LINUX块设备驱动开发

www.farsight.com.cn

知识点

- 块设备驱动程序简介
- 块设备特点及驱动程序工作原理
- 块设备驱动程序的请求队列机制
- 块设备驱动程序的分层实现

块设备驱动程序简介

- 块设备驱动程序提供了面向数据块的设备的访问
- 块设备一般以随机的方式传输数据，并且数据总是具有固定大小的块
- 典型的块设备是磁盘驱动器类设备

块设备驱动程序的特点

- 块设备接口相对复杂，不如字符设备明晰易用
- 块设备驱动程序对整个系统的性能影响较大，速度和效率是设计块设备驱动程要重点考虑的问题
- 系统中使用缓冲区与访问请求的优化管理（合并与重新排序）来提高系统性能

块设备与字符设备的区别

属性\类别	块设备	字符设备
访问单位/次	有固定大小	无固定大小
随机访问 (lseek)	支持	不支持
用户直接访问	不可	可以
驱动程序	复杂	相对简单

block_device_operations结构体

- 在块设备驱动中，有1个类似于字符设备驱动中file_operations结构体的block_device_operations结构体，它是对块设备操作的集合
- struct block_device_operations
  {
  int(*open)(struct inode *, struct file*); //打开
  int(*release)(struct inode *, struct file*); //释放
  int(*ioctl)(struct inode *, struct file *, unsigned, unsigned long); //ioctl
  int(*media_changed)(struct gendisk*); //介质被改变？
  int(*revalidate_disk)(struct gendisk*); //使介质有效
  int(*getgeo)(struct block_device *, struct hd_geometry*);//填充驱动器　信息
  struct module *owner; //模块拥有者
  };

打开和释放

- int (*open)(struct inode *inode, struct file *filp);
- int (*release)(struct inode *inode, struct file *filp);
- 与字符设备驱动类似，当设备被打开和关闭时将调用它们。

IO控制

- int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,unsigned long arg);
- 上述函数是ioctl()系统调用的实现，块设备包含大量的标准请求，这些标准请求由Linux块设备层处理，因此大部分块设备驱动的ioctl()函数相当短。

gendisk结构体

- 在Linux内核中，使用gendisk（通用磁盘）结构体来表示1个独立的磁盘设备（或分区）
- struct gendisk

```
{
  int major; /* 主设备号 */
  int first_minor;  /*第1个次设备号*/
  int minors; /* 最大的次设备数，如果不能分区，则为1*/
  char disk_name[32]; /* 设备名称 */
  struct hd_struct **part; /* 磁盘上的分区信息 */
  struct block_device_operations *fops; /*块设备操作结构体
*/
  struct request_queue *queue;  /*请求队列*/
  void *private_data;  /*私有数据*/
  sector_t capacity; /*扇区数，512字节为1个扇区*/
  …
}
```

操作gendisk(2)

- 增加gendisk

  gendisk结构体被分配之后，系统还不能使用这磁盘，需要调用如下函数来注册这个磁盘设备。

  void add_disk(struct gendisk *gd);

  特别要注意的是对add_disk()的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

操作gendisk(3)

- 释放gendisk

  当不再需要一个磁盘时，应当使用如下函数释放gendisk。

  void del_gendisk(struct gendisk *gd);

request队列结构体

- 一个块请求队列是一个块I/O请求的队列

```
struct request_queue
{/* 保护队列结构体的自旋锁 */
 spinlock_t *queue_lock;
 /* 队列设置 */
 unsigned long nr_requests; /* 最大的请求数量 */
 unsigned short max_sectors;  /* 最大的扇区数 */
 unsigned short hardsect_size;  /* 硬件扇区尺寸 */
 …
}
```

如何执行实际的数据传输

- struct request结构在<linux/blkdev.h>中被定义
- 通常驱动程序访问该结构是通过CURRENT
- 该结构中的字段描述了在缓冲区和物理设备之间传递信息的所有信息
- kdev_t rq_dev;这个字段是请求访问的设备。当request函数处理不同的次设备请求时， rq_dev可以用来表示实际操作的次设备。
- int cmd;该字段描述了要进行的操作，可以是READ或WRITE。
- unsigned long sector;表示本次请求要传输的第一个扇区编号
- unsigned long current_nr_sectors;
- unsigned long nr_sectors;
- 表示当前请求的要传输的扇区（sectors）数目。
- char *buffer;传递一个指针，指向数据要被写入（cmd=READ）或者要被读出（cmd=WRITE）的缓冲区高速缓存区域
- struct buffer_head *bh;该结构描述了本次请求对应缓冲区链表的第一个缓冲区，即缓冲区头

请求队列简介

- request函数是块设备驱动程序里面最重要的函数
- request函数要完成以下几个功能
- 测试请求的有效性
  - 检查请求是否合法有效
  - 做出错误处理
- 执行实际的数据传输
  - 用CURRENT宏来检索当前请求的细节
  - CURRENT宏是一个指向blk[MAJOR_NR].request_queue的指针
- 清除已经处理过的请求
  - 清除操作由end_request函数执行
  - end_request管理请求队列并唤醒等待I/O操作的进程
  - end_request函数同时修改CURRENT，确保它指向下一个待处理的请求
- 返回开头，并且开始处理下一条请求

操作请求队列（1）

- 初始化请求队列

  request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);

  该函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的模块加载函数中调用。

操作请求队列（3）

- 提取请求

```
struct request *elv_next_request(request_queue_t *queue);
```

上述函数用于返回下一个要处理的请求（由I/O调度器决定），如果没有请求则返回NULL。
elv_next_request()不会清除请求。 因为elv_next_request()不从队列里清除请求，因此连续调用它两次，两次会返回同一个请求结构体。

挂载和卸载如何工作

- 块设备不同于字符设备一个明显特征就是——块设备可以被挂载到文件系统上
- mount –t vfat /dev/hda6 /mnt/hda6 将一个硬盘分区hda6挂载到文件系统/mnt/hda6节点上
- 在内核挂载块设备时，调用open方法来访问驱动程序。
- f_mode标志告诉驱动程序，设备是以只读方式(f_mode == FMODE_READ)还是以读写方式(f_mode == (FMODE_READ|FMODE_WRITE))挂载
- 设备打开之后，内核就会调用request方法传输数据块

操作请求队列（5）

- 启停请求队列

void blk_stop_queue(request_queue_t *queue);

void blk_start_queue(request_queue_t *queue);

如果块设备到达不能处理等候的命令的状态，应调用blk_stop_queue()来告知块设备层。之后，请求函数将不被调用，除非再次调用blk_start_queue()将设备恢复到可处理请求的状态。

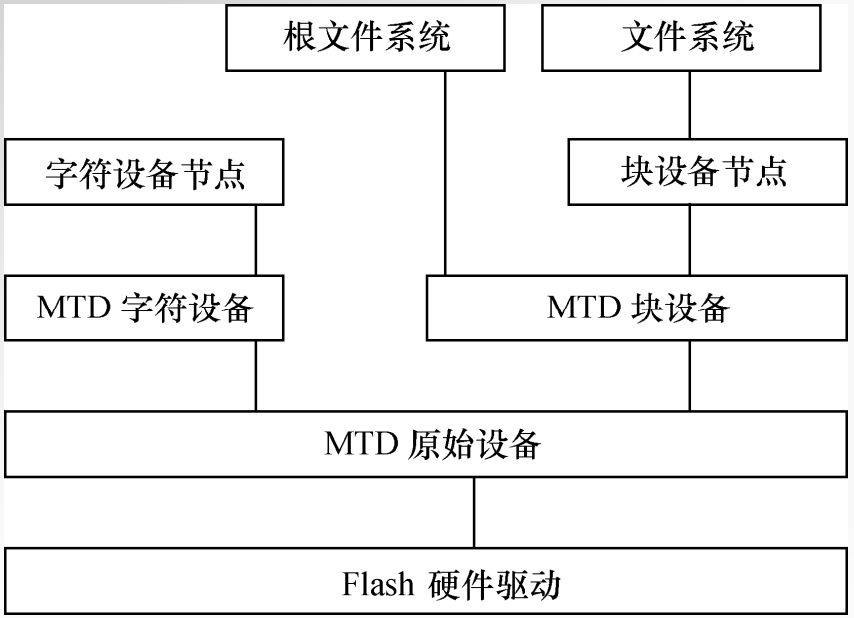
块设备驱动注册与注销

- 驱动注册
- int register_blkdev(unsigned int major, const char *name);
- 驱动注销

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

Linux MTD系统层次

- Flash设备驱动及接口可分为4层 :设备节点、MTD设备层、MTD原始设备层和硬件驱动层



MTD字符设备驱动程序

- MTD字符驱动程序允许直接访问flash器件
- 通常用来在flash期间上创建文件系统，也可以用来直接访问不频繁修改的数据

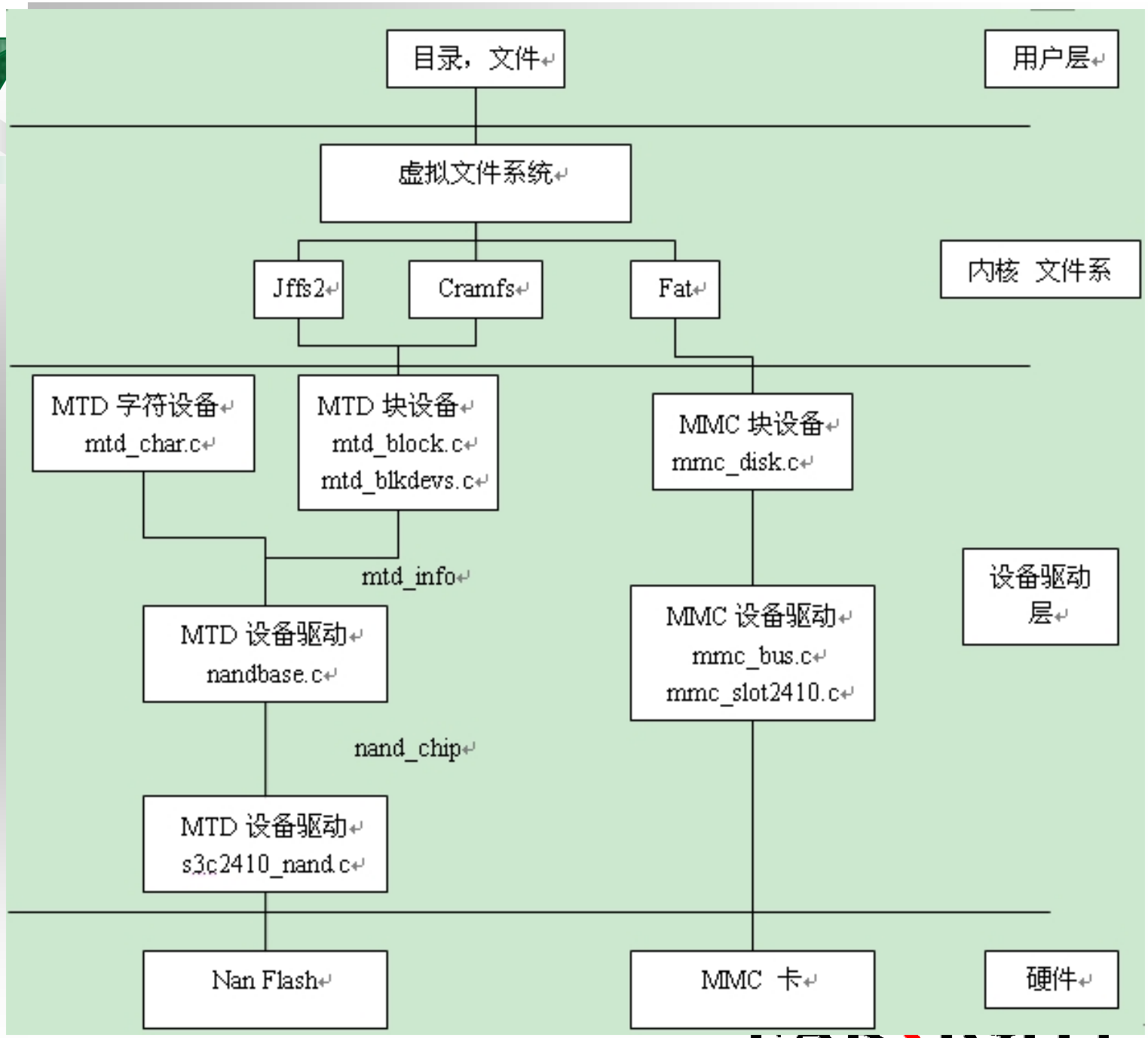
MTD块设备驱动程序

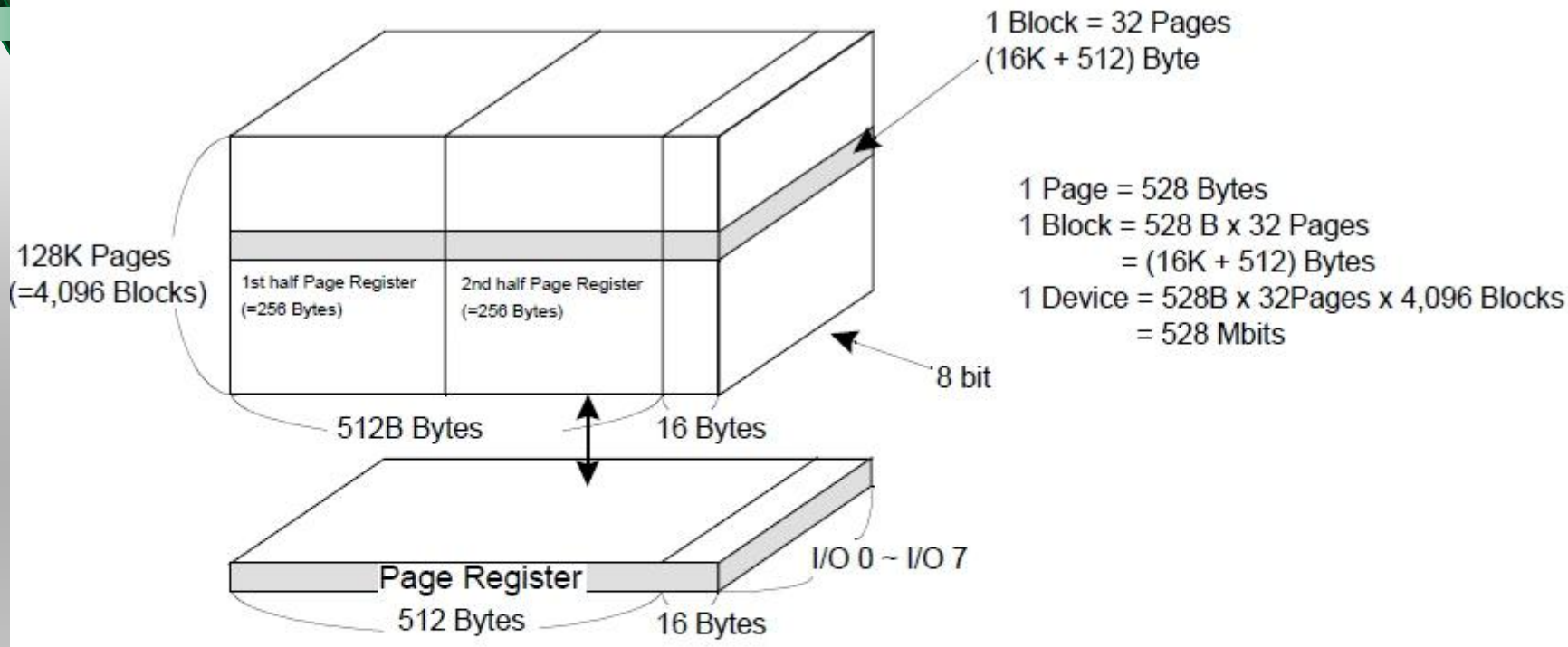
- MTD块设备驱动程序可以让flash器件伪装成块设备
- 实际上它通过把整块的erase block放到ram里面进行访问，然后再更新到flash
- 用户可以在这个块设备上创建通常的文件系统

MTD设备驱动程序框架

块设备	字符设备	FTL/NFTL
MTD 核心程序		
Nor Flash驱动	Nand Flash驱动	Doc存储卡驱动

FARSIGHT


目录，文件
用户层
虚拟文件系统
Jffs2
Cramfs
Fat
内核 文件系
MTD 字符设备
mtd_char.c
MTD 块设备
mtd_block.c
mtd_blkdevs.c
MMC 块设备
mmc_disk.c
mtd_info
设备驱动层
MTD 设备驱动
nandbase.c
MMC 设备驱动
mmc_bus.c
mmc_slot2410.c
nand_chip
MTD 设备驱动
s3c2410_nand.c
Nan Flash
MMC 卡
硬件



	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	**Column Address**
2nd Cycle	A9	A10	A11	A12	A13	A14	A15	A16	**Row Address (Page Address)**
3rd Cycle	A17	A18	A19	A20	A21	A22	A23	A24	
4th Cycle	A25	*L	*L	*L	*L	*L	*L	*L	

FARSIGHT

Table 1. Command Sets

Function	1st. Cycle	2nd. Cycle	3rd. Cycle	Acceptable Command during Busy
Read 1	00h/01h(1)	-	-	
Read 2	50h	-	-	
Read ID	90h	-	-	
Reset	FFh	-	-	O
Page Program (True)(2)	80h	10h	-	
Page Program (Dummy)(2)	80h	11h	-	
Copy-Back Program(True)(2)	00h	8Ah	10h	
Copy-Back Program(Dummy)(2)	03h	8Ah	11h	
Block Erase	60h	D0h	-	
Multi-Plane Block Erase	60h----60h	D0h	-	
Read Status	70h	-	-	O
Read Multi-Plane Status	71h(3)	-	-	O

Linux MTD系统接口

- 用于描述MTD原始设备的数据结构是mtd_info，这其中定义了大量关于MTD的数据和操作函数

```
struct mtd_info
{
        u_char type; //内存技术的类型
        u_int32_t flags; //标志位
        u_int32_t size; //mtd设备的大小
        u_int32_t erasesize; //主要的擦除块大小（同一个
                    //mtd设备可能有数种不同的erasesize）
        u_int32_t oobblock; // oob块大小
        u_int32_t oobsize; // oob数据大小
        u_int32_t ecctype; //ecc类型
        u_int32_t eccsize; //ecc工作的范围
…
}
```

JFFS2文件系统

- JFFS2文件系统是专门为flash设计的完整的文件系统，而不仅仅是把flash模拟成一个块设备
- JFFS2文件系统可以避免write after copy failure等可能造成数据破坏的现象
- JFFS2文件系统可以使擦除和编程操作均匀分布到所有block上，避免影响期间寿命

YAFFS文件系统

- YAFFS文件系统是为nand flash器件优化的文件系统
- 和jffs2对比，yaffs在nand flash上提供了更好的性能，包括
  - 小得多的内存消耗
  - 更快速的文件系统加载