Linux 驱动开发

传智播客-虚竹[1]

2015-03-07

[1]http://www.itcast.cn

前言

学习目标

成为linux/unix系统程序员

学习态度

* 谦虚
* 严谨
* 勤思
* 善问

学习方法

只听不练肯定学不会Linux，每个知识点都需要去动手实践

目录

第 1 章

linux驱动简介

1.1 下载kernel

1.去kernel官网下载，www.kernel.org
2.通过git方式下载，注意，需要3G左右磁盘空间

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

1.2 设备分类

1.字符设备驱动
2.块设备驱动
3.网络设备驱动
文件类型分类:
1.普通文件
2.目录文件
3.符号链接文件
4.管道文件
5.网络文件
6.字符设备文件
7.块设备文件
8.未知文件

第 2 章

驱动核心结构体

2.1 kobject

```
/* include/linux/kobject.h */

struct kobject {
    const char                *name;
    struct list_head        entry;
    struct kobject                *parent;
    struct kset                *kset;
    struct kobj_type        *ktype;
    struct kernfs_node        *sd;
    struct kref                kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
    struct delayed_work        release;
#endif
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

2.2 ktype

```
/* include/linux/kobject.h */

struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
    const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
    const void *(*namespace)(struct kobject *kobj);
};
```

2.3 kset

```
/* include/linux/kobject.h */

struct kset {
    struct list_head list;
    spinlock_t list_lock;
    struct kobject kobj;
    const struct kset_uevent_ops *uevent_ops;
};
```

2.4 struct device

2.5 struct device_driver

2.6 struct bus_type

2.7 struct class

2.8 struct class_device

2.9 bus、device、driver、class属性

2.9.1 struct bus_attribute

2.9.2 struct dirver_attribute

2.9.3 struct class_attribute

2.9.4 struct class_device_attribute

2.10 cdev

第 3 章

udev机制

第 4 章

内核数据结构

4.1 链表

linux内核链表与众不同，他不是把将数据结构塞入链表，而是将链表节点塞入数据，在2.1内核中引入了官方链表，从此内核中所有的链表使用都采用此链表，千万不要在重复造车轮子了！链表实现定义在<linux/list.h>,使用内核链表时，包含此文件。

4.1.1 链表基础数据结构

首先需要理解把链表节点塞入数据，如何定位链表节点位置?

内核链表节点原型

```
/* linux/types.h */
struct list_head {
    struct list_head *next, *prev;
};
```

gcc特有的语法支持，根据结构体成员和结构体，算出此成员所在结构体内的偏移量

```
/* linux/kernel.h */

/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:    the pointer to the member.
 * @type:   the type of the container struct this is embedded in.
 * @member: the name of the member within the struct.
 */
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

container_of在Linux Kernel中的应用非常广泛,它用于获得某结构中某成员的入口地址.

关于offsetof见stddef.h中:

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

TYPE是某struct的类型 0是一个假想TYPE类型struct,MEMBER是该struct中的一个成员.由于该struct的基地址为0, MEMBER的地址就是该成员相对与struct头地址的偏移量.

关于typeof,这是gcc的C语言扩展保留字,用于声明变量类型.

const typeof(((type *)0->member) *mptr = (ptr);意思是声明一个与member同一个类型的指针常量 *mptr,并初始化为ptr.

(type *)((char *)mptr - offsetof(type,member));意思是mptr的地址减去member在该struct中的偏移量得到的地址, 再转换成type型指针. 该指针就是member的入口地址了.

链表中使用此宏例子：

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

4.1.2 链表操作主要函数

* 声明和初始化

实际上Linux只定义了链表节点，并没有专门定义链表头，那么一个链表结构是如何建立起来的呢？让我们来看看LIST_HEAD()这个宏：

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INIT(name)
```

当我们用LIST_HEAD(nf_sockopts)声明一个名为nf_sockopts的链表头时，它的next、prev指针都初始化为指向自己，这样，我们就有了一个空链表，因为Linux用头指针的next是否指向自己来判断链表是否为空：

```
static inline int list_empty(const struct list_head *head)
{
        return head->next == head;
}
```

除了用LIST_HEAD()宏在声明的时候初始化一个链表以外，Linux还提供了一个INIT_LIST_HEAD宏用于运行时初始化链表：

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
```

```
    list->next = list;
    list->prev = list;
}
```

我们用INIT_LIST_HEAD(&nf_sockopts)来使用它。

```
* 插入/删除/合并
```

a 插入

对链表的插入操作有两种：在表头插入和在表尾插入。Linux为此提供了两个接口：

```
static inline void list_add(struct list_head *new, struct list_head *head);
static inline void list_add_tail(struct list_head *new, struct list_head *head);
```

因为Linux链表是循环表，且表头的next、prev分别指向链表中的第一个和最末一个节点，所以，list_add和list_add_tail的区别并不大，实际上，Linux分别用

```
__list_add(new, head, head->next);       /*头插*/
__list_add(new, head->prev, head);       /*尾插*/
```

来实现两个接口，可见，在表头插入是插入在head之后，而在表尾插入是插入在head->prev之后。

假设有一个新nf_sockopt_ops结构变量new_sockopt需要添加到nf_sockopts链表头，我们应当这样操作：

```
list_add(&new_sockopt.list, &nf_sockopts);
```

从这里我们看出，nf_sockopts链表中记录的并不是new_sockopt的地址，而是其中的list元素的地址。如何通过链表访问到new_sockopt呢？下面会有详细介绍。

b 删除

```
static inline void list_del(struct list_head *entry);
```

当我们需要删除nf_sockopts链表中添加的new_sockopt项时，我们这么操作：

```
list_del(&new_sockopt.list);
```

被剔除下来的new_sockopt.list，prev、next指针分别被设为LIST_POSITION2和LIST_POSITION1两个特殊值，这样设置是为了保证不在链表中的节点项不可访问—对LIST_POSITION1和

LIST_POSITION2的访问都将引起页故障。与之相对应，list_del_init()函数将节点从链表中解下来之后，调用LIST_INIT_HEAD()将节点置为空链状态。

c 搬移

Linux提供了将原本属于一个链表的节点移动到另一个链表的操作，并根据插入到新链表的位置分为两类：

```
static inline void list_move(struct list_head *list, struct list_head *head);
static inline void list_move_tail(struct list_head *list, struct list_head *head);
```

例如list_move(&new_sockopt.list,&nf_sockopts)会把new_sockopt从它所在的链表上删除，并将其再链入nf_sockopts的表头。

d 合并

除了针对节点的插入、删除操作，Linux链表还提供了整个链表的插入功能：

```
static inline void list_splice(struct list_head *list, struct list_head *head);
```

假设当前有两个链表，表头分别是list1和list2（都是struct list_head变量），当调用list_splice(&list1,&list2)时，只要list1非空，list1链表的内容将被挂接在list2链表上，位于list2和list2.next（原list2表的第一个节点）之间。新list2链表将以原list1表的第一个节点为首节点，而尾节点不变。如图（虚箭头为next指针）

当list1被挂接到list2之后，作为原表头指针的list1的next、prev仍然指向原来的节点，为了避免引起混乱，Linux提供了一个list_splice_init()函数：

```
static inline void list_splice_init(struct list_head *list, struct list_head *head);
```

该函数在将list合并到head链表的基础上，调用INIT_LIST_HEAD(list)将list设置为空链。

* 遍历

遍历是链表最经常的操作之一，为了方便核心应用遍历链表，Linux链表将遍历操作抽象成几个宏。在介绍遍历宏之前，我们先看看如何从链表中访问到我们真正需要的数据项。

a 由链表节点到数据项变量

我们知道，Linux链表中仅保存了数据项结构中list_head成员变量的地址，那么我们如何通过这个list_head成员访问到作为它的所有者的节点数据呢？Linux为此提供了一个list_entry(ptr,type,member)宏，其中ptr是指向该数据中list_head成员的指针，也就是存储在链表中的地址值，type是数据项的类型，member则是数据项类型定义中list_head成员的变量名，例如，我们要访问nf_sockopts链表中首个nf_sockopt_ops变量，则如此调用：

```
list_entry(nf_sockopts->next, struct nf_sockopt_ops, list);
```

这里”list”正是nf_sockopt_ops结构中定义的用于链表操作的节点成员变量名。

list_entry的使用相当简单，相比之下，它的实现则有一些难懂：

```
#define list_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({                          \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);        \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

b 遍历宏

在的nf_register_sockopt()函数中有这么一段话：

```
        ......
struct list_head *i;
        ......
    list_for_each(i, &nf_sockopts) {
        struct nf_sockopt_ops *ops = (struct nf_sockopt_ops *)i;
        ......
    }
    ......
```

函数首先定义一个(struct list_head *)指针变量i，然后调用list_for_each(i,&nf_sockopts)进行遍历。在<include/linux/list.h>中，list_for_each()宏是这么定义的：

```
#define list_for_each(pos, head) \
    for (pos = (head)->next, prefetch(pos->next); pos != (head); \
        pos = pos->next, prefetch(pos->next))
```

它实际上是一个for循环，利用传入的pos作为循环变量，从表头head开始，逐项向后(next方向)移动pos，直至又回到head(prefetch()可以不考虑，用于预取以提高遍历速度)。

那么在nf_register_sockopt()中实际上就是遍历nf_sockopts链表。为什么能直接将获得的list_head成员变量地址当成struct nf_sockopt_ops数据项变量的地址呢？我们注意到在struct nf_sockopt_ops结构中，list是其中的第一项成员，因此，它的地址也就是结构变量的地址。更规范的获得数据变量地址的用法应该是：

```
struct nf_sockopt_ops *ops = list_entry(i, struct nf_sockopt_ops, list);
```

大多数情况下，遍历链表的时候都需要获得链表节点数据项，也就是说list_for_each()和list_entry()总是同时使用。对此Linux给出了一个list_for_each_entry()宏：

```
#define list_for_each_entry(pos, head, member)                  ......
```

与list_for_each()不同，这里的pos是数据项结构指针类型，而不是(struct list_head *)。nf_register_sockopt()函数可以利用这个宏而设计得更简单：

```
……
struct nf_sockopt_ops *ops;
list_for_each_entry(ops,&nf_sockopts,list){
    ……
}
……
```

某些应用需要反向遍历链表，Linux提供了list_for_each_prev()和list_for_each_entry_reverse()来完成这一操作，使用方法和上面介绍的list_for_each()、list_for_each_entry()完全相同。

如果遍历不是从链表头开始，而是从已知的某个节点pos开始，则可以使用list_for_each_entry_continue(pos
有时还会出现这种需求，即经过一系列计算后，如果pos有值，则从pos开始遍历，如果没有，则从链表头开始，为此，Linux专门提供了一个list_prepare_entry(pos,head,member)宏，将它的返回值作为list_for_each_entry_continue()的pos参数，就可以满足这一要求。

* 安全性考虑

在并发执行的环境下，链表操作通常都应该考虑同步安全性问题，为了方便，Linux将这一操作留给应用自己处理。Linux链表自己考虑的安全性主要有两个方面：

a list_empty()判断

基本的list_empty()仅以头指针的next是否指向自己来判断链表是否为空，Linux链表另行提供了一个list_empty_careful()宏，它同时判断头指针的next和prev，仅当两者都指向自己时才返回真。这主要是为了应付另一个cpu正在处理同一个链表而造成next、prev不一致的情况。但代码注释也承认，这一安全保障能力有限：除非其他cpu的链表操作只有list_del_init()，否则仍然不能保证安全，也就是说，还是需要加锁保护。

b 遍历时节点删除

前面介绍了用于链表遍历的几个宏，它们都是通过移动pos指针来达到遍历的目的。但如果遍历的操作中包含删除pos指针所指向的节点，pos指针的移动就会被中断，因为list_del(pos)将把pos的next、prev置成LIST_POSITION2和LIST_POSITION1的特殊值。

当然，调用者完全可以自己缓存next指针使遍历操作能够连贯起来，但为了编程的一致性，Linux链表仍然提供了两个对应于基本遍历操作的“_safe”接口：list_for_each_safe(pos, n, head)、list_for_each_entry_safe(pos, n, head, member)，它们要求调用者另外提供一个与pos同类型的指针n，在for循环中暂存pos下一个节点的地址，避免因pos节点被释放而造成的断链。

4.2 队列

内核队列定义在

```
/*include/kernel/kfifo.h*/
/*lib/kfifo.c*/此文件原来在定义在kernel/kfifo.c
```

linux的kfifo和多数其他队列实现类似，提供了两个主要操作：enqueue(入队)和dequeue(出队)。kfifo对象维护了两个偏移量：入口偏移和出口偏移。入口偏移是指下一次入队时的位置，出口偏移是指下一次出队列时的位置。出口偏移总是小于等于入口偏移，否则无意义，因为那样说明要出队列的元素根本还有入队列。

队列缓冲区申请时必须是2的幂，如选用PAGE_SIZE(4096)

```
int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask)  //动态初始化
void kfifo_free(struct kfifo *fifo)     //释放kfifo_alloc申请的存储空间
void kfifo_init(struct kfifo *fifo, void *buffer, unsigned int size)  //自己为队列指定缓冲区
DECLEARE_KFIFO(name, size)
INIT_KFIFO(name)      //静态初始化队列，不常用
unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len)   //入队，把from指针所指的len个字节入队，成功返回入队字节个数
unsigned int kfifo_out(struct kfifo *fifo, void *to, unsigned int len)  //出队
unsigned int kfifo_out_peek(struct kfifo *fifo, void *to, unsigned int len, unsigned offset)    //不改变出队偏移，获取队列中数据
static inline unsigned int kfifo_size(struct kfifo *fifo)    //队列空间总大小
static inline unsigned int kfifo_len(struct kfifo *fifo)    //已入队数据大小
static inline unsigned int kfifo_avail(struct kfifo *fifo)    //队列剩余空间大小
static inline int kfifo_is_empty(struct kfifo *fifo)    //队列是否为空
static inline int kfifo_is_full(struct kfifo *fifo)    //队列是否为满
static inline void  kfifo_reset(struct kfifo *fifo)    //抛弃队列里所有内容，重置
```

上述函数以宏定义的形式定义在kfifo.h中，这些函数又去调用各自的底层函数如kfifo_alloc调用kfifo_alloc，kfifo_alloc定义在kfifo.c中。如：

```
/*linux/kfifo.h*/
/**
 * kfifo_alloc - dynamically allocates a new fifo buffer
 * @fifo: pointer to the fifo
 * @size: the number of elements in the fifo, this must be a power of 2
 * @gfp_mask: get_free_pages mask, passed to kmalloc()
 *
 * This macro dynamically allocates a new fifo buffer.
 *
 * The numer of elements will be rounded-up to a power of 2.
 * The fifo will be release with kfifo_free().
 * Return 0 if no error, otherwise an error code.
 */
#define kfifo_alloc(fifo, size, gfp_mask) \
__kfifo_int_must_check_helper( \
({ \
    typeof((fifo) + 1) __tmp = (fifo); \
    struct __kfifo *__kfifo = &__tmp->kfifo; \
    __is_kfifo_ptr(__tmp) ? \
    __kfifo_alloc(__kfifo, size, sizeof(*__tmp->type), gfp_mask) : \
    -EINVAL; \
}) \
)


/*lib/kfifo.c*/
```

```
int __kfifo_alloc(struct __kfifo *fifo, unsigned int size,
        size_t esize, gfp_t gfp_mask)
{
    /*
     * round down to the next power of 2, since our 'let the indices
     * wrap' technique works only in this case.
     */
    size = roundup_pow_of_two(size);

    fifo->in = 0;
    fifo->out = 0;
    fifo->esize = esize;

    if (size < 2) {
        fifo->data = NULL;
        fifo->mask = 0;
        return -EINVAL;
    }

    fifo->data = kmalloc(size * esize, gfp_mask);

    if (!fifo->data) {
        fifo->mask = 0;
        return -ENOMEM;
    }
    fifo->mask = size - 1;

    return 0;
}
EXPORT_SYMBOL(__kfifo_alloc);
```

4.3 映射

4.4 二叉树-红黑树

4.4.1 红黑树节点

```
struct NODE {
   int key;
   int color;
   struct NODE *left;
   struct NODE *right;
   struct NODE *p;
};

| 要素  | 含义     |
|-------+----------|
| key   | 数据域   |
| color | 节点颜色 |
| left  | 左孩子   |
```

```
| right | 右孩子   |
| p     | 父节点   |
```

4.4.2 红黑树判定

```
1. 每个节点要么红，要么黑
2. 根节点是黑的
3. 每个叶节点都是黑的（叶节点是指树尾端NIL或NULL节点）
4. 如果一个节点是红的，那么它的俩个儿子都是黑的
5. 对于任一节点，其到叶节点树尾端NIL结点的每一条路径都包含相同数目的黑节点
（注：上述描述的NULL节点和NIL节点均为叶节点（最末端为NULL的节点），非所谓的子节点）
```

增删节点后的恢复操作 左旋：某节点进行左旋，变成其右孩子的左子树，原右孩子的左子树变为此左旋节点的右孩子。

右旋：某节点进行右旋，变成其左孩子的右子树，原左孩子的右子树变为此右旋节点的左孩子。

树的旋转能保证原树的搜索性质，但不能保证树的红黑性质。

在树插入或删除节点后，利用旋转和重着色，维持其红黑树性质。

在对树进行插入操作时我们一般总是插入红色节点，这样可以在插入操作时尽量避免对树的调整。

由于我们是按照二叉树的方式插入的，因此二叉树的搜索性质不会改变，还是有序二叉树。

插入操作引起的红黑树失衡

* 插入节点（红色）是根节点，则会引起性质2被破坏。
* 插入节点（红色）的父节点是红色，则引起性质4被破坏。
* 总而言之，插入节点为红色时，只能引起性质2或性质4被破坏。如果能解决此问题，则插入操作后仍然是红黑树

插入失衡后的恢复策略

* 把违背红黑树性质的节点上移，如果能移到根节点，那么就能通过直接修改根节点来恢复红黑树的性质。
* 穷举所有的可能性，之后归类处理，不能直接处理的归为几种特殊情况处理。

1. 插入的是根节点

原树是空树，此时只违背性质2。

策略：直接把此节点涂为黑色

2. 插入节点的父节点是黑色

这种情况不会违背性质2和性质4，红黑树性质没有破坏。

策略：什么都不做

3. 插入节点的父节点是红色且祖父节点的另一个子节点（叔叔节点）是红色

此时插入节点父节点的父节点一定存在，否则插入前就已不是红黑树

策略：

4.4.3 内核中的红黑树

内核中提供了一个高效的红黑树，但是没有提供查找和插入操作，因为这个操作和具体元素有关，c语言不太容易提供这种泛型，需要使用红黑树的用户自己实现。

红黑树定义在

```
/*include/linux/rbtree.h*/
/*lib/rbtree.c*/
/*Documentation/rbtree.txt*/   红黑树使用例子和介绍文档
```

第 5 章

驱动开发工具

第 6 章

sysfs系统

第 7 章

proc系统

第 8 章

VFS文件系统

8.1 struct file

```
/* include/linux/fs.h */

struct file {
    union {
        struct llist_node        fu_llist;       //文件对象链表
        struct rcu_head         fu_rcuhead;     //2.6内核后增加的RCU锁，read copy update
    } f_u;
    struct path                f_path;                 //包含目录项
#define f_dentry        f_path.dentry
    struct inode                *f_inode;        /* cached value */
    const struct file_operations        *f_op;  //包含着对文件操作的函数，如lseek，read等

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t                f_lock;                //单个文件结构锁
    atomic_long_t                f_count;           //文件对象的使用计数
    unsigned int                f_flags;            //打开文件时指定的标志，如O_NONBLOCK
    fmode_t                        f_mode;                //文件访问标志，如O_RDONLY
    struct mutex                f_pos_lock;
    loff_t                        f_pos;                //文件当前位移量（文件指针）
    struct fown_struct        f_owner;            //拥有者通过信号进行异步I/O数据的传送
    const struct cred        *f_cred;
    struct file_ra_state        f_ra;           //预读状态

    u64                        f_version;                //版本号，每次操作加1
#ifdef CONFIG_SECURITY
    void                        *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void                        *private_data;          //tty设备驱动的钩子

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head        f_ep_links;         //事件池链表
```

```
    struct list_head        f_tfile_llink;      //
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space        *f_mapping;     //页缓存映射
} __attribute__((aligned(4)));        /* lest something weird decides that 2 is OK */
```

* struct path f_path;

被定义在linux/include/linux/namei.h中，其原型为：

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

在早些版本的内核中并没有此结构，而是直接将path的两个数据成员作为struct file的数据成员，struct vfsmount *mnt的作用是指出该文件的已安装的文件系统，表示当前文件所在文件系统的挂载根目录,struct dentry *dentry是与文件相关的目录项对象。

8.2 struct files_struct

```
/* include/linux/fdtable.h */

/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;                                     //引用计数
    struct fdtable __rcu *fdt;                          //指向fdtable
    struct fdtable fdtab;     //内核专门使用fdtable结构（该结构也称为文件描述符表）来描述文件描述符。该字段为初始的文件描述符表
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    int next_fd;                                        //最近关闭的文件描述符中数值最小的下一个可用的文件描述符
    unsigned long close_on_exec_init[1];                //执行exec()时需要关闭的文件描述符
    unsigned long open_fds_init[1];                     //当前已经打开的文件描述符
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];      //文件对象的初始化数组,NR_OPEN_DEFAULT x86下为32
};
```

8.3 struct dentry

```
/* include/linux/dcache.h */

struct dentry {
    /* RCU lookup touched fields */
    unsigned int d_flags;               /* protected by d_lock */
    seqcount_t d_seq;               /* per dentry seqlock */
    struct hlist_bl_node d_hash;        /* lookup hash list */
    struct dentry *d_parent;        /* parent directory */
    struct qstr d_name;
    struct inode *d_inode;              /* Where the name belongs to - NULL is
                    * negative */
    unsigned char d_iname[DNAME_INLINE_LEN];        /* small names */

    /* Ref lookup also touches following */
    struct lockref d_lockref;       /* per-dentry lock and refcount */
    const struct dentry_operations *d_op;
    struct super_block *d_sb;       /* The root of the dentry tree */
    unsigned long d_time;               /* used by d_revalidate */
    void *d_fsdata;                       /* fs-specific data */

    struct list_head d_lru;               /* LRU list */
    /*
     * d_child and d_rcu can share memory
     */
    union {
        struct list_head d_child;       /* child of parent list */
        struct rcu_head d_rcu;
    } d_u;
    struct list_head d_subdirs;        /* our children */
    struct hlist_node d_alias;        /* inode alias list */
};
```

8.4 struct inode

```
/* include/linux/fs.h */

struct inode {
    umode_t                         i_mode;
    unsigned short                i_opflags;
    kuid_t                          i_uid;
    kgid_t                          i_gid;
    unsigned int                i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl        *i_acl;
    struct posix_acl        *i_default_acl;
#endif
```

```
    const struct inode_operations       *i_op;
    struct super_block        *i_sb;
    struct address_space        *i_mapping;

#ifdef CONFIG_SECURITY
    void                        *i_security;
#endif

    /* Stat data, not accessed from path walking */
    unsigned long               i_ino;
    /*
     * Filesystems may only read i_nlink directly.  They shall use the
     * following functions for modification:
     *
     *    (set|clear|inc|drop)_nlink
     *    inode_(inc|dec)_link_count
     */
    union {
        const unsigned int i_nlink;
        unsigned int __i_nlink;
    };
    dev_t                       i_rdev;
    loff_t                       i_size;
    struct timespec               i_atime;
    struct timespec               i_mtime;
    struct timespec               i_ctime;
    spinlock_t                i_lock;        /* i_blocks, i_bytes, maybe i_size */
    unsigned short          i_bytes;
    unsigned int               i_blkbits;
    blkcnt_t                i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t                i_size_seqcount;
#endif

    /* Misc */
    unsigned long               i_state;
    struct mutex                i_mutex;

    unsigned long               dirtied_when;        /* jiffies of first dirtying */

    struct hlist_node        i_hash;
    struct list_head        i_wb_list;        /* backing dev IO list */
    struct list_head        i_lru;                /* inode LRU list */
    struct list_head        i_sb_list;
    union {
        struct hlist_head        i_dentry;
        struct rcu_head                i_rcu;
    };
    u64                        i_version;
    atomic_t                i_count;
    atomic_t                i_dio_count;
    atomic_t                i_writecount;
#ifdef CONFIG_IMA
    atomic_t                i_readcount; /* struct files open RO */
#endif
```

```
    const struct file_operations    *i_fop;     /* former ->i_op->default_file_ops */
    struct file_lock        *i_flock;
    struct address_space        i_data;
#ifdef CONFIG_QUOTA
    struct dquot            *i_dquot[MAXQUOTAS];
#endif
    struct list_head        i_devices;
    union {
        struct pipe_inode_info      *i_pipe;
        struct block_device     *i_bdev;
        struct cdev             *i_cdev;
    };

    __u32                       i_generation;

#ifdef CONFIG_FSNOTIFY
    __u32                       i_fsnotify_mask; /* all events this inode cares about */
    struct hlist_head       i_fsnotify_marks;
#endif

    void                        *i_private; /* fs or device private pointer */
};
```

8.5 super_operations

```
/* include/linux/fs.h */

struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*dirty_inode) (struct inode *, int flags);
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    int (*drop_inode) (struct inode *);
    void (*evict_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*umount_begin) (struct super_block *);

    int (*show_options)(struct seq_file *, struct dentry *);
    int (*show_devname)(struct seq_file *, struct dentry *);
    int (*show_path)(struct seq_file *, struct dentry *);
    int (*show_stats)(struct seq_file *, struct dentry *);
#ifdef CONFIG_QUOTA
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
#endif
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
```

```
    long (*nr_cached_objects)(struct super_block *, int);
    long (*free_cached_objects)(struct super_block *, long, int);
};
```

8.6 inode_operations

```
/* include/linux/fs.h */
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    int (*permission) (struct inode *, int);
    struct posix_acl * (*get_acl)(struct inode *, int);

    int (*readlink) (struct dentry *, char __user *,int);
    void (*put_link) (struct dentry *, struct nameidata *, void *);

    int (*create) (struct inode *,struct dentry *, umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
            struct inode *, struct dentry *);
    int (*rename2) (struct inode *, struct dentry *,
            struct inode *, struct dentry *, unsigned int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
              u64 len);
    int (*update_time)(struct inode *, struct timespec *, int);
    int (*atomic_open)(struct inode *, struct dentry *,
               struct file *, unsigned open_flag,
               umode_t create_mode, int *opened);
    int (*tmpfile) (struct inode *, struct dentry *, umode_t);
    int (*set_acl)(struct inode *, struct posix_acl *, int);
} ____cacheline_aligned;
```

8.7 dentry_operations

```
/* include/linux/dcache.h */

struct dentry_operations {
```

```
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, struct qstr *);
    int (*d_compare)(const struct dentry *, const struct dentry *,
            unsigned int, const char *, const struct qstr *);
    int (*d_delete)(const struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_prune)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
    char *(*d_dname)(struct dentry *, char *, int);
    struct vfsmount *(*d_automount)(struct path *);
    int (*d_manage)(struct dentry *, bool);
} ____cacheline_aligned;
```

8.8 file_operations

```
/* include/linux/fs.h */

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
   unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    int (*show_fdinfo)(struct seq_file *m, struct file *f);
};
```

8.9 struct fs_struct

```
/* include/linux/fs_struct.h */
struct fs_struct {
    int users;
    spinlock_t lock;
    seqcount_t seq;
    int umask;
    int in_exec;
    struct path root, pwd;
};
```

8.10 struct mmt_namespace

新内核中已经取消

8.11 struct file_system_type

```
/* include/linux/fs.h */

struct file_system_type {
    const char *name;
    int fs_flags;
#define FS_REQUIRES_DEV                 1
#define FS_BINARY_MOUNTDATA         2
#define FS_HAS_SUBTYPE                4
#define FS_USERNS_MOUNT                 8        /* Can be mounted by userns root */
#define FS_USERNS_DEV_MOUNT         16 /* A userns mount does not imply MNT_NODEV */
#define FS_RENAME_DOES_D_MOVE       32768      /* FS will handle d_move() during rename() internally. */
    struct dentry *(*mount) (struct file_system_type *, int,
               const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct hlist_head fs_supers;

    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
    struct lock_class_key s_vfs_rename_key;
    struct lock_class_key s_writers_key[SB_FREEZE_LEVELS];

    struct lock_class_key i_lock_key;
    struct lock_class_key i_mutex_key;
    struct lock_class_key i_mutex_dir_key;
};
```

8.12 struct vfsmount

```
/* include/linux/mount.h */
struct vfsmount {
    struct dentry *mnt_root;        /* root of the mounted tree */
    struct super_block *mnt_sb;        /* pointer to superblock */
    int mnt_flags;
};
```

第 9 章

内核模块

第 10 章

内核编译

第 11 章

自旋锁

第 12 章

字符设备驱动

第 13 章

misc杂项设备驱动

第 14 章

Linux 中断

Linux中断分为两个半部：上半部（tophalf）和下半部(bottom half)。上半部的功能是“登记中断”，当一个中断发生时，它进行相应地硬件读写后就把中断例程的下半部挂到该设备的下半部执行队列中去。因此，上半部执行的速度就会很快，可以服务更多的中断请求。但是，仅有“登记中断”是远远不够的，因为中断的事件可能很复杂。因此，Linux引入了一个下半部，来完成中断事件的绝大多数使命。下半部和上半部最大的不同是下半部是可中断的，而上半部是不可中断的，下半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断！下半部则相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。

中断号的查看可以使用下面的命令：“cat/proc/interrupts”。

Linux实现下半部的机制主要有tasklet和工作队列。

14.1 进程上下文和中断上下文

处理器总处于以下状态中的一种：

1、内核态，运行于进程上下文，内核代表进程运行于内核空间；
2、内核态，运行于中断上下文，内核代表硬件运行于内核空间；
3、用户态，运行于用户空间。

用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存器值、变量等。所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。

硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。所谓的“中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。

当一个进程在执行时,CPU的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够必得到切换时的状态执行下去。在LINUX中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时,内核就在被中断

进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到的资源，以便中继服务结束时能恢复被中断进程的执行。

14.2 上半部

在linux设备驱动中，使用中断的设备需要申请和释放对应的中断，分别使用内核提供的request（）和free（）函数。

申请IRQ

```
int request_irq(unsigned int irq, void (*handler)(int irq, void *dev_id ...),
                unsigned long irqflags, const char *devname, void *dev_id);
其中
irq:要申请的硬件中断号
irqflags是要处理的中断属性：
IRQF_SHARED:就是旧时代的SA_SHIRQ，共享中断号
IRQF_DISABLED:就是就时代的SA_INTERRUPT,设置了该标志，则执行ISR时关本地中断
IRQF_SAMPLE_RANDON：告诉内核，本中断源可以用作随机数发生器的熵池
dev_id:在中断共享时会用到，一般设置这个设备的设备结构体或者NULL
request_irq()返回0表示成功，返回-INVAL表示中断号无效或处理函数指针为NULL，返回-EBUSY表示中断已经被占用且不能共享。
```

释放IRQ

```
void free_irq(unsigned int irq, void *dev_id);
```

使能和屏蔽中断

```
下列函数用于屏蔽和使能一个中断源
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);

disable_irq与disable_irq_nosync的区别在于前者等待到irq的处理已经完成才返回，后者返回前并不保证irq的处理函数已经完成

这3个函数作用于可编程控制器，因此对所有的CPU都生效。

屏蔽本CPU的所有中断：
void local_irq_disable(void);
void local_irq_save(unsigned long flags);
恢复对应中断：
void local_irq_enable(void);
void local_irq_restore(unsigned long flags);
```

14.3 下半部

14.3.1 使用tasklet作为下半部的处理中断的设备驱动程序模板如下：

```
/*定义tasklet和下半部函数并关联*/

struct tasklet my_tasklet;
void my_do_tasklet(unsigned long);

DECLARE_TASKLET(my_tasklet,my_tasklet_func, 0);

/*中断处理下半部*/
void my_do_tasklet(unsigned long)
{
 ……/*编写自己的处理事件内容*/
}

/*中断处理上半部*/
irpreturn_t my_interrupt(unsigned intirq,void *dev_id)
{
……
/*调度my_tasklet函数，根据声明将去执行my_tasklet_func函数*/
tasklet_schedule(&my_tasklet);
……
}

/*设备驱动的加载函数*/
int __init xxx_init(void)
{
……
/*申请中断,转去执行my_interrupt函数并传入参数*/
result=request_irq(my_irq,my_interrupt,IRQF_DISABLED,"xxx",NULL);
……
}

/*设备驱动模块的卸载函数*/
void __exit xxx_exit(void)
{
……
/*释放中断*/
free_irq(my_irq,my_interrupt);
……
}
```

14.3.2 使用工作队列处理中断下半部的设备驱动程序模板如下：

```
/*定义工作队列和下半部函数并关联*/
struct work_struct my_wq;
void my_do_work(unsigned long);

/*中断处理下半部*/
void my_do_work(unsigned long)
{
```

```
 ……/*编写自己的处理事件内容*/
}

/*中断处理上半部*/
irpreturn_t my_interrupt(unsigned intirq,void *dev_id)
{
……
/*调度my_wq函数，根据工作队列初始化函数将去执行my_do_work函数*/
schedule_work(&my_wq);
……
}

/*设备驱动的加载函数*/
int __init xxx_init(void)
{
……
/*申请中断,转去执行my_interrupt函数并传入参数*/
result=request_irq(my_irq,my_interrupt,IRQF_DISABLED,"xxx",NULL);
……
/*初始化工作队列函数,并与自定义处理函数关联*/
INIT_WORK(&my_irq,(void (*)(void*))my_do_work);
……
}

/*设备驱动模块的卸载函数*/
void __exit xxx_exit(void)
{
……
/*释放中断*/
free_irq(my_irq,my_interrupt);
……
}
```

第 15 章

杂项

64位ubuntu配置环境

```
sudo apt-get install lib32z1
安装交叉编译器
```

15.1 查看设备号

```
cat /proc/devices
```

15.2 查看驱动加载信息

```
dmesg
```

15.3 加载卸载驱动

```
insmod xxx.ko
rmmod xxx
lsmod
```

15.4 创建设备文件