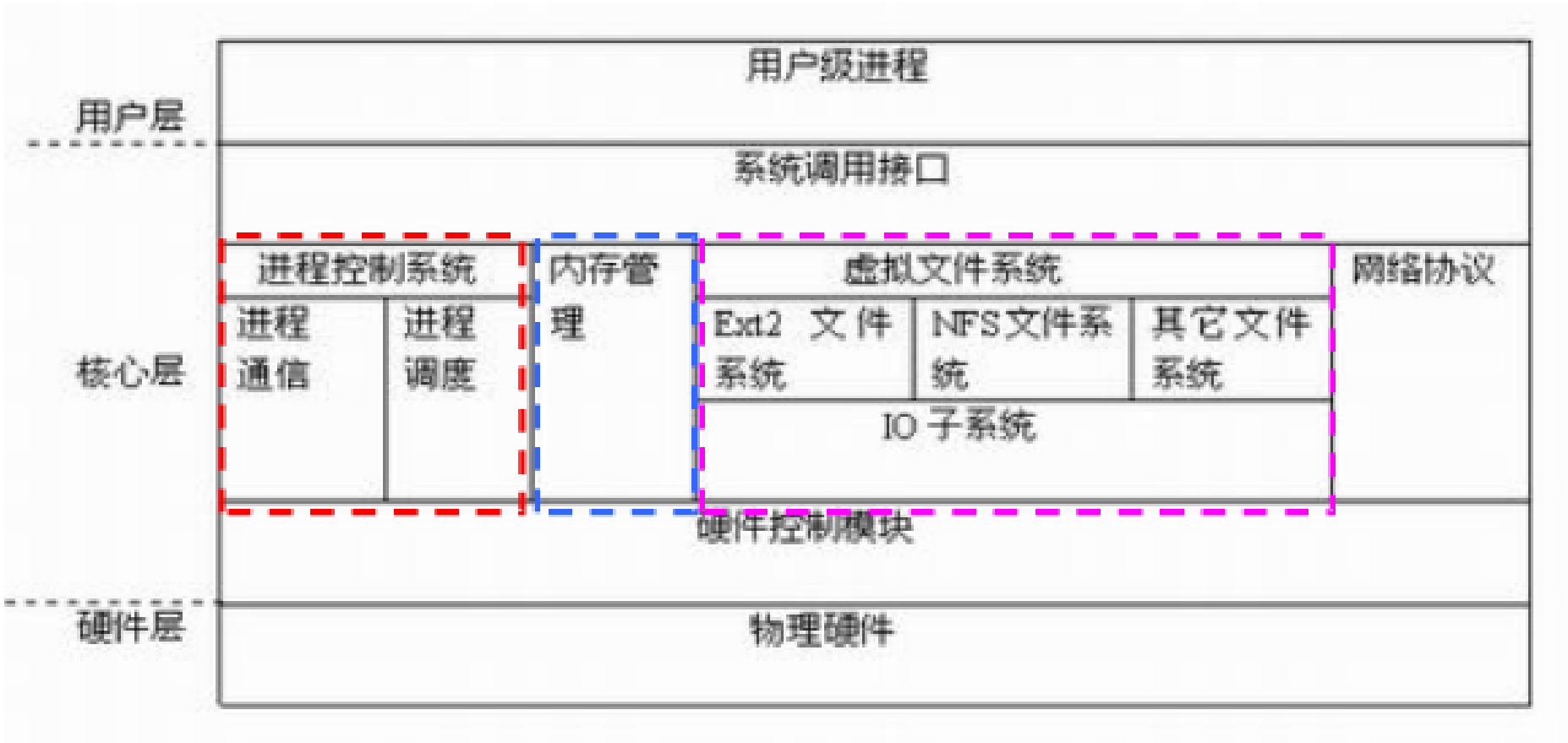
Software Institute of
Nanjing University



# Linux内核简介

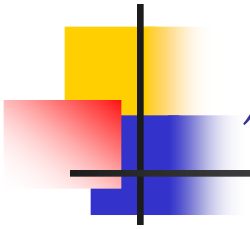
- 什么是内核
  - 操作系统是一系列程序的集合，其中最重要的部分构成了内核
  - 单内核/微内核
    - 单内核是一个很大的进程，内部可以分为若干模块，运行时是一个独立的二进制文件，模块间通讯通过直接调用函数实现
    - 微内核中大部分内核作为独立的进程在特权下运行，通过消息传递进行通讯
  - Linux内核的能力
    - 内存管理，文件系统，进程管理，多线程支持，抢占式，多处理支持
  - Linux内核区别于其他UNIX商业内核的优点
    - 单内核，模块支持
    - 免费/开源
    - 支持多种CPU，硬件支持能力非常强大
    - Linux开发者都是非常出色的程序员
    - 通过学习Linux内核的源码可以了解现代操作系统的实现原理

# 层次结构


用户级进程
用户层
系统调用接口
进程控制系统
进程通信
进程调度
内存管理
虚拟文件系统
Ext2 文件系统
NFS文件系统
其它文件系统
IO 子系统
网络协议
核心层
硬件控制模块
硬件层
物理硬件

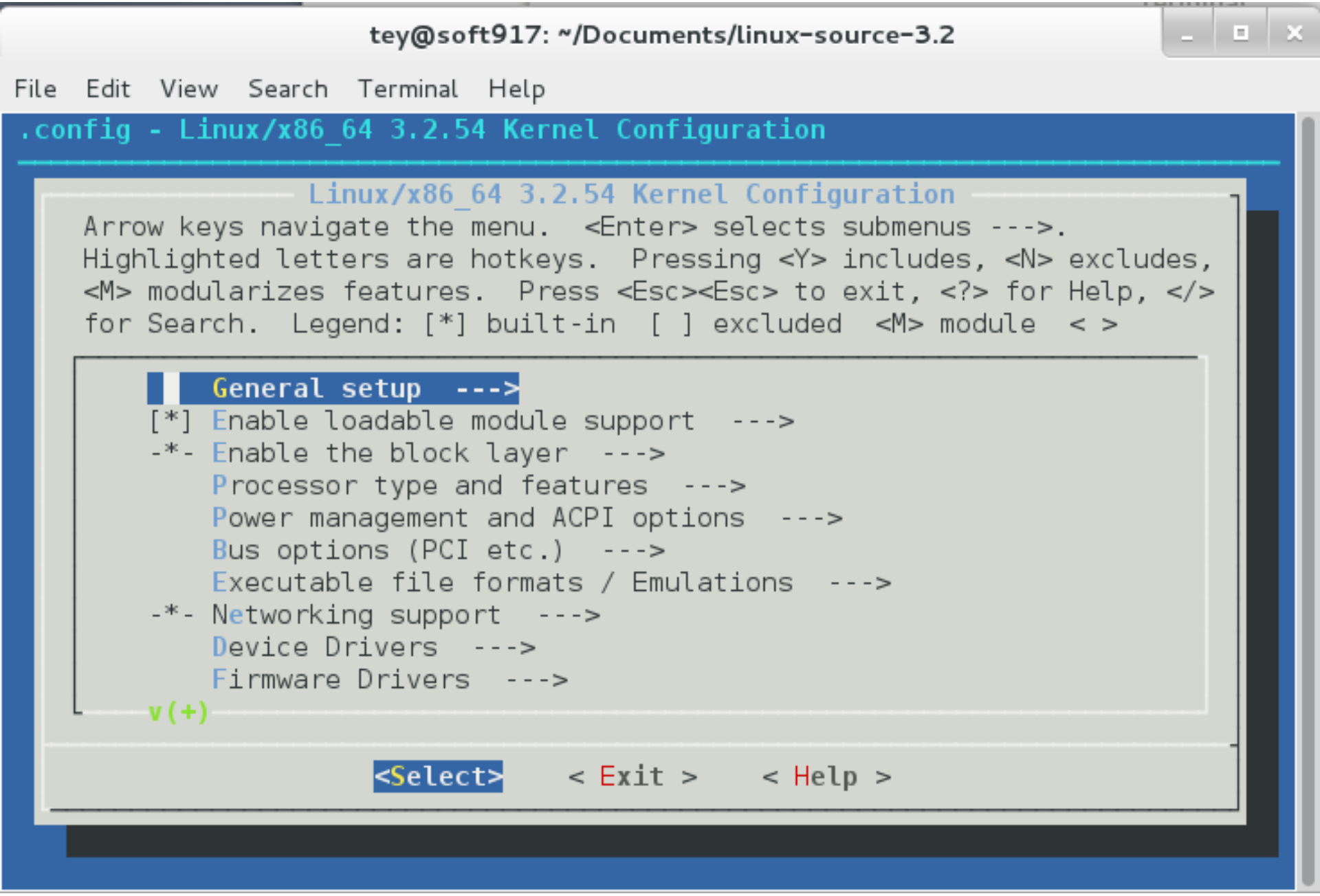
# 内核源代码获取

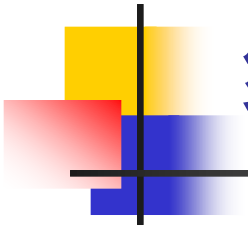
- https://www.kernel.org/
- apt-get方式
  - apt-cache search linux-source //查看内核版本
  - apt-get install linux-source-3.2
  - 下载下来的位置一般在/usr/src
- 从Ubuntu的源码库中获得内核源码
  - git clone git://kernel.ubuntu.com/ubuntu/ubuntu-hardy.git



# 后续操作

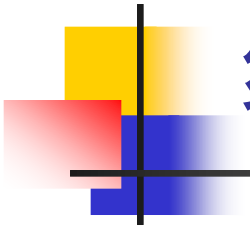
- 解压
  - tar jxvf /home/ldd/linux-3.2.tar.bz2
- 清除先前编译产生的目标文件
  - make clean
- 配置内核
  - make menuconfig


tey@soft917: ~/Documents/linux-source-3.2
File Edit View Search Terminal Help
.config - Linux/x86_64 3.2.54 Kernel Configuration
Linux/x86_64 3.2.54 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >
General setup --->
[*] Enable loadable module support --->
-*- Enable the block layer --->
Processor type and features --->
Power management and ACPI options --->
Bus options (PCI etc.) --->
Executable file formats / Emulations --->
-*- Networking support --->
Device Drivers --->
Firmware Drivers --->
v(+)
<Select> < Exit > < Help >



# 编译选项

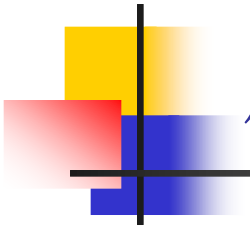
- 内核组件
  - Y(*) 要集成该组件
  - N( ) 不需要该组件，以后会没有这项功能
  - M 以后再加该组件为一个外部模块



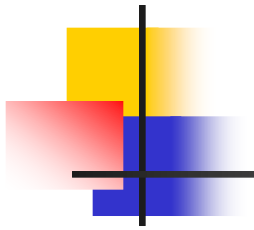
# 编译内核

- make
- make zImage
- make bzImage
- make modules

# 启用新内核



- make install□(慎用)
  - 将编译好的内核copy到/boot
- 配置引导菜单



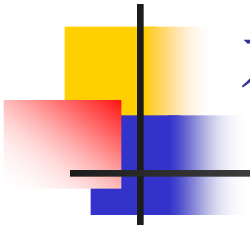
```
## ## End Default Options ##

title          Ubuntu 8.04.2, kernel 2.6.24-23-generic
root           (hd0,0)
kernel              /boot/vmlinuz-2.6.24-23-generic root=U
initrd              /boot/initrd.img-2.6.24-23-generic
quiet
```

```
title Win7
chainloader (hd0,0)/bootmgr
```

```
title win7
find --set-root  /bootmgr
chainloader  /bootmgr

title XP
find --set-root   /ntldr
chainloader  /ntldr
```

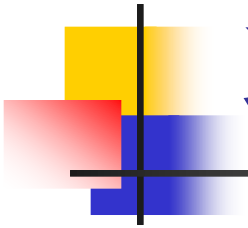


# 初始化程序的建立

- initrd
  - mkinitrd /boot/initrd.img $(uname -r)
- initramfs
  - mkinitramfs -o /boot/initrd.img 2.6.24-16
  - update-initramfs -u

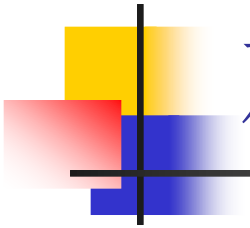
# Debian和Ubuntu的简便办法

- make-kpkg
  - 用于make menuconfig之后
- 好处
  - 后面所有的部分自动做完
  - 会把编译好的内核打成deb安装包
  - 可以拷到其它机器安装



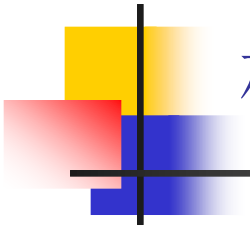
# 驱动

- 许多常见驱动的源代码集成在内核源码里
- 也有第三方开发的驱动，可以单独编译成模块.ko
- 编译需要内核头文件的支持



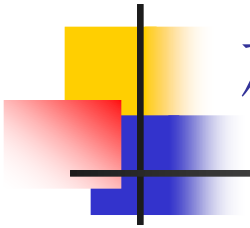
# 加载模块

- 底层命令
  - insmod
  - rmmod
- 高层命令
  - modprobe
  - modprobe -r



# 模块依赖

- 一个模块A引用另一个模块B所导出的符号，我们就说模块B被模块A引用。
- 如果要装载模块A，必须先要装载模块B。否则，模块B所导出的那些符号的引用就不可能被链接到模块A中。这种模块间的相互关系就叫做模块依赖。



# 模块的依赖

- 自动按需加载
- 自动按需卸载

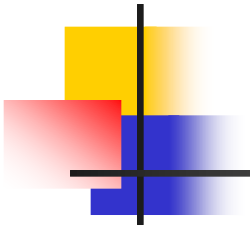
- moddep
- lsmod
- modinfo

# 模块之间的通讯

- 模块是为了完成某种特定任务而设计的。其功能比较的单一，为了丰富系统的功能，所以模块之间常常进行通信。其之间可以共享变量，数据结构，也可以调用对方提供的功能函数。

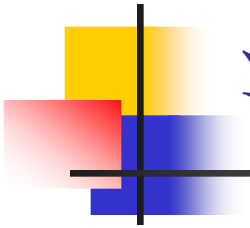
# 模块相关命令

- insmod <module.ko> [module parameters]
  - Load the module
  - 注意，只有超级用户才能使用这个命令
- rmmod
  - Unload the module
- lsmod
  - List all modules loaded into the kernel
  - 这个命令和cat  /proc/modules等价
- modprobe [-r] <module name>
  - – Load the module specified and modules it depends



# Linux内核模块与应用程序的区别

| | C语言程序 | Linux内核模块 |
|---|---|---|
| 运行 | 用户空间 | 内核空间 |
| 入口 | main() | module_init()指定; |
| 出口 | 无 | module_exit()指定; |
| 运行 | 直接运行 | insmod |
| 调试 | gdb | kdbug, kdb, kgdb等 |



# 注意点

- 不能使用C库来开发驱动程序
- 没有内存保护机制
- 小内核栈
- 并发上的考虑

# 最简单的内核模块例子

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
static int __init hello_init(void)
{
        printk(KERN_INFO "Hello world\n");
        return 0;
}
static void __exit hello_exit(void)
{
        printk(KERN_INFO "Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

- static int __init hello_init(void)
- static void __exit hello_exit(void)
  - Static声明，因为这种函数在特定文件之外没有其它意义
  - __init标记，该函数只在初始化期间使用。模块装载后，将该函数占用的内存空间释放
  - __exit标记 该代码仅用于模块卸载。
- Init/exit
  - 宏：module_init/module_exit
  - 声明模块初始化及清除函数所在的位置
  - 装载和卸载模块时，内核可以自动找到相应的函数

    module_init(hello_init);

    module_exit(hello_exit);

# 编译内核模块

- Makefile文件

```
obj-m := hello.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

- Module includes more files

```
obj-m:=hello.o
hello-objs := a.o b.o
```

# 和硬件打交道

```
//file name: ioremap_driver.c
#include<linux/module.h>
#include<linux/init.h>
#include<asm/io.h>
//用于存放虚拟地址和物理地址
volatile unsigned long virt,phys;
//用与存放三个寄存器的地址
volatile unsigned long*GPBCON,*GPBDAT,*GPBUP;
void led_device_init(void){
    //0x56000010+0x10包揽全所有的IO引脚寄存器地址
    phys=0x56000010;
    // 0x56000010=GPBCON
    //在虚拟地址空间中申请一块长度为0x10的连续空间
    //这样，物理地址phys到phys+0x10对应虚拟地址
virt到virt+0x10
    virt=(unsigned long)ioremap(phys,0x10);
```

```
void led_device_init(void)
{
    // 0x56000010 + 0x10 包揽全所有的IO引脚寄存器地址
    phys=0x56000010 ;
    // 0x56000010=GPBCON
    //在虚拟地址空间中申请一块长度为0x10的连续空间
    //这样，物理地址phys到phys+0x10对应虚拟地址virt到virt+0x10
    virt=(unsigned long)ioremap(phys,0x10);
    GPBCON=(unsigned long*)(virt+0x00);
    //指定需要操作的三个寄存器的地址
    GPBDAT=(unsigned long*)(virt+0x04);
    GPBUP=(unsigned long*)(virt+0x08);
}
//led配置函数,配置开发板的GPIO的寄存器
void led_configure(void)
{
```

```
}
//led配置函数,配置开发板的GPIO的寄存器
void led_configure(void)
{

*GPBCON&=~(3<<10)&~(3<<12)&~(3<<16)&~(3<<20);
    //GPB12 defaule 清零
    *GPBCON|=(1<<10)|(1<<12)|(1<<16)|(1<<20);
    //output  输出模式
    *GPBUP|=(1<<5)|(1<<6)|(1<<8)|(1<<10);
    //禁止上拉电阻
}
//点亮led
void led_on(void)
{

*GPBDAT&=~(1<<5)&~(1<<6)&~(1<<8)&~(1<<10);
}
//灭掉led
void led_off(void)
```

```
*GPBDAT&=~(1<<5)&~(1<<6)&~(1<<8)&~(1<<10);
}
//灭掉led
void led_off(void)
{
    *GPBDAT|=(1<<5)|(1<<6)|(1<<8)|(1<<10);
}
//模块初始化函数
static int __init led_init(void)
{
    led_device_init();
    //实现IO内存的映射
    led_configure();
    //配置GPB5 6 8 10为输出
    led_on();
    printk("hello ON!\n");
    return 0 ;
}
//模块卸载函数
static void __exit led_exit(void)
```

```
    led_on();
    printk("hello ON!\n");
    return 0 ;
}
//模块卸载函数
static void __exit led_exit(void)
{
    led_off();
    iounmap((void*)virt);
    //撤销映射关系
    printk("led OFF!\n");
}
module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("hurryliu<>");
MODULE_VERSION("2012-8-5.1.0");
```

# 模块参数传递

- 有些模块需要传递一些参数
- 参数在模块加载时传递

  #insmod hello.ko test=2

- 参数需要使用module_param宏来声明

  module_param(变量名称，类型，访问许可掩码)

- 支持的参数类型

  Byte, short, ushort, int, uint, long, ulong, bool, charp

  Array (module_param_array(name, type, nump, perm))

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

static int test;
module_param(test, int, 0644);

static int __init hello_init(void)
{
    printk(KERN_INFO “Hello world  test=%d \n” , test);
    return 0;
}
static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye world\n");
}
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("xxx");
module_init(hello_init);
module_exit(hello_exit);
```

# 模块之间的通讯实例

- 本实例通过两个模块来介绍模块之间的通信。模块add_sub提供了两个导出函数add_integer()和sub_integer()，分别完成两个数字的加法和减法。模块test用来调用模块add_sub提供的两个方法，完成加法或者减法操作。
- 1．add_sub模块
- 2．test模块
- 3．编译模块

```
#ifndef _ADD_SUB_H_
#define _ADD_SUB_H_
long add_integer(long a, long b);
long sub_integer(long a, long b);
#endif
```

```
01  #include <linux/init.h>
02  #include <linux/module.h>
03  #include "add_sub.h"
04  long add_integer(int a, int b)
05  {
06          return a+b;
07  }
08  long sub_integer(int a, int b)
09  {
10          return a-b;
11  }
12  EXPORT_SYMBOL(add_integer);
13  EXPORT_SYMBOL(sub_integer);
14  MODULE_LICENSE("Dual BSD/GPL");
```

```
#include <linux/init.h>
#include <linux/module.h>
#include "add_sub.h"                    /* 不要使用<>包含文件，否则找不到
/* 定义模块传递的参数 a，b */
static long a = 1;
static long b = 1;
static int AddOrSub =1;
static int test_init(void)              /* 模块加载函数 */
{
        long result=0;
        printk(KERN_ALERT "test init\n");
        if(1==AddOrSub)
        {
            result=add_integer(a, b);
        }
        else
        {
            result=sub_integer(a, b);
        }
        printk(KERN_ALERT "The %s result is %ld",AddOrSub==1?"
        "Sub",result);
        return 0;
}
```

```
obj-m := test.o
KERNELDIR ?= /linux-2.6.29.4/linux-2.6.29.4
PWD := $(shell pwd)
SYMBOL_INC = $(obj)/../include
EXTRA_CFLAGS += -I $(SYMBOL_INC)
KBUILD_EXTRA_SYMBOLS=$(obj)/../print/Module.symvers
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
modules_install:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versi
.PHONY: modules modules_install clean
```

# 导出符号表

- 如果一个模块需要向其他模块导出符号（方法或全局变量），需要使用：

  EXPORT_SYMBOL(name);

  EXPORT_SYMBOL_GPL(name);

  *注意：符号必须在模块文件的全局部分导出，不能在函数部分导出。更多信息可参考 <linux/module.h>文件

- Modules仅可以使用由Kernel或者其他Modules导出的符号不能使用Libc

- /proc/kallsyms 可以显示所有导出的符号

```
[root@localhost sample]# cat /proc/kallsyms
```

# 内核模块操作/proc文件

- /proc文件系统，这是内核模块和系统交互的两种主要方式之一。
- /proc文件系统也是Linux操作系统的特色之一。
- /proc文件系统不是普通意义上的文件系统，它是一个伪文件系统。
- 通过/proc，可以用标准Unix系统调用(比如open()、read()、write()、 ioctl()等等)访问进程地址空间
- 可以用cat、more等命令查看/proc文件中的信息。
- 用户和应用程序可以通过/proc得到系统的信息，并可以改变内核的某些参数。
- 当调试程序或者试图获取指定进程状态的时候，/proc文件系统将是你强有力的支持者。通过它可以创建更强大的工具，获取更多信息。

# /proc相关函数

- create_proc_entry() 创建一个文件
- proc_symlink() 创建符号链接
- proc_mknod() 创建设备文件
- proc_mkdir() 创建目录
- remove_proc_entry() 删除文件或目录

# 驱动类型

- Linux系统将设备分为3种类型：字符设备、块设备和网络接口设备。
- 1．字符设备 Character Driver
- 2．块设备 Block Driver
- 3．网络接口设备 Network Driver

# 简单的字符设备驱动程序

- 在Linux设备驱动程序的家族中，字符设备驱动程序是较为简单的驱动程序，同时也是应用非常广泛的驱动程序。所以学习字符设备驱动程序，对构建Linux设备驱动程序的知识结构非常的重要。

# 文件操作

## ——字符设备驱动的对上接口

- ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
- ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
- int (*flush) (struct file *);
- int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
- ……

# 文件操作(file_operations)

## ——字符设备驱动的对上接口

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

# 两个基本结构

- file 结构体
- inode 结构体
  - dev_t i_rdev;
  - struct cdev *i_cdev;

# 字符设备驱动程序的初始化加载过程

- 申请设备号
- 定义文件操作结构体 file_operations
- 创建并初始化定义结构体 cdev
- 将cdev注册到系统，并和对应的设备号绑定
- 在/dev文件系统中用mknod创建设备文件，并将该文件绑定到设备号上

# 主设备号和次设备号

- 一个字符设备或者块设备都有一个主设备号和次设备号。
- 主设备号和次设备号统称为设备号。主设备号用来表示一个特定的驱动程序。
- 次设备号用来表示使用该驱动程序的各设备。

# 申请和释放设备号

- int register_chrdev_region(dev_t first, unsigned int count, char *name);
- int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
- void unregister_chrdev_region(dev_t first, unsigned int count);

# cdev结构体

- 在linux内核中使用cdev结构体来描述字符设备。该结构体是所有字符设备的抽象，其包含了大量字符设备所共有的特性。

# cdev结构体的初始化

- struct cdev *my_cdev = cdev_alloc();
- my_cdev->ops = &my_fops;

- void cdev_init(struct cdev *cdev, struct file_operations *fops);

# 自定义结构

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;     /* Char device structure      */
};
```

# 设备注册

- 将设备注册到系统中：
- int cdev_add(struct cdev *dev, dev_t num, unsigned int count);

- 释放一个已经注册的设备：
- void cdev_del(struct cdev *dev);

# 实现file_operations中的各个函数

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        scull_trim(dev); /* ignore errors */
    }
    return 0;          /* success */
}
```

# 创建设备文件，并绑定到设备号

- 定义设备名device=scull
- 定义主设备号major=15
- 用户可以通过访问/dev/scull0来访问当前的驱动程序

```
mknod /dev/${device}0 c $major 0
```

End