韦东山一期考试题 理论

1.  谈谈你对 u-boot, bootloader 的理解? 作用是什么？ u-boot 参数的作用是什么？你使用过那些？

1）Bootloader的功能是初始化watchdog，系统时钟，存储控制器等硬件，同时将代码复制到内存里。它在系统上电时开始执行，不同的硬件会有不同的bootloader。Uboot是universal bootloader(通用bootloader), 它可以支持多种嵌入式操作系统内核和处理器，有开放源码，支持NFS挂载和上电自检，特殊的可以作为XIP内核引导。

2）Uboot的作用是从flash读出内核并且启动内核（boot kernel）。Uboot分为两个阶段，第一阶段的任务是：

* 设为管理模式
* 硬件相关初始化（关watchdog，屏蔽中断，sdram init，设置栈sp，时 钟）
* 从flash读出内核or重定位（code从flash copy到sdram），清bss段
* Call start\_armboot

第二阶段的任务是：

* gd（全局数据）区清零
* init\_sequence初始化函数循环
* 初始化nor flash，nand flash，初始化标准输入输出设备（kerboard，LCD etc）
* 初始化环境变量，板级初始化（sd卡，网卡 etc）
* 主循环，读取bootcmd

3）uboot参数的作用是uboot传递参数给内核，使用do\_bootm\_linux()（/lib\_arm/bootm.c定义）跳到入口地址启动linux kernel。

具体原理为：  
U-BOOT 在启动内核时,会向内核传递一些参数. 每一个启动参数对应一个 struct tag。启动 kernel 时，把这个结构体的物理地址传给 kernel；Linux kernel 通过这个地址分析出u-boot传递的参数。相应的tag为：  
setup\_start\_tag (bd); 这里设置了参数存放的位置  
setup\_revision\_tag (&params);  
setup\_memory\_tags (bd);  
setup\_commandline\_tag (bd, commandline); // commandline是bootargs  
setup\_initrd\_tag (bd, images->rd\_start, images->rd\_end);  
setup\_videolfb\_tag ((gd\_t \*) gd);  
setup\_end\_tag (bd);设置的，   
U-boot中通过环境参数保存一些配置，这些配置可以通过修改环境参数、保存环境参数、读取环境参数等操作进行灵活的配置，便于调试开发。

4）使用过的环境变量：

* Saveenv
* Bootdelay ---执行自动启动的等候秒数
* Baudrate ---设置串口波特率
* Serverip ---服务器端的ip地址
* Ipaddr ---本地ip地址
* Printenv ----查看环境变量
* Bootargs ----传递给内核的启动参数
* Bootcmd ----自动启动时执行的命令

2.  谈谈你对 Linux Kernel 的理解? 作用是什么？

Kernel initializes its own before setting up a proper environment for the running of C code, then jumps to the architecture-independent start\_kernel( ) function, which initializes the high-level kernel functionality, mounts the root filesystem, and starts the init process.

Linux kernel不是一套完整的操作系统，它是用[C语言](https://zh.wikipedia.org/wiki/C%E8%AF%AD%E8%A8%80)和[汇编语言](https://zh.wikipedia.org/wiki/%E6%B1%87%E7%BC%96%E8%AF%AD%E8%A8%80)写成。Kernel是属于系统级别的，与user space相对应。有两种mode：kernel mode 和user mode。在driver程序里，User mode不可以直接访问kernel mode，必须通过函数copy\_to\_user或copy\_from\_user, 对process进行block和wakeup的操作只能在kernel里进行。作为执行单位的thread分为kernel thread和user thread。在RT system里，如powerpc，kernel thread又被称为task，是可以参与kernel schedule的object。一般讲thread是user thread。

Linux 2.6是标准kernel，也是multi-thread kernel。标准kernel属于os的一部分，driver code调用kernel都在kernel space，driver死，整个os死掉。

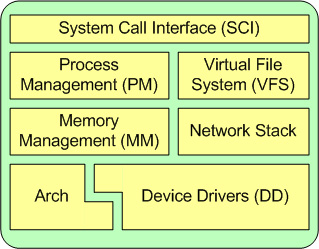
QNX (Rt system)使用的是微内核技术，driver通过ITC（任务间通信）和macro kernel联系，所以只是某一个driver dead，不影响kernel和其他driver。



标准Linux操作系统分为user space和kernel space，如上图。

Linux 内核可以进一步划分成 3 层。最上面是系统调用接口，它实现了一些基本的功能，例如 read 和 write。系统调用接口之下是内核代码，这些代码之下是依赖于体系结构的代码，构成了通常称为 BSP（Board Support Package）

Linux 内核的主要组件如下：



系统调用接口

SCI 层提供了某些机制执行从用户空间到内核的函数调用。这个接口依赖于体系结构，

进程管理

进程管理的重点是资源，包括内存和设备。进程管理还包括处理活动进程之间共享 CPU 的需求。内核实现了一种新型的调度算法，不管有多少个线程在竞争 CPU，这种算法都可以在固定时间内进行操作。这种算法就称为 O(1) 调度程序，这个名字就表示它调度多个线程所使用的时间和调度一个线程所使用的时间是相同的。 O(1) 调度程序也可以支持多处理器（称为对称多处理器或 SMP）。

内存管理

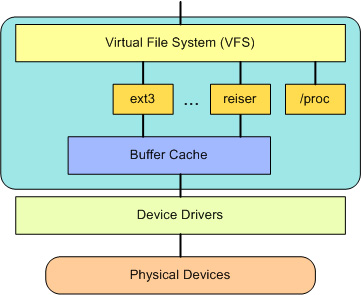
如果由硬件管理虚拟内存，内存是按照所谓的内存页 方式进行管理的（对于大部分体系结构来说都是 4KB）。Linux 包括了管理可用内存的方式，以及物理和虚拟映射所使用的硬件机制。

不过内存管理要管理的可不止 4KB 缓冲区。Linux 提供了对 4KB 缓冲区的抽象，例如 slab 分配器。这种内存管理模式使用 4KB 缓冲区为基数，然后从中分配结构，并跟踪内存页使用情况，比如哪些内存页是满的，哪些页面没有完全使用，哪些页面为空。这样就允许该模式根据系统需要来动态调整内存使用。

为了支持多个用户使用内存，有时会出现可用内存被消耗光的情况。由于这个原因，页面可以移出内存并放入磁盘中。这个过程称为交换，因为页面会被从内存交换到硬盘上。

虚拟文件系统

虚拟文件系统（VFS）为文件系统提供了一个通用的接口抽象。VFS 在 SCI 和内核所支持的文件系统之间提供了一个交换层（请参看下图 ）。



在 VFS 上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。它们是给定文件系统（超过 50 个）的插件。

文件系统层之下是缓冲区缓存，它为文件系统层提供了一个通用函数集（与具体文件系统无关）。这个缓存层通过将数据保留一段时间（或者随即预先读取数据以便在需要是就可用）优化了对物理设备的访问。缓冲区缓存之下是设备驱动程序，它实现了特定物理设备的接口。

网络堆栈

网络堆栈在设计上遵循模拟协议本身的分层体系结构。Internet Protocol (IP) 是传输协议（TCP）下面的核心网络层协议。TCP 上面是 socket 层，它是通过 SCI 进行调用的。

socket 层是网络子系统的标准 API，它为各种网络协议提供了一个用户接口。从原始帧访问到 IP 协议数据单元（PDU），再到 TCP 和 User Datagram Protocol (UDP)，socket 层提供了一种标准化的方法来管理连接，并在各个终点之间移动数据。

设备驱动程序

Linux 内核中的设备驱动程序能够运转特定的硬件设备。Linux 源码树提供了一个驱动程序子目录，这个目录又进一步划分为各种支持设备，例如 Bluetooth、I2C、serial 等。

依赖体系结构的代码

./linux/arch 子目录定义了内核源代码中依赖于体系结构的部分，其中包含了各种特定于体系结构的子目录（共同组成了 BSP）。

Kernel的作用：

kernel是整个操作系统的最底层，它负责整个硬件的驱动，以及提供各种系统所需的核心功能，包括防火墙机制、是否支持LVM或Quota等文件系统等等，如果内核不认识某个最新的硬件，那么硬件也就无法被驱动，你也就无法使用该硬件。内核就是控制这些芯片如何工作。

3.  谈谈你对 File System 的理解? 作用是什么？

文件系统是包括在一个磁盘(包括光盘、软盘、闪盘及其它存储设备)或分区的目录结构；一个可应用的磁盘设备可以包含一个或多个文件系统；如果进入一个文件系统，首先挂载(mount)文件系统；为了挂载(mount)文件系统，必须指定一个挂载点。

**主要嵌入式采用的文件系统:**

Linux 中，rootfs 是必不可少的。嵌入式中一般不从 HD 启动，而是从 Flash 启动，最简单的方法是将 rootfs load 到 RAM 的 RAMDisk，稍复杂的就是 直接从Flash 读取的 Cramfs，更复杂的是在 Flash 上分区，并构建 JFFS2 等文件系统。

RAMDisk 将制作好的 rootfs 压缩后写入 Flash，启动的时候由 Bootloader load 到RAM，解压缩，然后挂载到 /。这种方法操作简单，但是在 RAM 中的文件系统不是压缩的，因此需要占用许多嵌入式系统中稀有资源 RAM。

ramdisk 就是用内存空间来模拟出硬盘分区,ramdisk通常使用磁盘文件系统的压缩存放在flash中,在系统初始化时,解压缩到SDRAM并挂载根文件系统,

initrd 是 RAMDisk 的格式，kernel 2.4 之前都是 image-initrd，Kernel 2.5 引入了 cpio-initrd.

Cramfs 是很简单的文件系统，可以直接从 Flash 上运行，不须 load 到 RAM 中，因此节约了 RAM。但是 Cramfs 是只读的，对于需要运行时修改的目录(如： /etc, /var, /tmp)多有不便，因此，一般将这些目录做成ramfs 等可写的 fs。

**建一个包含所有文件的目录:**

1。建一个目录rootfs 用来装文件系统

2。mkdir bin dev etc lib proc sbin tmp usr var

3. ln -fs bin/busybox linuxrc(使用busybox)

4. 到系统 /dev 把所有的device打一个包，拷贝到 dev下面(最省事的做法)；或者使用mknod来自己建所需要的device,

举例： mknod console c 5 1

5。将编译好的busybox拷贝到/bin下面，除了busybox外，所有其他的命令都是他的link. 所有的命令可以在busybox下面用make menuconfig来增减.

6。同样/sbin下面也是busybox的link

7。同样/usr/bin下面也是busybox的link

8。同样/usr/sbin下面放着所有编译完的可执行文件

9。/lib，找到你编译环境的target目录，把需要的lib文件先用strip压縮(非target目录下的，而已编译环境提供的strip)，先把最基本的libc, ld等等，必须同样做跟target/lib里面一样的link。然后根据特定的应用加相应的lib,不要把不用的加进去，lib比较占空间。

**生成一个cramfs:**

找到cramfs的toolchain。/mkcramfs -r $(FS1\_DIR) $(FS\_NAME).

**嵌入式系统文件系统比较 : jffs2, yaffs, cramfs, romfs, ramdisk, ramfs/tmpfs**

Linux启动时，第一个必须挂载的是根文件系统；若系统不能从指定设备上挂载根文件系统，则系统会出错而退出启动。之后可以自动或手动挂载其他的文件系统。因此，一个系统中可以同时存在不同的文件系统。

　 在嵌入式Linux应用中，主要的存储设备为 RAM(DRAM, SDRAM)和ROM(常采用FLASH存储器)，常用的基于存储设备的文件系统类型包括：jffs2, yaffs, cramfs, romfs, ramdisk, ramfs/tmpfs等。

　　基于FLASH的文件系统:

　　Flash(闪存)作为嵌入式系统的主要存储媒介，有其自身的特性。Flash的写入操作只能把对应位置的1修改为0，而不能把0修改为1(擦除Flash就是把对应存储块的内容恢复为1)，因此，向Flash写入内容时，需要先擦除对应的存储区间，这种擦除是以块(block)为 单位进行的。

　　闪存主要有NOR和NAND两种技术。Flash存储器的擦写次数是有限的，NAND闪存还有特殊的硬件接口和读写时序。因此，必须针对Flash的硬件特性设计符合应用要求的文件系统；传统的文件系统如ext2等，用作Flash的文件系统会有诸多弊端。

　　在嵌入式Linux下，MTD(Memory Technology Device,存储技术设备)为底层硬件(闪存)和上层(文件系统)之间提供一个统一的抽象接口，即Flash的文件系统都是基于MTD驱动层的(参见上面的Linux下的文件系统结构图)。使用MTD驱动程序的主要优点在于，它是专门针对各种非易失性存储器(以闪存为主)而设计的，因而它对Flash有 更好的支持、管理和基于扇区的擦除、读/写操作接口。

　　一块Flash芯片可以被划分为多个分区，各分区可以采用不同的文件系统；两块Flash芯片也可以合并为一个分区使用，采用一个文件系统。即文件系统是针对于存储器分区而言的，而非存储芯片。

jffs2:

Jffs2用于NOR型闪存，基于MTD驱动层，特点是：可读写的、支持数据压缩的、基于哈希表的日志型文件系统，并提供了崩溃/掉电安全保护，提供“写平衡”支持等。缺点主要是当文件系统已满或接近满时，因为垃圾收集的关系而使jffs2的运行速度大大放慢。

**yaffs：Yet Another Flash File System**

yaffs/yaffs2与jffs2相比，它减少了一些功能(例如不支持数据压缩)，所以速度更快，挂载时间很短，对内存的占用较小。另外，它还是跨平台的文件系统，除了Linux和eCos，还支持WinCE, pSOS和ThreadX等。

　yaffs/yaffs2自带NAND芯片的驱动，并且为嵌入式系统提供了直接访问文件系统的API，用户可以不使用Linux中的MTD与VFS，直接对文件系统操作。yaffs也可与MTD驱动程序配合使用。

yaffs与yaffs2的主要区别在于，前者仅支持小页(512 Bytes) NAND闪存，后者则可支持大页(2KB) NAND闪存。同时，yaffs2在内存空间占用、垃圾速度、读/写速度等方面均有大幅提升。

**Cramfs：Compressed ROM File System**

　Cramfs是一种只读的压缩文件系统。它也基于MTD驱动程序。

　在cramfs文件系统中，每一页(4KB)被单独压缩，可以随机页访问，其压缩比高达2:1,为嵌入式系统节省大量的Flash存储空间，使系统可通过更低容量的FLASH存储相同的文件，从而降低系统成本。

　Cramfs文件系统以压缩方式存储，在运行时解压缩，所以不支持应用程序以XIP(eXecute In Place，芯片内执行)方式运行，所有的应用程序要求被拷到RAM里去运行，但这并不代表比Ramfs需求的RAM空间要大一点，因为Cramfs是采用分页压缩的方式存放档案，在读取档案时，不会一下子就耗用过多的内存空间，只针对目前实际读取的部分分配内存，尚没有读取的部分不分配内存空间，当我们读取的档案不在内存时，Cramfs文件系统自动计算压缩后的资料所存的位置，再即时解压缩到RAM中。

　另外，它的速度快，效率高，其只读的特点有利于保护文件系统免受破坏，提高了系统的可靠性。Cramfs映像通常是放在Flash中。

**Romfs**

　　传统型的Romfs文件系统是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存，按顺序存放数据，因而支持应用程序以 XIP(eXecute In Place，片内运行)方式运行，在系统运行时，节省RAM空间。uClinux系统通常采用Romfs文件系统。

　其他文件系 统：fat/fat32也可用于实际嵌入式系统的扩展存储器(例如PDA, Smartphone, 数码相机等的SD卡)，这主要是为了更好的与最流行的Windows桌面操作系统相兼容。ext2也可以作为嵌入式Linux的文件系统，不过将它用于 FLASH闪存会有诸多弊端。

**基于RAM的文件系统**

**Ramdisk**

　Ramdisk是将一部分固定大小的内存当作分区来使用。它并非一个实际的文件系统，而是一种将实际的文件系统装入内存的机制，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件(如只读的根文件系统)通过Ramdisk放在内存中，可以明显地提高系统的性能。

　在Linux的启动阶段，initrd提供了一套机制，可以将内核映像和根文件系统一起载入内存。

**ramfs/tmpfs**

　Ramfs是一种基于内存的文件系统，工作于虚拟文件系统(VFS)层，不能格式化，可以创建多个，在创建时可以指定其最大能使用的内存大 小。(实际上，VFS本质上可看成一种内存文件系统，它统一了文件在内核中的表示方式，并对磁盘文件系统进行缓冲。)

　　Ramfs/tmpfs文件系统把所有的文件都放在RAM中，所以读/写操作发生在RAM中，可以用ramfs/tmpfs来存储一些临时性或经常要修改的数据，例如/tmp和/var目录，这样既避免了对Flash存储器的读写损耗，也提高了数据读写速度。

　　Ramfs/tmpfs相对于传统的Ramdisk的不同之处主要在于：不能格式化，文件系统大小可随所含文件内容大小变化。

　　Tmpfs的一个缺点是当系统重新引导时会丢失所有数据。

**网络文件系统NFS (Network File System)**

　　NFS在嵌入式Linux系统的开发调试阶段，可以利用该技术在主机上建立基于NFS的根文件系统，挂载到嵌入式设备，可以很方便地修改根文件系统的内容。

　　附录：NOR闪存与NAND闪存比较

|  |  |
| --- | --- |
| NOR FLASH | 接口时序同SRAM,易使用读取速度较快擦除速度慢，以64-128KB的块为单位写入速度慢(因为一般要先擦除)随机存取速度较快，支持XIP(eXecute In Place，芯片内执行)，适用于代码存储。在嵌入式系统中，常用于存放引导程序、根文件系统等。单片容量较小，1－32MB最大擦写次数10万次 |
| NAND FLASH | 读取速度较慢地址/数据线复用，数据位较窄擦除速度快，以8－32KB的块为单位写入速度快顺序读取速度较快，随机存取速度慢，适用于数据存储(如大容量的多媒体应用)。在嵌入式系统中，常用于存放用户文件系统等单片容量较大，8－128MB，提高了单元密度最大擦写次数100万次 |

**4. 谈谈你对 Linux Device Driver 的理解?**

设备驱动的基本框架如下：

Struct file operations drv\_fops = {

.read = led\_read;

.owner =This\_Moudle;

.write = led\_write;

}

static int led\_open(struct inode \*inode, struct file \*file)

{设置硬件信息：寄存器清零并确定是输入或是输出

}

ssize\_t led\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offp)

{从硬件寄存器读出数值并用copy\_to\_user(buf,key\_val,sizeof(key\_val))传到用户app

}

ssize\_t led\_write(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offp)

{用copy\_from\_user(&val, buf, count)把数据从user app读入，并把此值写入相应的寄存器;

}

static int \_\_init led\_init(void)

{ register\_chrdev(118, "led\_zxb", &led\_fops)；//注册设备

用ioremap把寄存器的物理地址PA转为虚拟地址VA

}

static void \_\_exit led\_exit(void)

{unregister\_chrdev(118, "led\_zxb");//卸载register\_chrdev

iounmap(gpbcon);//对应ioremapap

}

//用module修饰一下led\_init，led\_exit使之成为入口和出口函数

module\_init(led\_init);

module\_exit(led\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("zxb");

**设备驱动学习体会集总：**

* 设备驱动有polling mode和interrupt mode。select or poll：等多个设备。

针对输入read：是否等只看buffer里是否有数据，有：copy\_to\_user，无，block，wakeup条件为buffer非空；没有数据，return分两种：1.硬件结束return Eoof 2.硬件没结束 block read buffer，check是否有数据，

* 针对输出write：最好也有个写buffer，直接copy到buffer，前提是数据量比较大，一个字节直接write
* block等在这里，条件是buffer不满，在中断里wake up

int (\*fsync) (struct file \*file, struct dentry \*dentry, int datasync);

有buffer，一定写fsync，因为有buffer，write有问题，不可保证什末时候写到硬件，保证写到硬件：用户用fsync-user Api， sleep，知道硬件全部写完，buffer全部清空

* block不占cpu时间，如果不想等，干别的事，注册一个信号handler， 等数据来，信号通知， int fasync\_helper(int fd, struct file \*filp, int mode, struct fasync\_struct \*\*fa); 有个结构帮助初始化和填充，call SETFL时call此函数，0是清掉，1为设上信号，此函数修改指针里的相关参数。
* void kill\_fasync(struct fasync\_struct \*\*fa, int sig, int band);

数据ready时，多为输入ready，此时发个信号，kill\_fasync帮发此信号，但如果状态为0--表示clear，就不发，1，就发

* **Workqueue**

如果按键次数不是那么频繁，即使很频繁，用poll引起大量的context switch，效率低下，解决：让系统去poll，彻底sleep，直到等到数据，一次context switch---task queue -------2.6叫work queue

用法：

1。不属于任何一个进程，属于系统，当前运行的context是系统的context-----中断context，返回值为1（中断context）

2.copy\_to\_user copy\_from\_user是进程，需要内存空间，只有进程才可以跨kernel和user空间，

3.不可访问用户空间，不可访问kernel指针（指向当前的进程），因为不是进 程，

4.不可sleep，block不可schedule---换kernel，ex.jiq.c

唯一不属于系统进程，属于queue event，等系统闲（idle)时调，注册只运行一次，这次注册，下次闲时运行。

* block api：schedule，sleep\_on，wait\_event，waitup和printk可以
* tasklet：替代tq\_immediately用在中断的bottom half

1.固定cpu，第一次在此cpu run，forever

2.tasklet---小task，一直在cache，不用重复往cache上加载，快

* kernel timer：定多长时间后运行，当前jiffis+自己写的
* Barrier：对内存，比如ram：读操作，值不变，可把内存临时放到cache，cpu里，优化-内存搬到寄存器，enable cache；对外设：外设指针放到寄存器，因为外设会变，要用volatile
* semaphore里面都有memory barrier，底层操作，应用app不用考虑
* 指令乱序

硬件执行是不按照你写的顺序，因为pipeline造成指令乱序---对内存无关（地址不相关）。对外设，地址不相关，但硬件状态相关，错误，加硬件memory barrier

不要用mb，因为这些宏真正的实现是和硬件平台相关的，ex。powerpc平台先写后读有问题---wmb，但是先读后写没问题，所以rmb什末都不干。

对外设读写，寄存器内存操作用readb，writb，不要用指针直接来，不用考虑字节顺序。用指针，不可知，因为指针系统可能会优化，即使用上volatile，有指针字节顺序问题，因为不同cpu的字节顺序可能是不同，little endian，big endian，不用考虑字节顺序

* 中断三种：

输入数据来了，出错，写完成，输出完成

中断：典型driver

写比较简单，先写到一个内存buffer，再给个写命令，数据放到写寄存器，硬 件工作，发完后，硬件就产生一个中断给cpu，我们要做的：

1。driver写一个中断的handler，中断call 这个handler

2. 初始化driver里，把中断handler注册到相关的中断号处

中断和read和write一般为异步，如果是share buffer，为防止**race condition**， read write里关中断，处理完后开中断

* 写中断handler：

1.写一个task

2.不许block，schedule

3.不许访问用户空间，因为是中断context

4.不许访问current

linux机制：

中断来了后，首先屏蔽所有中断，在确认你的中断号，看中断号对应的handler是否为fast，不是，离开此状态，进入mask自己状态；如果是slow，一旦做完，return，schedule，执行tasklet

* 中断driven的read：

注册handler放在哪里比较好：只有自己用，可放在init里注册

shared 中断，放在open里

* 中断driven的write：write buffer，写操作，数据写入buffer，设硬件命令，让此发，不用等再次，return，如果buffer full，block。

硬件发，做个list，产生中断（buffer有一部分空出来了），不闲着，继续发，wakeup那些满的写的。

* 防止race condition：

using a circular buffer and avoiding shared variables

 Using spinlocks to enforce mutual exclusion

 Using lock variables that are atomically incremented and cremented

* 中断处理程序希望尽快结束，但要结束任务，因此中断处理程序分为2部分：top half声明已经服务了中断，并将hard work交给bottom half处理。底半部在所有中断都enabled后的某个时间点执行。
* 中断处理可分为两步：顶半部和底半部。底半部是同步的，由内核决定何时执行。相应的底半部技术有：softirq、tasklets和工作队列。软中断有strong locking requirement,常用于performance-sensitive子系统，如：网络层，SCSI层，和内核时钟。Tasklet建立在软中断的基础上，推荐使用tasklet除非您在关键性领域或速度要求。softirq和tasklet的主要区别在于：softirq是可重入的，而tasklet不是。一个softirq的不同实例可以并行运行在不同CPU上，而tasklet不行。 内核调度softirq执行，或者在离开中断上下文时，或者由ksoftirqd负责处理。 tasklet\_init()动态初始化一个tasklet，而tasklet\_schedule()声明相应的tasklet已经挂到可执行里面了。tasklet\_enable()使能tasklets，tasklet\_disable()禁止tasklet直到某执行中的tasklet实例退出，tasklet\_disable\_nosync()不会等待处于active的tasklet实例停止执行。
* 中断处理程序和tasklets都不可重入，也不能睡眠；中断处理程序、tasklets和softirqs不能被抢占。工作队列是另外一种延迟执行任务的方法，它们处于进程上下文而且允许睡