# 第19章 设备驱动程序

# 实验目的

- 理解Linux设备驱动程序的基本知识
- 掌握设备驱动程序的编写原则和过程
- 学会编写设备驱动程序，并进行测试

# 主要内容

- 背景知识
  - 设备管理概述
  - 字符设备
  - 块设备
- 实验内容
  - 实现一个基于主存的虚拟块设备驱动程序

# 设备管理概述

## 设备分类

- 设备分两种类型：字符设备、块设备。与之对应的设备驱动程序也分两类，一类针对字符设备，一类针对块设备。

## 主设备号与次设备号

- Linux中设备也看作文件，称设备文件。每个设备文件对应两个设备号：主设备号和次设备号。主设备号指明唯一的设备类型，即标识设备对应的驱动程序类型，次设备号用于在一组主设备号相同的设备之间唯一标识特定设备。

## 创建设备文件

- 使用mknod命令可创建指定类型的设备文件，同时指定主设备号和次设备号。
- 所有注册的硬件设备的主设备号可在/proc/devices文件中找到。

# 设备文件的VFS处理

- 设备文件与普通文件用相同的函数访问，当访问普通文件时，文件系统将用户操作转换成对磁盘分区中数据块的操作；当访问设备文件时，文件系统需要将用户操作转换成对设备的驱动操作。
- VFS改变打开的设备文件的文件操作，它可以把对设备文件的任一系统调用转换成对设备相关的函数调用，用来对硬件设备完成进程请求的I/O。
- 用户对设备文件进行访问时，文件系统读取设备文件在磁盘上相应的inode，并存入主存inode结构中。内核将文件的主设备号与次设备号写入inode结构中的i_rdev字段，并将i_fop字段设置成def_blk_fops（如果为块设备）或def_chr_fops（如果为字符设备）。

# 主要内容

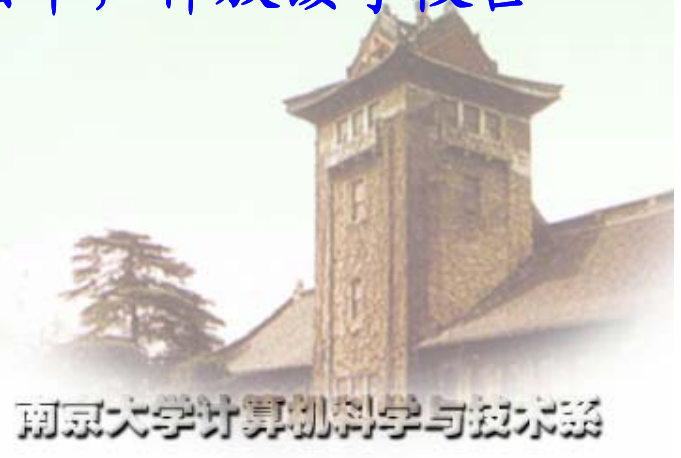
- 背景知识
  - 设备管理概述
  - 字符设备
  - 块设备
- 实验内容
  - 实现一个基于主存的虚拟块设备驱动程序

# 字符设备驱动程序3个内核数据结构(1)

- struct file_operations {
- ……
- 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 *);
- int (*readdir) (struct file *, void *, filldir_t);
- unsigned int (*poll) (struct file *, struct poll_table_struct *);
- int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
- int (*open) (struct inode *, struct file *);
- int (*flush) (struct file *, fl_owner_t id);
- int (*release) (struct inode *, struct file *);
- ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
- ……
- };

# 字符设备驱动程序3个内核数据结构(2)

- struct file{
- …
- mode_t f_mode;　/*文件模式，用于标记文件是否可读或可写*/
- loff_t f_pos;　/*当前的读/写位置*/
- file_operations *f_op; /*与文件相关操作。内核执行open( )操作时对这个指针赋值*/
- void *private_data;　/*驱动程序可将这个字段用于任何目的，但是要在release( )方法中，释放该字段占用的主存*/
- …
- }



# 字符设备驱动程序3个内核数据结构(3)

- 内核用inode结构在内部标识文件，对于单个文件，可能会有许多个表示打开的文件描述符的file结构，但他们都指向同一个inode结构。
- 对于编写字符驱动程序，inode结构中有两个重要的字段:
- struct cdev *i_cdev;
- dev_t i_rdev;
- i_cdev表示字符设备的内核数据结构，当inode指向一个字符设备文件时，该字段包含指向struct cdev结构的指针。i_rdev包含设备编号，内核提供两个宏供开发者使用:
- unsigned int imajor(struct inode *inode); /*获得主设备号*/
- unsigned int iminor(struct inode *inode); /*获得次设备号*/

# 静态申请设备编号

- 静态申请设备编号的内核函数为register_chrdev_region( )，其原型为：
- int register_chrdev_region(dev_t from, unsigned count, const char *name);
- 其中from是要分配的设备编号范围的起始值，count是所请求的连续设备编号的个数。如果count非常大，name是和该编号范围关联的设备名称。在使用register_chrdev_region( )时，开发者必须明确知道所需要的设备编号。一部分主设备号已经静态地分配给大部分常见设备，内核源代码的documentation/devices.txt文件中有这个列表。对于这些设备，由于设备号已经预先确定，因此使用register_chrdev_region( )函数没有什么问题。

# 动态申请设备编号

- 对于一些新设备，设备号并没有预先确定，驱动程序需要对设备号进行动态申请。动态申请块设备号的内核函数原型为:
- **int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char *name);**
- **dev**用于输入参数，在成功完成调用后将保存已分配范围的第**1**个编号。**baseminor**是要使用的被请求第**1**个次设备号。**count**和**name**的含义与**register_chrdev_region( )**的含义一样。使用动态申请设备号的缺点在于每次分配的主设备号不能够保持一致，因此用户不能够预先在**/dev**目录中用**mknod**命令创建设备文件。用户需要在驱动程序注册后，根据**/proc/devices**读取到设备所用到的设备编号，然后创建相应的设备文件。

# 释放设备编号

- 当设备不再使用这些设备编号时，驱动程序需要将申请的设备编号释放。动态申请和静态申请一样，都采用**unregister_chrdev_region( )**释放设备编号，该内核函数的原型为:
- **void unregister_chrdev_region(dev_t from, unsigned count);**
- 驱动程序一般通过模块加载，可在模块的初始化函数进行设备号的申请，并在模块的清理函数中，对设备号进行释放。

# 设备注册

- 内核用cdev结构来表示字符设备，在内核调用设备的驱动操作之前，必须分配并注册一个结构。分配和初始化cdev结构有两种方式，开发者在运行时获得一个独立的cdev结构，可采用如下代码:
- struct cdev *my_cdev = cdev_alloc( );
- my_cdev->ops = &my_fops;
- 开发者也可以通过内核函数cdev_init( )来进行cdev结构的初始化工作。在cdev结构设置好之后，最后的步骤是通过cdev_add( )内核函数将cdev结构添加到内核中，其原型为:
- int cdev_add(struct cdev *p, dev_t dev, unsigned count);
- 这里，p是指向cdev结构的指针，num是该设备对应的第1个设备编号，count是应该和该设备关联的设备编号的数量。如果cdev_add成功返回，驱动程序就随时可被内核调用了，因此，驱动程序在没有完全准备好时，不应该调用cdev_add( )。

# 设备注销

- 从系统中移除一个字符设备，执行如下调用：
- **void cdev_del(struct cdev *dev);**
- 当cdev结构被传递到cdev_del( )后，就不应该再访问cdev结构。

# 主要内容

- 背景知识
  - 设备管理概述
  - 字符设备
  - 块设备
- 实验内容
  - 实现一个基于主存的虚拟块设备驱动程序

# 块设备

## 申请、释放设备号

- 块设备驱动程序在使用前，必须向内核注册主设备号，函数原型为:

**int register_blkdev(unsigned int major, const char *name);**

**major**是需注册的主设备号，如果**major**为**0**，内核动态地分配一个主设备号，并将该设备号返回;**name**为驱动程序在**/proc/dev**中显示的名字。

- 在驱动程序卸载时，注册的主设备号需要归还给内核，该内核函数原型为:

**int unregister_blkdev(unsigned int major, const char *name);**

# 注册磁盘

- 内核中由gendisk结构表示一个独立的磁盘设备，块设备驱动程序在注册主设备号之后，需要设置gendisk结构，并将该结构添加到内核中。

```
struct gendisk {
        int major;              /*块设备的主设备号*/
        int first_minor;        /*块设备的第1个次设备号*/
        int minors;             /*块设备包含的次设备号数量*/
        char disk_name[32];/*磁盘设备名字*/
        struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data; /*供驱动程序自由使用的字段*/
        sector_t capacity;   /*磁盘包含的扇区数*/
        int flags;              /*描述驱动器状态*/
};
```

# gendisk的fops是块设备驱动程序提供给内核的接口

fops的类型为block_device_operations，主要成员：

- int (*open) (struct inode *, struct file *);
- int (*release) (struct inode *, struct file *);
- int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
- int (*media_changed) (struct gendisk *);
- int (*revalidate_disk) (struct gendisk *);

block_device_operations中并没有读和写操作函数，在块设备中，对设备的读写是通过请求队列来实现。

# 分配、删除和注册gendisk结构

- gendisk是动态分配的结构，它需要内核的特殊处理来进行初始化，驱动程序不能自己动态分配该结构，必须执行内核函数:

  **struct gendisk *alloc_disk(int minors);**

  minors是该磁盘使用的次设备号的数量。gendisk在申请后，其minors值不能够被改变。
- 用户不再需要使用gendisk时，必须通过del_gendisk函数将gendisk结构删除。
- 驱动程序将gendisk结构设置完毕后，需要将gendisk注册进内核。gendisk的注册内核函数是:

  **void add_disk(struct gendisk *gd);**

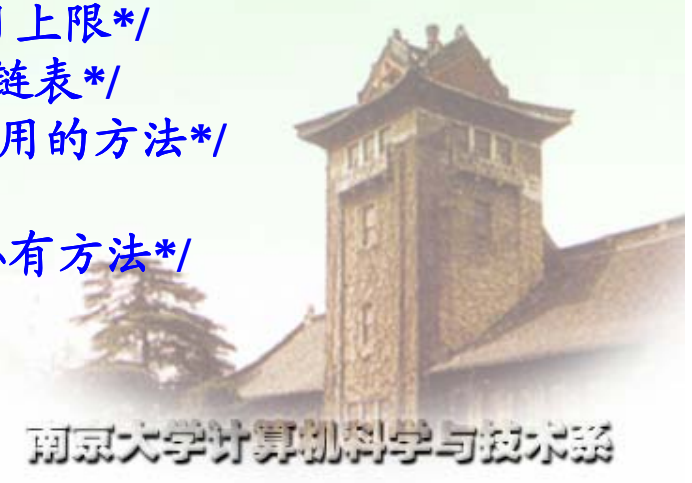
  一旦调用add_disk，内核就可随时操作gendisk。gendisk只有在初始化完毕后才能调用add_disk进行注册。

# bio结构体(1)

- 新的磁盘控制器支持聚散I/O，它能够传输主存中不邻接的数据块。
- v2.5内核版本起为块设备引入bio结构体来实现聚散I/O。
- bio以片段（segment）链表形式组织块I/O操作，一个片段是需要传输的一小块连续的主存缓冲区。即使一个缓冲区分散在主存的多个位置上，bio结构体也能对内核保证I/O操作的执行。

# bio结构体(2)

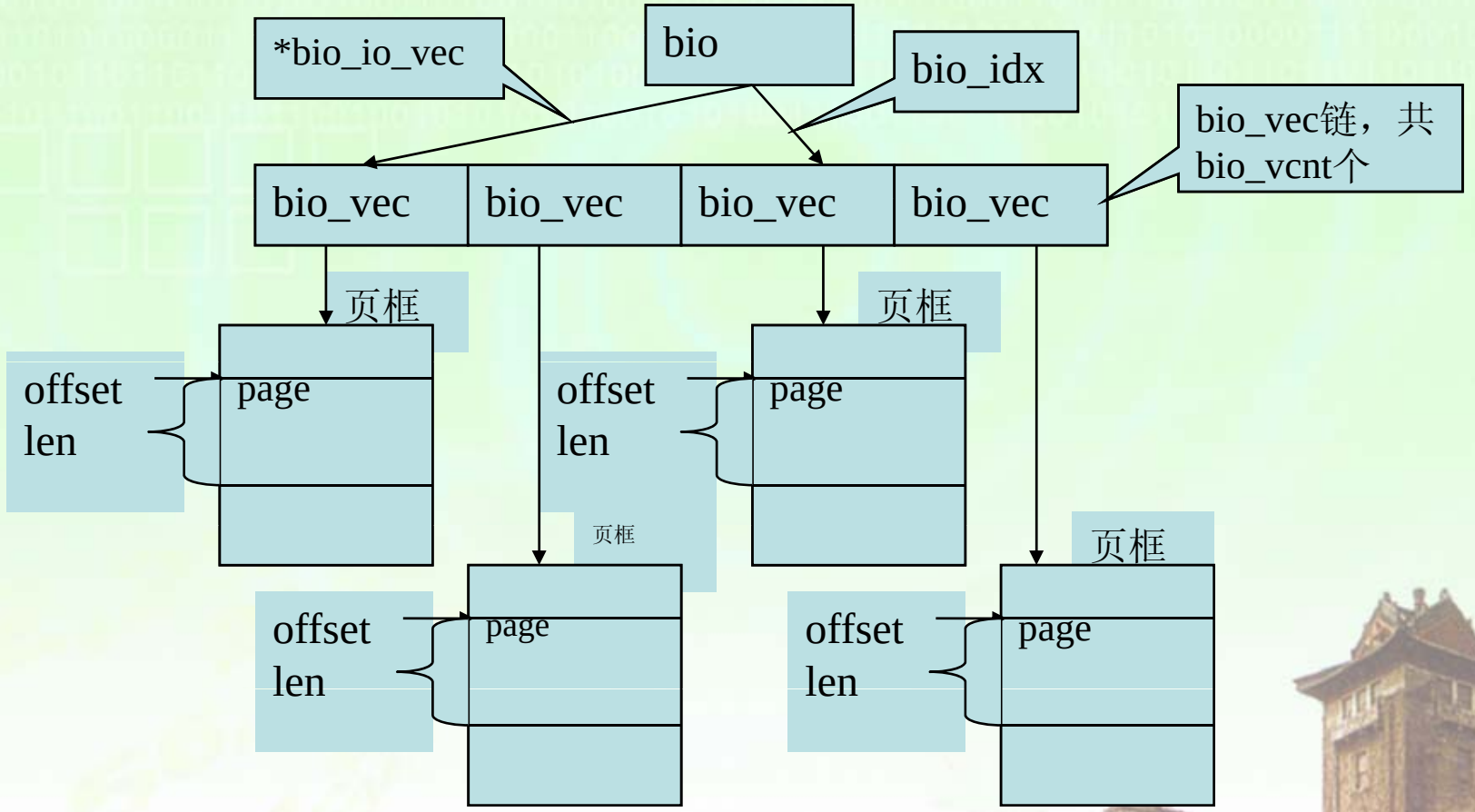
```
struct bio {
    sector_t            bi_sector;          /*磁盘上相关扇区*/
    struct bio          *bi_next;           /*请求队列链表*/
    struct block_device *bi_bdev;           /*相关的块设备*/
    unsigned long       bi_flags;           /*状态和命令标志*/
    unsigned long       bi_rw;              /*读还是写？*/
    unsigned short      bi_vcnt;            /*bio_vec的个数*/
    unsigned short      bi_idx;             /*bi_vec数组当前元素的序号*/
    unsigned short      bi_phys_segments;/*合并后的片段数*/
    unsigned short      bi_hw_segments;   /*DMA重映射时的片段数*/
    unsigned int        bi_size;            /*总的大小、字节为单位*/
    unsigned int        bi_hw_front_size; /*第1个可合并的段大小*/
    unsigned int        bi_hw_back_size;  /*最后－个可合并的段大小*/
    unsigned int        bi_max_vecs;        /*bio_vecs数目上限*/
    struct bio_vec      *bi_io_vec;         /*指向bio_vec链表*/
    bio_end_lO_t        *bi_end_io;         /*I/O完成后调用的方法*/
    atomic_t            bi_cnt;             /*使用计数*/
    void                *bi_private;        /*拥有者的私有方法*/
    bio_destructor_t    *bi_destructor;   /*销毁方法*/
};
```



# bio_vec结构定义

- struct bio_vec{
- struct page *bv_page;
- /*指向该缓冲区所驻留的页框*/
- unsigned int bv_len;
- /*该缓冲区以字节为单位的大小*/
- unsigned int bv_offset;
- /*缓冲区所驻留的页框中以字节为单位的偏移量*/
- };

# bio、bio_vec与page的关系



# 请求队列

- 每个块设备有自己的I/O请求队列。
- 块设备将挂起的块I/O请求保存在请求队列中，该队列由request_queue结构体表示。
- 内核将I/O请求加入到对应请求队列中。请求队列只要不为空，队列对应的块设备驱动程序就会从队列头获取请求，然后将其送入对应的块设备上去操作。
- 请求队列中的每一项都是一个单独的请求，由request结构体表示。
- 因为一个请求可能要操作多个连续的磁盘块，所以每个请求可以由多个bio结构体组成，虽然磁盘上的块必须连续，但是在主存中这些块并不一定要连续，每个bio结构体都可以描述多个片段，而每个请求也可以包含多个bio结构体。

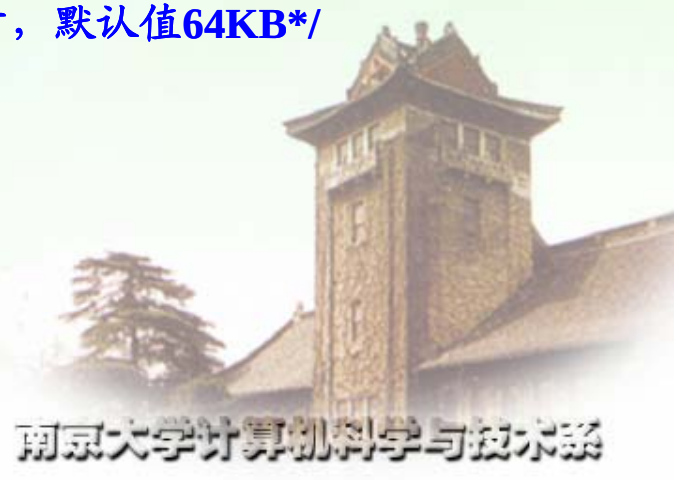
# I/O请求request

- 块I/O层建立请求结构request后，把它放在请求队列中，然后传递给底层驱动程序，触发驱动程序的request_fn( )内核函数来处理请求。

```
struct request {
 struct list_head queuelist;     /*请求队列链表*/
 unsigned long flags;            /*标志*/
 sector_t sector;                /*将提交的下个扇区*/
 unsigned long nr_sectors;       /*提交请求中的所有扇区数*/
 unsigned int current_nr_sectors; /*当前bio中当前片段将提交的扇区数*/
 sector_t hard_sector;           /*将完成的下个扇区*/
 unsigned int hard_nr_sectors;   /*将完成的剩余扇区数*/
 unsigned long hard_cur_sectors; /*当前片段中将完成的剩余扇区数*/
 unsigned long nr_cbio_sectors;  /*当前bio将提交的剩余扇区数*/
 struct bio *cbio;               /*将提交的下个bio*/
 struct bio *bio;                /*将完成的下个未完成的bio*/
 struct gendisk *rq_disk;        /*通用磁盘结构*/
 char *buffer;
 int ref_count;                  /*引用计数*/
 request_queue_t *q;             /*请求队列*/
 struct requeue_list *rl;        /*请求的空闲链表*/
 unsigned short nr_phys_segments; /*合并物理地址后的片段数*/
 …
};
```

# I/O请求队列结构

```
struct request_queue {
    struct list_head queue_head;
    struct request *last_merge;
    elevator_t elevator;
    struct request_list rq;                    /*请求空闲表结构*/
    request_fn_proc *request_fn;               /*指向块设备底层操作函数*/
    merge_request_fn *back_merge_fn;           /*向后合并请求函数*/
    merge_request_fn *front_merge_fn;          /*向前合并请求函数*/
    merge_request_fn *merge_request_fn;/*合并请求函数*/
    make_request_fn *make_request_fn;          /*生成请求函数*/
    unsigned long nr_request;                  /*最大请求数*/
    unsigned short max_sectors;/*能处理的最大尺寸(扇区为单位) */
    unsigned short max_phys_segments;          /*一个请求中能处理的最大片段数*/
    unsigned short max_hw_segments;;           /*一个请求中能处理的最多dma片段数*/
    unsigned short hardsect_size;              /*硬件扇区大小*/
    unsigned int max_segment_size;             /*簇的最大片段尺寸，默认值64KB*/
    struct blk_queue_tag *queue_tags;          /*队列标志*/
};
```



# 创建和初始化请求队列

- 驱动程序创建请求队列：
  **request_queue_t*blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);**
- **rfn**是驱动程序提供的一个函数指针，该函数是真正用于实现设备的读/写请求的；**lock**是驱动程序提供的一个自旋锁，内核通过该自旋锁实现请求队列的同步保护；函数的返回值是一个请求队列的指针。请求队列在初始化后，应该通过**blk_queue_hardsect_size( )**设置请求队列所使用的扇区大小。硬件扇区大小作为一个参数放在请求队列中，而不是放在**gendisk**结构中。

# 归还请求队列

- 驱动程序将请求队列归还给内核，执行该任务的内核函数:

**void blk_cleanup_queue(request_queue_t * q);**

调用该函数之后，就不应该再次访问请求队列了。请求队列中有一个成员为**queuedata**，它是供驱动程序自行使用的，如果驱动程序用该成员指向一段动态分配的主存，那么当驱动程序被卸载时，要负责将该段主存回收。

# 页高速缓存

- 用来减少对磁盘的I/O操作次数，通过把磁盘中的数据缓存到物理主存中，把对磁盘的访问变为对物理主存的访问。它的价值在于:
- 1)访问磁盘的速度要远远低于访问主存的速度，因此，从主存访问数据比从磁盘访问速度更快;
- 2)数据一旦被访问，很有可能在短期内再次被访问到。基于程序局部性原理，如果在第1次访问数据时缓存它，那就极有可能在短期内再次被页高速缓存命中。
- 3)页高速缓存是由主存中的物理页组成的，缓存中每一页都对应着磁盘中的多个块。每当内核开始执行一个页I/O操作时，首先会检查需要的数据是否在高速缓存中，如果在，内核就直接使用高速缓存中的数据，从而避免访问磁盘。

# 页高速缓冲address_space结构体

```
struct address_space {
    struct inode_*hast;                 /*属主的inode*/
    struct list_head clean_pages;   /*干净页面链表*/
    struct list_head dirty_pages;    /*脏页面链表*/
    struct list_head locked_pages;  /*锁定页面链表*/
    struct list_head io_pages;          /*I/O使用页面链表*/
    struct address_space_operations *a_ops;/*操作函数表*/
    struct list_head i_mmap;            /*私有映射链表*/
struct list_head i_mmap_shared;  /*共享映射链表*/
    struct semaphore i_shared_sem; /*保护链表信号量*/
    unsigned long nrpages;                  /*页面总数*/
…
struct list_head private_list;          /*私有address_space链表*/
struct address_space *assoc_mapping;  /*缓冲区*/
};
```

# address_space_operations结构

- struct address_space_operations {
- writepage( ); /*把页写入磁盘*/
- readpage( ); /*从磁盘读入页*/
- sync_page( ); /*启动页中安排的I/O操作，传输数据*/
- prepare_write( ); /*准备写操作 */
- commit_write( ); /*完成写操作*/
- bmap( ); /*从文件块索引获得逻辑块号*/
- flushpage( ); /*删除来自磁盘的页*/
- releasepage( ); /*日志文件系统准备释放页*/
- direct_io( ); /*数据页的直接I/O传输*/
- };

# 块设备文件的读写操作


虚拟文件系统
磁盘高速缓存
磁盘文件系统
磁盘文件系统
磁盘文件系统
通用块设备层
I/O调度程序
块设备驱动程序
块设备驱动程序

# 磁盘文件I/O步骤

- read( )函数将文件描述符、文件内偏移量传递给VFS。
- VFS确定需要获得的数据是否已经缓存在磁盘高速缓存区中。如果数据存在，直接将数据回送给用户并返回。如果数据不在缓存区中，则继续执行。
- 内核通过内核映射层确定数据的物理位置，映射层主要执行下面两个步骤:

  a) 文件被看成是由相同大小的块组成。映射层可以通过块大小、文件偏移量确定需要读取的数据相对于文件开始的块号。

  b) 内核在步骤a)中计算出来的块号只是相对于文件起始地址的块号，还需要将该块号转换成相对于磁盘/分区起始地址的物理块号。内核通过读取文件的inode节点，获得相应信息并计算出该块号。
- 在获得数据在磁盘中的块号后，内核利用通用块层启动I/O操作来传送所请求的数据。一般而言，每个I/O操作只针对磁盘上一组连续的块。由于请求的数据不一定在邻接块中，所以通用块层可能启动几次I/O操作，每次I/O操作通过一个bio结构描述。
- 通用块层下的“I/O调度程序”根据预先定义的内核策略将待处理的I/O数据传送请求进行归类。
- 最后，块设备驱动程序向磁盘控制器的硬件接口发送适当命令，进行实际的数据传送。

# 主要内容

- 背景知识
  - 设备管理概述
  - 字符设备
  - 块设备
- 实验内容
  - 实现一个基于主存的虚拟块设备驱动程序

# 实验 实现一个基于主存的虚拟块设备驱动程序

## 实验说明

- 它是一个RAM-disk驱动程序，当该驱动程序挂载后，用户可以和普通块设备一样在该块设备上创建文件系统，且可通过模块挂载该驱动程序。
- 可以设置虚拟块设备的主设备号，并且可以设置块设备的扇区大小和扇区数量。

# 解决方案

## 基本数据结构

- 块设备管理数据结构 应该包括:
- #define KERNEL_SECTOR_SIZE 512
- struct vblkdev {
- int size;
- u8 *data;
- spinlock_t lock;
- struct request_queue *queue;
- struct gendisk *gd;
- };

# 定义模块参数

加载虚拟块设备时，可定义主设备号，并将扇区大小和扇区数量传递给驱动程序。为此，需要定义3个模块参数：

```
vblkdev_major;        /*设备的主设备号*/
hardsect_size;        /*扇区大小*/
nsectors;             /*扇区数量*/
```

3个参数在模块中的定义如下：

```
static int vblkdev_major = 0;
static int hardsect_size = 512;
static int nsectors = 1024;
module_param(vblkdev_major, int, 0);
module_param(hardsect_size, int, 0);
module_param(nsectors, int, 0);
```

# 设备初始化(1)

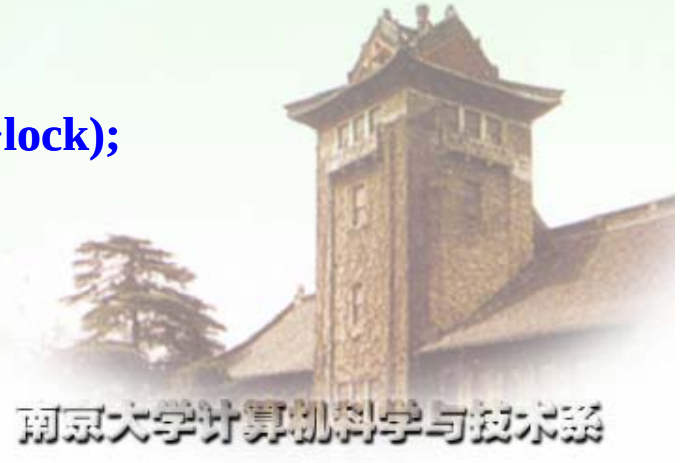
设备初始化主要涉及到如下几个任务：

- 申请并注册主设备号；
- 为虚拟块设备管理数据结构申请主存；
- 为虚拟块设备申请主存；
- 初始化请求队列；
- 初始化**gendisk**结构。

# 设备初始化(2)

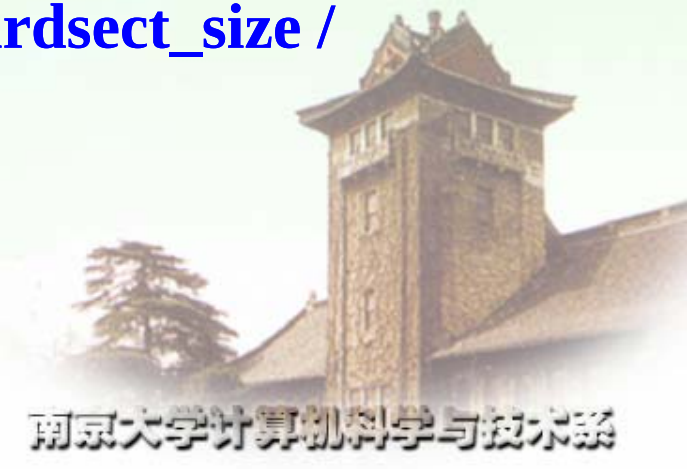
设备初始化任务：

```
{
……
vblkdev_major = register_blkdev(vblkdev_major, "vblkdev");/*申请并注册主设备号*/
…….
dev = kmalloc(sizeof(struct vblkdev), GFP_KERNEL); /*为管理数据结构申请主存*/
memset(dev, 0, sizeof(*dev));
……
dev->size = nsectors * hardsect_size;
dev->data = vmalloc(dev->size); /*为虚拟块设备申请主存*/
……
}
  {
   ……
spin_lock_init(&dev->lock);
   dev->queue = blk_init_queue(vblkdev_request, &dev->lock);
   blk_queue_hardsect_size(dev->queue, hardsect_size);
dev->queue->queuedata = dev;
   ……
}
```



# 设置gendisk结构

- {
- ……
- dev->gd = alloc_disk(MINOR_NO);
- dev->gd->major = vblkdev_major;
- dev->gd->first_minor =1; /*只有一个分区*/
- dev->gd->fops = &vblkdev_ops;
- dev->gd->queue = dev->queue;
- snprintf(dev->gd->disk_name, 32, "vblkdev");
- set_capacity(dev->gd, nsectors * (hardsect_size / KERNEL_SECTOR_SIZE));
- add_disk(dev->gd);
- ……
- }



# Request( )内核函数

- **request( )**内核函数完成数据的读/写操作。
- 该函数通过将请求队列中的每一个请求取出，并根据请求在虚拟块设备的数据数组中进行读写操作。
- 在取出请求前，程序需要获得虚拟块设备数据数组的位置。数据数组记录在**vblkdev**数据结构中。在请求队列初始化时，程序已经将该结构的指针设置在请求队列的**queuedata**成员中，因此程序能够方便的获得虚拟块设备数据数组的位置。

# 本虚拟块设备的request( ) 为vblkdev_request( )

```
static void vblkdev_request(request_queue_t *q)
{
                ……
    struct request *req;
    struct vblkdev *dev = q->queuedata;
                ……
    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
            /*将数据写入数组*/
        end_request(req,1);
    }
}
```

# 使用虚拟块设备

- 通过**insmod**加载模块
- 指定或动态获得主设备号
- 通过**/proc/devices**文件获得主设备号
- 通过**mknod**创建设备文件
- 使用**mount**命令挂载该设备到某个目录
- 使用该虚拟块磁盘设备

# 部分参考源程序(1)

- #include <linux/kernel.h>
- #include <linux/module.h>
- #include <linux/slab.h>
- #include <linux/spinlock.h>
- #include <linux/genhd.h>
- #include <linux/blkdev.h>
- #include <linux/errno.h>
- #include <linux/bio.h>
- #include <linux/hdreg.h>
- static int vblkdev_major = 0; /*主设备号*/
- static int hardsect_size = 512; /*扇区大小*/
- static int nsectors = 1024; /*扇区数量*/
- module_param(vblkdev_major, int, 0);/*定义模块参数*/
- module_param(hardsect_size, int, 0);
- module_param(nsectors, int, 0);

# 部分参考源程序(2)

```
#define KERNEL_SECTOR_SIZE 512
/*定义设备结构*/
struct vblkdev {
    int                 size;/*扇区大小*/
    u8                  *data;
    spinlock_t          lock;
    struct request_queue *queue;
    struct gendisk      *gd;
};
#define MINOR_NO 8
```

# 部分参考源程序(3)

```
/*进行数据传送*/
static void vblkdev_transfer(struct vblkdev *dev, unsigned long sector,unsigned long nsect, char *buffer, int write) {
 unsigned long offset = sector * KERNEL_SECTOR_SIZE;
 unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
```

# 部分参考源程序(4)

```
/* 传送一个bio. */
static int vblkdev_xfer_bio(struct vblkdev *dev, struct bio *bio)
{
    int i;
    struct bio_vec *bvec;
    sector_t sector = bio->bi_sector;
        /* 单独执行每段 */
bio_for_each_segment(bvec, bio, i)
{
        char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
       /* 通过vblkdev_transfer写入数据； */
        sector += bio_cur_sectors(bio);
        __bio_kunmap_atomic(bio, KM_USER0);
    }
    return 0;
}
```

# 部分参考源程序(5)

- /* 传送一个请求 */
- static int vblkdev_xfer_request(struct vblkdev *dev, struct request *req)
- {
- struct bio *bio;
- int nsect = 0;
- rq_for_each_bio(bio, req) {
- /*传送一个bio*/
- }
- return nsect;
- }

# 部分参考源程序(6)

- /* 处理请求. */

```
static void vblkdev_request(request_queue_t *q)
{
    struct request *req;
    int sectors_xferred;
    struct vblkdev *dev = q->queuedata;
while ((req = elv_next_request(q)) != NULL)
{
            /*忽略非读写命令*/
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        /*通过vblkdev_xfer_request传输一个请求*/
 sectors_xferred = vblkdev_xfer_request(dev, req);
        if (! end_that_request_first(req, 1, sectors_xferred)) {
            blkdev_dequeue_request(req);
            end_that_request_last(req, sectors_xferred);
    }
}
```

# 部分参考源程序(7)

```
/*打开设备*/
static int vblkdev_open(struct inode *inode, struct file *filp)
{
    try_module_get(THIS_MODULE);
    struct vblkdev *dev = inode->i_bdev->bd_disk->private_data;
    filp->private_data = dev;
    spin_lock(&dev->lock);
    check_disk_change(inode->i_bdev);
    spin_unlock(&dev->lock);
    return 0;
}
```

# 部分参考源程序(8)

- /*关闭设备*/
- static int vblkdev_release(struct inode *inode, struct file *filp)
- {
- module_put(THIS_MODULE);
- return 0;
- }

# 部分参考源程序(9)

- /*定义块设备操作函数*/
- static struct block_device_operations vblkdev_ops = {
- .owner = THIS_MODULE,
- .open = vblkdev_open,
- .release = vblkdev_release
- };

# 部分参考源程序(10)

```
/*初始化虚拟设备*/
static int setup_device(struct vblkdev *dev)
{
        /*分配dev结构所需主存；*/
        /*初始化请求队列；*/
        /*初始化gendisk；*/
}
struct vblkdev *device;
```

# 部分参考源程序(11)

```
/*模块初始化函数*/
static int __init vblkdev_init(void)
{
        /*注册块设备*/
 vblkdev_major = register_blkdev(vblkdev_major, "vblkdev");
   if (vblkdev_major <= 0) {
      printk(KERN_WARNING "vblkdev: unable to get major number\n");
      return -EBUSY;
   }
   device = kmalloc(sizeof(struct vblkdev), GFP_KERNEL);
   if (device == NULL)
      goto out_unregister;
   if (setup_device(device) != 0)
      goto out_unregister;
    return 0;
 out_unregister:
   unregister_blkdev(vblkdev_major, "vblkdev");
   return -ENOMEM;
}
module_init(vblkdev_init);
```

# 部分参考源程序(12)

```
/*模块清理函数*/
static void __exit vblkdev_exit(void)
{/*删除设备所占用的主存空间；*/
 if (device->gd) {
      del_gendisk(device->gd);
      put_disk(device->gd);
    }
    if (device->queue) {
      blk_cleanup_queue(device->queue);
    }
    if (device->data)
      vfree(device->data);
    unregister_blkdev(vblkdev_major, "vblkdev");
    kfree(device);
}
module_exit(vblkdev_exit);
MODULE_LICENSE("Dual BSD/GPL");
```

# vblkdev验证

- Make
- insmod vblkdev
- cd /dev
- 
- ls -al　　/*可看到名为vblkdev的设备，以及主设备号和次设备号*/
- 
- mknod vblkdev b 254 1　/*创建设备文件。这里b表示块设备，254表示主设备号，1表示次设备号*/
- 
- mkfs.ext2 -b 1024 -i 1024 -m 5 /dev/vblkdev　/*将ext2文件系统装入vblkdev设备中*/
- 
- mount -t ext2 vblkdev /mnt　/*将文件系统类型为ext2，设备类型为vblkdev的设备挂载在/mnt安装点上*/
- 
- cp /home/demo/seminar@weihai/device/vblkdev.c ./　　/*将源码拷到这个设备上*/