```
        inode->i_count--;
        return;
    }
    // 如果是块设备上的 inode，则将 inode 的数据区同步到磁盘上并等待 inode 解锁
    if (S_ISBLK(inode->i_mode)) {
        sync_dev(inode->i_zone[0]);
        wait_on_inode(inode);
    }
repeat:
    // 如果 inode 引用计数大于 1，递减 inode 引用计数
    if (inode->i_count>1) {
        inode->i_count--;
        return;
    }
    // 如果 inode 链接数为 0，重置并释放 inode
    if (!inode->i_nlinks) {
        truncate(inode);
        free_inode(inode);
        return;
    }
    // 如果 inode 为脏，将 inode 信息写入缓冲区，等待缓冲区刷新时写入磁盘
    if (inode->i_dirt) {
        write_inode(inode);
        wait_on_inode(inode);
        goto repeat;
    }
    inode->i_count--;
    return;
}
```

5.5　read 函数原理

读/写函数通常可以通过 4 种方式来实现：基于管道类型的 inode、基于字符类型的 inode、基于块设备的 inode 和基于目录文件的 inode。一般基于块设备和基于目录文件的 inode 的读/写函数使用最多，因此我们以这两种类型的读/写函数展开分析。

5.5.1　sys_read 函数

该函数首先对用户传入的文件描述符、字节缓冲区和要写入的字节数进行判断，然后

验证存放的缓冲区的内存限制。通过获取到的文件 inode 判断读取的方式。

```
// 代码路径：linux-0.11\fs\read_write.c

int sys_read(unsigned int fd,char * buf,int count)
{
    struct file * file;
    struct m_inode * inode;

    // 如果传入的文件描述符大于一个进程可打开的最大进程数
    // 或用户传入要读取的字节数小于 0
    // 或当前进程根据文件描述符获取的文件为空
    // 返回错误码
    if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
        return -EINVAL;
    // 如果用户传入要读取的字节数小于 0，返回 0
    if (!count)
        return 0;
    // 根据用户传入的字节缓冲区和字节数，验证存放的缓冲区的内存限制
    verify_area(buf,count);
    // 获取当前进程的文件 inode
    inode = file->f_inode;
    // 如果 inode 是管道类型，判断文件是否可读，通过管道读取文件
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
    // 如果 inode 是字符文件，通过字符设备读取字符
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
    // 如果 inode 是块设备文件，通过块设备读取字节
    if (S_ISBLK(inode->i_mode))
        return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    // 如果 inode 是目录文件或常规文件
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        // 如果从文件的位置开始读取的字节数超出了文件大小，截取未超出部分的 count 值
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode,file,buf,count);
    }
    printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
    return -EINVAL;
}
```

5.5.2　block_read 函数

该函数用于读取块设备的字节。根据用户传入的字节数进行遍历，首先判断用户读取的数据是否只在一个块中，如果用户读取的数据只在一个块中，则仅读取一个块的字节。如果用户读取的数据不只在一个块中，则需要将多个块的字节放入用户传入的缓冲区中。

然后进行预读，读取当前缓冲块和后两个缓冲块，但是并不会读入到用户传入的缓冲区。

根据用户传入的字节数读取完成后，释放缓冲区，返回已读字节数。

```
// 代码路径：linux-0.11\fs\block_dev.c

int block_read(int dev, unsigned long * pos, char * buf, int count)
{
    // 根据文件位置计算出进行写的块号和写第一个字节距离 block 的偏移量 offset
    int block = *pos >> BLOCK_SIZE_BITS;
    int offset = *pos & (BLOCK_SIZE-1);
    int chars;
    int read = 0;
    struct buffer_head * bh;
    register char * p;

    while (count>0) {
        // 如果要读的字节数未超出一个块大小，读取 chars 大小的字节
        chars = BLOCK_SIZE-offset;
        if (chars > count)
            chars = count;
        // 预读当前块和后两个块的数据，如果失败，返回错误号
        if (!(bh = breada(dev,block,block+1,block+2,-1)))
            return read?read:-EIO;
        block++;
        // 根据获取缓冲区的数据区的首地址和偏移量得到要读取数据的起始位置
        p = offset + bh->b_data;
        offset = 0;
         // 每次移动已经读取的字节数
        *pos += chars;
         // 记录已经读取的字节数
        read += chars;
        // 记录未写字节数
        count -= chars;
        // 从缓冲区中读取数据的起始位置并将字符放入缓冲区
        while (chars-->0)
            put_fs_byte(*(p++),buf++);
```

```
        // 释放缓存区
        brelse(bh);
    }
    return read;
}
```

5.5.3 file_read 函数

该函数用于文件类型的读取数据。通过 inode 和文件所处位置获取对应的逻辑块号，通过设备号和逻辑块号获取对应的缓冲块，将数据区的内容赋值到缓冲区中，返回已读字数。

```
// 代码路径：linux-0.11\fs\file_dev.c

// 块的大小为 1024
#define BLOCK_SIZE 1024

int file_read(struct m_inode * inode,struct file * filp,char * buf,int count)
{
    int left,chars,nr;
    struct buffer_head * bh;

    // 如果要读取的字节数量小于或等于 0，返回 0
    if ((left=count)<=0)
        return 0;

    while (left) {
        // 根据 inode 和文件位置所处的块获取对应的逻辑块号
        if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) {
            if (!(bh=bread(inode->i_dev,nr)))
                break;
        } else
            bh = NULL;
        // 获取文件在缓冲块中的偏移值
        nr = filp->f_pos % BLOCK_SIZE;
        chars = MIN( BLOCK_SIZE-nr , left );
        filp->f_pos += chars;
        left -= chars;
        // 复制数据区中的数据到缓冲区中
        if (bh) {
            // p 指向数据区要读的起始地址
            char * p = nr + bh->b_data;
            while (chars-->0)
                put_fs_byte(*(p++),buf++);
```

```
            brelse(bh);
        } else {
            while (chars-->0)
                put_fs_byte(0,buf++);
        }
    }
    inode->i_atime = CURRENT_TIME;
    return (count-left)?(count-left):-ERROR;
}
```

5.5.4　bmap 函数

该函数用于创建 inode 的逻辑块。在 inode 的 i_zone 逻辑块数组中，前 7 项用于直接获取逻辑块号，第 7 项指向间接逻辑块数组获取逻辑块，第 8 项使用二次寻址的方式建立二级间接逻辑块数组来获取逻辑块。

```
// 代码路径：linux-0.11\fs\inode.c

int bmap(struct m_inode * inode,int block)
{
    return _bmap(inode,block,0);
}

static int _bmap(struct m_inode * inode,int block,int create)
{
    struct buffer_head * bh;
    int i;

    // 如果传入的块号超出范围，死机
    if (block<0)
        panic(" bmap: block<0");
    if (block >= 7+512+512*512)
        panic(" bmap: block>big");

    if (block<7) {
        // 如果 inode 的逻辑块数组为 0，将 inode 的逻辑块数组指向新建逻辑块，初始化
inode 并返回
        if (create && !inode->i_zone[block])
            if (inode->i_zone[block]=new_block(inode->i_dev)) {
                inode->i_ctime=CURRENT_TIME;
                inode->i_dirt=1;
            }
        return inode->i_zone[block];
```

```
    }

    // 如果逻辑块号大于或等于 7 且小于 512，表示该逻辑块是一级间接块，先将 inode 逻辑块
数组中第 7 项指向新建的逻辑块，初始化 inode
    block -= 7;
    if (block<512) {
        if (create && !inode->i_zone[7])
            if (inode->i_zone[7]=new_block(inode->i_dev)) {
                inode->i_dirt=1;
                inode->i_ctime=CURRENT_TIME;
            }
        // 校验 inode 逻辑块数组第 7 项是否存在、能否根据 inode 的设备号和块号获取对应
缓冲块
        if (!inode->i_zone[7])
            return 0;
        if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
            return 0;
        // 获取间接逻辑块数组第 block 项的逻辑块号
        i = ((unsigned short *) (bh->b_data))[block];
        // 如果逻辑块号为 0，将 i 指向新建逻辑块，设置间接逻辑块数组第 block 项的逻辑块
号为新逻辑块号
        if (create && !i)
            if (i=new_block(inode->i_dev)) {
                ((unsigned short *) (bh->b_data))[block]=i;
                bh->b_dirt=1;
            }
        brelse(bh);
        return i;
    }

     // 程序执行到这里，表示为二级间接块，先将 inode 逻辑块数组第 8 项指向新建的逻辑块，
初始化 inode
    block -= 512;
    if (create && !inode->i_zone[8])
        if (inode->i_zone[8]=new_block(inode->i_dev)) {
            inode->i_dirt=1;
            inode->i_ctime=CURRENT_TIME;
        }
    // 校验 inode 逻辑块数组第 8 项是否存在、能否根据 inode 的设备号和块号获取对应缓冲块
    if (!inode->i_zone[8])
        return 0;
    if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
        return 0;
    // 获取间接逻辑块数组第 block/512 项的逻辑块号
```

```
    i = ((unsigned short *)bh->b_data)[block>>9];
      // 如果逻辑块号为 0，将 i 指向新建逻辑块，设置间接逻辑块数组第 block/512 项的逻
辑块号为新逻辑块号
    if (create && !i)
        if (i=new_block(inode->i_dev)) {
            ((unsigned short *) (bh->b_data))[block>>9]=i;
            bh->b_dirt=1;
        }
    brelse(bh);
    // 校验逻辑块号是否存在、能否根据 inode 设备号和块号获取缓冲块
    if (!i)
        return 0;
    if (!(bh=bread(inode->i_dev,i)))
        return 0;
    // 再次创建新的逻辑块
    i = ((unsigned short *)bh->b_data)[block&511];
    if (create && !i)
        if (i=new_block(inode->i_dev)) {
            ((unsigned short *) (bh->b_data))[block&511]=i;
            bh->b_dirt=1;
        }
    brelse(bh);
    return i;
}
```

5.5.5　new_block 函数

该函数用于创建新的超级块。搜索逻辑块位图，找到一个空闲的逻辑块，将逻辑块对应的缓冲块数据清空后，初始化缓冲块并返回。

```
// 代码路径：linux-0.11\fs\bitmap.c

int new_block(int dev)
{
    struct buffer_head * bh;
    struct super_block * sb;
    int i,j;

    // 根据指定设备获取超级块，如果失败，死机
    if (!(sb = get_super(dev)))
        panic("trying to get new block from nonexistant device");

    j = 8192;
    // 搜索逻辑块位图，找到一个空闲的逻辑块
```

```
    for (i=0 ; i<8 ; i++)
        if (bh=sb->s_zmap[i])
            if ((j=find_first_zero(bh->b_data))<8192)
                break;
    // 如果超出逻辑块范围或根据逻辑块找到的缓冲块不存在，返回 0
    if (i>=8 || !bh || j>=8192)
        return 0;
    // 设置新的逻辑块的比特位，如果该比特位已存在，死机
    if (set_bit(j,bh->b_data))
        panic("new_block: bit already set");
    bh->b_dirt = 1;
    // 获取逻辑块号
    j += i*8192 + sb->s_firstdatazone-1;
    // 如果逻辑块号大于超级块中已有逻辑块的数量，返回 0
    if (j >= sb->s_nzones)
        return 0;
    // 根据设备号和块号获取对应缓冲块，如果失败，死机
    if (!(bh=getblk(dev,j)))
        panic("new_block: cannot get block");
    // 如果缓冲块引用计数不为 1，死机
    if (bh->b_count != 1)
        panic("new block: count is != 1");
    // 将缓存块的数据区清零，然后初始化缓冲块，释放缓冲区并返回逻辑块号
    clear_block(bh->b_data);
    bh->b_uptodate = 1;
    bh->b_dirt = 1;
    brelse(bh);
    return j;
}
```

5.5.6 get_super 函数

该函数用于获取指定设备的超级块。搜寻整个超级块数组，如果找到了指定设备的超级块就返回，如果超级块被别的设备使用了，就重新搜索超级块数组。

```
// 代码路径：linux-0.11\fs\super.c

// 定义超级块的最大数量为 8
#define NR_SUPER 8

struct super_block * get_super(int dev)
{
    struct super_block * s;
```

```
    // 如果传入设备不存在，返回空
    if (!dev)
        return NULL;
    // 获取超级块的起始地址
    s = 0+super_block;

    // 搜索整个超级块数组，找到指定设备的超级块并返回
    while (s < NR_SUPER+super_block)
        // 如果搜寻的超级块是传入设备的超级块，等待超级块解锁
        if (s->s_dev == dev) {
            wait_on_super(s);
            // 在超级块解锁的瞬间，可能有别的设备已经使用了该超级块，因此需要再次进行
判断。如果当前设备还是用户传入的设备，返回超级块，否则从超级块数组的起始处再次搜索
            if (s->s_dev == dev)
                return s;
            s = 0+super_block;
            // 如果搜寻的超级块不是传入设备的超级块，继续搜索下一个超级块
        } else
            s++;
    return NULL;
}
```

5.6　write 函数原理

当应用程序进行写操作时将会经历以下步骤：通过 fd 传入一个缓冲区，放入要读的数据，然后通过 fd 找到 file，通过 file 找到 inode，通过 inode 找到对应的缓冲块。如果缓冲块存在，将数据放入缓冲区；如果不存在，分配一个缓冲块到磁盘里读取，读取到缓冲块后，把缓冲块数据放入用户缓冲区，然后返回。

5.6.1　sys_write 函数

根据用户指定写入的文件、写入的字节和写入的字节数，调用 write()函数进行写操作。内核通过 fd 文件下标获取文件，再根据文件获取 inode，通过判断 inode 是管道文件、字节文件、设备文件还是普通文件，执行不同的写入逻辑。

```
// 代码路径：linux-0.11\fs\read_write.c

int sys_write(unsigned int fd,char * buf,int count)
{
```

```
    struct file * file;
    struct m_inode * inode;

    // 如果 fd 文件下标超出进程可打开的最大文件数，或指定写入字节数小于 0，或当前进程指向的文件为空，返回错误号
    if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
        return -EINVAL;
    if (!count)
        return 0;
    // 获取文件的 inode
    inode=file->f_inode;
    if (inode->i_pipe)
        return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
    // 如果是块设备，根据 inode 的数据区、文件的偏移量、缓冲区和写入字节数进行块设备写操作
    if (S_ISBLK(inode->i_mode))
        return block_write(inode->i_zone[0],&file->f_pos,buf,count);
    if (S_ISREG(inode->i_mode))
        return file_write(inode,file,buf,count);
    printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
    return -EINVAL;
}
```

5.6.2 block_write 函数

该函数用于将用户传入的字节以块为单位写入缓冲区中，如图 5.10 所示。需要注意的是，字节写入缓冲区后，并不会立即同步到磁盘上，而是等待内核调用缓冲区同步函数 sync_dev 后，才会刷新缓冲区中的数据并同步到磁盘。

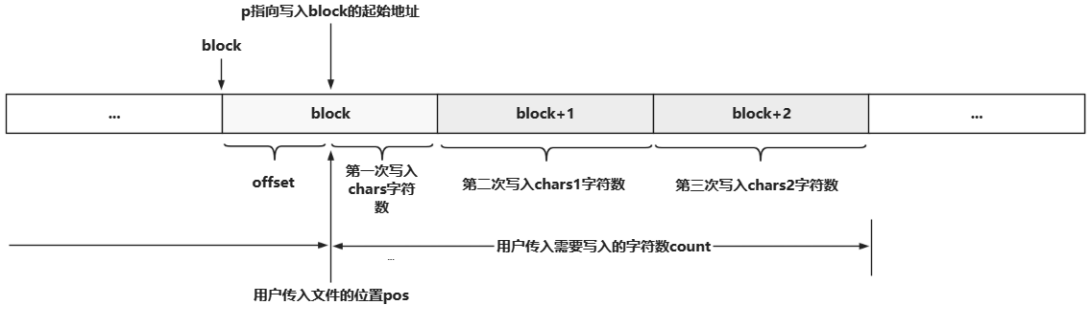

图 5.10　图解字节写入缓冲区

该函数首先需要根据文件的位置 pos 确定要写入的块号，找到要写入的块后，还需要根据 pos 确定距离写入块的起始地址的偏移量 offset。有了块的起始地址和偏移量，就可以确定当前写入 block 的地址。

然后对用户传入的字节进行分析，如果一个块的大小可以容纳用户传入的字节，那么调用 getblk 获取一个块的缓冲区即可。如果一个块的大小不够容纳用户传入的字节，那么为了提升性能，采取 breada 预读的方式，利用空间转换时间的思想，一次性多读两块进行填写。

在写入完成后，因为修改了缓冲区的数据，所以需要将当前已用缓冲区置脏，然后释放缓冲区。最后返回记录的已写字节数。

```
// 代码路径：linux-0.11\fs\block_dev.c

int block_write(int dev, long * pos, char * buf, int count)
{
    // 根据文件位置计算出进行写的块号和写第一个字节距离 block 的偏移量 offset
    int block = *pos >> BLOCK_SIZE_BITS;
    int offset = *pos & (BLOCK_SIZE-1);
    int chars;
    int written = 0;
    struct buffer_head * bh;
    register char * p;

    // 循环执行写入操作，直到要写入的字节数为 0
    while (count>0) {
        // 计算一个块可容纳的字节数
        chars = BLOCK_SIZE - offset;
        // 一个块可容纳的字节数大于用户传入的字节数，则将块容纳字节数设置为用户传入字
节数
        if (chars > count)
            chars=count;
        // 如果用户传入的字节数正好是一个块大小，根据设备号和块读取对应一个块的缓冲区
        if (chars == BLOCK_SIZE)
            bh = getblk(dev,block);
        // 否则预读当前块和后两个块对应缓冲区，然后将块号+1
        else
            bh = breada(dev,block,block+1,block+2,-1);
        block++;
        // 如果获取缓冲区失败，返回已经写入的字节数，如果写入字节数为 0，返回错误号
        if (!bh)
            return written?written:-EIO;
        // 根据缓冲区的数据区起始地址和偏移量，获取要写入的地址
        p = offset + bh->b_data;
```

```
        offset = 0;
        // 每次移动已写字节数
        *pos += chars;
        // 记录已经写入的字节数
        written += chars;
        // 记录未写字节数
        count -= chars;
        // 将用户传入的字节 buf 写入 p 指向的缓冲区
        while (chars-->0)
            *(p++) = get_fs_byte(buf++);
        // 置缓冲区为脏并释放缓冲区
        bh->b_dirt = 1;
        brelse(bh);
    }
    // 返回已写字节数
    return written;
}
```

5.6.3 file_write 函数

该函数用于文件类型的写入数据。通过 inode 和文件所处位置获取对应的逻辑块号，通过设备号和逻辑块号获取对应的缓冲块，将用户传入的字节缓冲区内容复制到高速缓冲区中，返回已写字节数。

```
// 代码路径：linux-0.11\fs\file_dev.c

int file_write(struct m_inode * inode,struct file * filp,char * buf,int count)
{
    off_t pos;
    int block,c;
    struct buffer_head * bh;
    char * p;
    int i=0;

// 如果文件的访问标志为 O_APPEND，即向文件末尾添加数据，那么将写入位置移到 inode 尾部。否则，获取文件位置写入
    if (filp->f_flags & O_APPEND)
        pos = inode->i_size;
    else
        pos = filp->f_pos;
    // 循环写入
    while (i<count) {
        // 如果创建逻辑块失败或获取缓冲块失败，退出写操作
```

```
        if (!(block = create_block(inode,pos/BLOCK_SIZE)))
            break;
        if (!(bh=bread(inode->i_dev,block)))
            break;
        // 获取要写入数据块的偏移值和起始位置
        c = pos % BLOCK_SIZE;
        p = c + bh->b_data;
        bh->b_dirt = 1;
        c = BLOCK_SIZE-c;
        if (c > count-i) c = count-i;
        pos += c;
        if (pos > inode->i_size) {
            inode->i_size = pos;
            inode->i_dirt = 1;
        }
        i += c;
        // 将用户传入的字节缓冲区复制到高速缓冲区
        while (c-->0)
            *(p++) = get_fs_byte(buf++);
        brelse(bh);
    }
    inode->i_mtime = CURRENT_TIME;
    if (!(filp->f_flags & O_APPEND)) {
        filp->f_pos = pos;
        inode->i_ctime = CURRENT_TIME;
    }
    return (i?i:-1);
}
```

5.7　高速缓冲区

如果我们想要计算机修改磁盘中的数据，那么 CPU 需要先将磁盘里的文件数据读入内存中才能进行修改操作。在没有缓存的情况下，CPU 的每次读写操作都需要经历从磁盘到内存（或从内存到磁盘）的过程。众所周知，越靠近 CPU 的硬件存取速度越快。对于单体主机而言，在不考虑网络的情况下，磁盘是存取速度最慢的。在 I/O 密集型的操作下，大量的读写在内存与磁盘中循环往复，极易出现同一文件刚从磁盘读入内存进行修改，放入磁盘的不久后，又需要被再次修改。这会使系统响应的时间加长，吞吐率降低。因此，内核通过一个高速缓冲区（buffer cache）来减小系统对磁盘的存取频率。系统将最近被使用过的磁盘里的数据块放入高速缓冲区中，每次批量读取磁盘里用户可能用到的数据并修

改完成后，不直接放入磁盘，而是等待一批数据完成修改后，再刷入磁盘中。

当系统从磁盘中读取数据时，首先尝试从高速缓冲区中读取，如果高速缓冲区中已经存在想要的数据，那么内核就不用再去磁盘上读取了。如果高速缓冲区中不存在想要的数据，此时内核再从磁盘中读取数据并将其放入高速缓冲区。

同理，内核要往磁盘上写入的数据也被暂时存放在高速缓冲区中，便于内核随后又试图读取它时，能迅速在高速缓冲区中找到它。内核也会通过判断数据是否很快会被修改或数据是否真的需要被写入磁盘，来减少磁盘写入操作的频率。内核通过让高速缓冲区预读或延迟写的方式提升性能。

一个缓冲区由两部分组成：缓冲头（buffer head）和数据区。缓冲头用来标识缓冲区，而数据区则用来存储磁盘上的数据。由于缓冲头与数据有一一对应的映射关系，因此通常说的缓冲区就是这两部分的统称。

一个缓冲区中的数据与一个逻辑块中的数据相对应，内核通过缓冲头中的标识字段来识别缓冲区中的内容。

系统将逻辑块中的内容映射到缓冲区，但缓冲区里存储的数据仅仅是临时的，经过一段时间后，内核就会将另一个逻辑块中的内容映射到该缓冲区，因此有限的缓冲区是会被无限次复用的。

5.7.1 buffer_head 结构体

缓冲区存在于内存和磁盘之间，它存在的意义是为了提升 I/O 性能。通过缓冲区来暂存数据，避免程序重复修改或读取文件时频繁访问磁盘。等待调用磁盘同步函数时，再将缓冲区中的数据写入磁盘。

缓冲区与磁盘之间存在映射关系，因为内核代码里的磁盘有磁盘块，所以为了与磁盘块对应，缓冲区也被划分为一个个的缓冲块。在内核中，用 b_dev 来表示磁盘的设备号，用 b_blocknr 来表示磁盘块号，通过设备号和块号来确定对应缓冲块。读者在后面会发现，在如 bread()、getblk()、get_hash_table()等函数中，都会要求传入 dev 设备号和 block 块号来确定唯一缓冲块。逆向亦是如此，只要将传入的 dev 和 block 转换为 buffer_head 缓冲头的 b_dev 和 b_blocknr 即可确定唯一磁盘块。

由于缓冲区是共享资源，因此同一个缓冲块可能被多个进程共享使用，为了避免缓冲区中数据被篡改的风险，引入了 b_uptodate 和 b_dirt 属性。

- ☑ b_uptodate 属性会在新建缓冲块时被置为 1，程序通过 b_uptodate 来判断当前缓冲块是否为最新的，如果缓冲块是最新的，就可以放心使用。当 b_uptodate 属性

为 0 时，则表明缓冲块中的数据并不是磁盘上的最新数据，如果进行读写的话，可能会引发错误。

- ☑ b_dirt 属性用于表示缓冲块是否为脏，脏的含义为缓冲块被其他进程修改过。因此程序在执行代码时，会根据 b_dirt 属性来判断是否需要将缓冲块的数据同步到磁盘。

脏的概念并非缓冲区独有，共享资源如文件和超级块，也会有标识是否为脏的属性。在文件 m_inode 结构体中，使用 i_dirt 表示文件是否为脏；在超级块 super_block 结构体中，使用 s_dirt 表示超级块是否为脏。由于数据的读写都需要经过缓冲区，因此文件和超级块中没有再提供 uptodate 属性。

另外，磁盘只有磁道、扇区和柱面的概念，没有磁盘块。块是内核独有的，是为了方便数据管理而产生的，正因为磁盘中没有磁盘块，仅在逻辑上存在，所以称之为逻辑块。因此，当数据从缓冲块写入磁盘时，还需要通过逻辑块号换算成磁盘中的磁盘、扇区和柱面。

对于缓冲头结构而言，位于高端地址的属性相对好理解，位于低端地址的两个双向链表则需要读者仔细琢磨理解。在缓冲区中存在两个双向链表，一个是数组加链表实现的 hash 队列，另一个则是 free_list 空闲链表。

hash 队列的作用是通过设备号和块号来锁定唯一缓冲块。而空闲链表的作用是在调用 getblk()函数的时候进行遍历查找，获取可用缓冲块。

这两个链表在后续代码中都会遇到，对于使用方法也要反复研读才能理解。

```
// 文件路径：linux-0.11\include\linux\fs.h

// 缓冲头
struct buffer_head {
    char * b_data;                          // 指向数据区的指针（1024B）
    unsigned long b_blocknr;                // 逻辑块号
    unsigned short b_dev;                   // 逻辑块设备号
    unsigned char b_uptodate;               // 缓冲块是否更新
    unsigned char b_dirt;                   // 缓冲块是否为脏（0 为脏块，1 为干净块）
    unsigned char b_count;                  // 缓冲区引用计数
    unsigned char b_lock;                   // 缓冲区是否上锁（0 为无锁，1 为上锁）
    // 缓冲区双向链表
    struct task_struct * b_wait;            // 在缓冲区中等待释放的进程
    struct buffer_head * b_prev;            // hash 表的前驱节点
    struct buffer_head * b_next;            // hash 表的后继节点
    struct buffer_head * b_prev_free;       // 缓冲区空闲链表的前驱节点
    struct buffer_head * b_next_free;       // 缓冲区空闲链表的后继节点
};
```

5.7.2 bread 函数

该函数根据指定的设备号和块号读取缓冲区。

```
// 文件路径：linux-0.11\fs\buffer.c

struct buffer_head * bread(int dev,int block)
{
    struct buffer_head * bh;
// 获取一个可用的缓冲块，如果失败，死机
    if (!(bh=getblk(dev,block)))
        panic("bread: getblk returned NULL\n");
    // 如果缓冲块已更新，返回缓冲块
    if (bh->b_uptodate)
        return bh;
    // 调用低等级读写块函数
    ll_rw_block(READ,bh);
    // 等待缓冲块解锁
    wait_on_buffer(bh);
    // 如果缓冲块已更新，返回缓冲块
    if (bh->b_uptodate)
        return bh;
    // 如果程序执行到这里，表明读取缓冲块失败，释放缓冲块，返回空
    brelse(bh);
    return NULL;
}
```

5.7.3 breada 函数

该函数根据传入的设备号和可变参数列表的块号来预读 n 个逻辑块上的内容，将读取内容放入缓冲块中并返回。

```
// 文件路径：linux-0.11\fs\buffer.c

struct buffer_head * breada(int dev,int first, ...)
{
    va_list args;
    struct buffer_head * bh, *tmp;

    // 获取传入的可变参数列表中的第一个参数，也就是块号
    va_start(args,first);
```

```
    // 根据设备号和块号读取对应缓冲块，如果失败，死机
    if (!(bh=getblk(dev,first)))
        panic("bread: getblk returned NULL\n");
    // 如果缓冲块未更新，调用低等级读写块函数，将磁盘数据读入缓冲区
    if (!bh->b_uptodate)
        ll_rw_block(READ,bh);
    // 遍历获取可变参数列表中的其他块号，依次获取 tmp 缓冲区
    while ((first=va_arg(args,int))>=0) {
        tmp=getblk(dev,first);
        // 如果缓冲块存在，但是未更新，调用低等级读写块函数，将磁盘数据读入缓冲区。将
缓冲区的引用计数-1
        if (tmp) {
            if (!tmp->b_uptodate)
                ll_rw_block(READA,bh); // 这里传入的应该是 tmp
            tmp->b_count--;
        }
    }
    va_end(args);
    // 等待缓冲块解锁，如果缓冲块已更新，返回缓冲块。否则释放缓冲块并返回空
    wait_on_buffer(bh);
    if (bh->b_uptodate)
        return bh;
    brelse(bh);
    return (NULL);
}
```

5.7.4　brelse 函数

该函数用于释放缓冲块。首先判断传入的缓冲块是否存在，然后等待缓冲块解锁，将缓冲块的引用计数-1 后，唤醒等待在空闲链表上的进程。

```
// 文件路径：linux-0.11\fs\buffer.c

void brelse(struct buffer_head * buf)
{
    if (!buf)
        return;
    wait_on_buffer(buf);
    if (!(buf->b_count--))
        panic("Trying to free free buffer");
    wake_up(&buffer_wait);
}
```

5.7.5 getblk 函数

此函数是缓冲区文件 buffer.c 中最重要的函数，核心逻辑都在此函数中。该函数用于获取缓冲块，首先尝试根据设备号和块号从缓冲区的哈希表中找到一个缓冲块，如果没有找到可用的缓冲区，再尝试从空闲链表中找到一个合适的缓冲块并返回。

```
// 代码路径：linux-0.11\fs\buffer.c

// 定义 BADNESS 宏，用于判断当前缓冲区是否为脏和上锁。缓冲区脏为 1，不脏为 0。缓冲区上锁为 1，未上锁为 0。因此 BADNESS 末两位排列组合可以表示 4 种状态
// 缓冲区不脏，未上锁：00
// 缓冲区脏，未上锁：10
// 缓冲区不脏，上锁：01
// 缓冲区脏，上锁： 11
// 由此可发现，缓冲区为脏的权重大于上锁的权重
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
struct buffer_head * getblk(int dev,int block)
{
    struct buffer_head * tmp, * bh;

repeat:
    // 根据设备号和块搜索哈希表，如果从哈希表中找到缓冲块，说明此缓冲块已经在使用，返回缓冲块指针
    if (bh = get_hash_table(dev,block))
        return bh;
    // 定义一个临时变量指向缓冲区链表的首地址，这里定义 free_list 是为了从缓冲区链表中找到一个空闲的缓冲块
    tmp = free_list;
    // 要找一个不为脏、未上锁并且未被引用的空闲缓冲区
    do {
        // 如果缓冲区链表的引用计数不为 0，说明当前缓冲块正在被使用，继续从链表找未使用的缓冲块
        if (tmp->b_count)
            continue;
        // 如果缓冲区为空，或者获取到的缓冲块权重小于上一个缓冲块的权重，说明获取到的缓冲块要么不脏，要么未上锁
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
            // 将 bh 指向获取到的缓冲块地址
            bh = tmp;
            // 如果判断缓冲块不脏且未上锁（即状态为 00），说明找到了可用的空闲缓冲块，退出
            if (!BADNESS(tmp))
                break;
        }
```

```
// 遍历整个缓冲区的空闲链表
    } while ((tmp = tmp->b_next_free) != free_list);
    // 遍历空闲链表，找到一个可用的缓冲块
    // 1：空闲链表为空，此时需要当前进程阻塞，所以调用 sleep_on()函数阻塞当前进程
    // 2：空闲链表不为空
    // 2.1：空闲链表中存在不脏的缓冲块（也即没有缓存写入的数据和没有被锁定的数据），
那么直接调用
    // 2.2：空闲链表中存在脏的缓冲块，那么找到一个不是特别脏的缓冲块（要么被锁定
b_block，要么是被写过数据 b_dirt）
    if (!bh) {
        // 如果缓冲块为空，将当前进程置为不可中断的等待状态，睡眠在缓冲区的等待队列上
        sleep_on(&buffer_wait);
        goto repeat;
    }
    // 使用原子性指令，等待缓冲块解锁。将等待队列上的进程置为睡眠状态
    wait_on_buffer(bh);
    // 如果在执行 wait_on_buffer()函数时，开中断的一瞬间有新的进程使用了此缓冲块，
则重新查找缓冲块
    if (bh->b_count)
        goto repeat;
    // 当找到的缓冲块为脏时，将设备所属的缓冲块中的数据同步到磁盘，然后等待同步完毕
    while (bh->b_dirt) {
        sync_dev(bh->b_dev);
        wait_on_buffer(bh);
         // 如果在执行 wait_on_buffer()函数时，开中断的一瞬间有新的进程使用了此缓冲
块，则重新查找缓冲块
        if (bh->b_count)
            goto repeat;
    }
// 当进程为了等待缓冲块而睡眠时，其他进程也可能会将此块加入缓冲块，因此要根据设备号和
块号检查缓冲区的 hash 表，确定此块是否已经被加入缓冲区，如果是，则需要重新查找缓冲区
    if (find_buffer(dev,block))
        goto repeat;
    // 当程序执行到这里时，就已经找到了合适的缓冲区，这个缓冲区未被使用、未上锁、未被
修改
    // 初始化缓冲块
    bh->b_count=1;
    bh->b_dirt=0;
    bh->b_uptodate=0;
 // 将缓冲块从空闲链表中移出，绑定缓冲块对应的设备号和块号后，再插入空闲链表
    remove_from_queues(bh);
    bh->b_dev=dev;
    bh->b_blocknr=block;
    insert_into_queues(bh);
    return bh;
}
```

5.7.6 get_hash_table 函数

该函数根据设备号和块号，从缓冲区的 hash 表中找到一个对应的缓冲块并返回。其实在调用 find_buffer()函数时，就已经确定了唯一缓冲块，但是这里又封装了一次 get_hash_table()函数，其目的在于缓冲区是公共资源，除了进程在读取缓冲块时会上锁，在未上锁的情况下，如果有别的进程使用了该缓冲块，并修改了其对应的设备号和块号，那么这个缓冲块是无法使用的，为了以防万一，需要再次判断缓冲块的设备号和块号是否与当前进程传入的值相同。

```
// 文件路径：linux-0.11\fs\buffer.c

struct buffer_head * get_hash_table(int dev, int block)
{
    struct buffer_head * bh;

    for (;;) {
        // 根据设备号和块号找到对应缓冲块
        if (!(bh=find_buffer(dev,block)))
            return NULL;
        // 缓冲块的引用计数+1
        bh->b_count++;
        // 等待缓冲块解锁
        wait_on_buffer(bh);
        // 如果缓冲块的设备号和块号与传入的值相对应，表示找到了缓冲块，返回
        if (bh->b_dev == dev && bh->b_blocknr == block)
            return bh;
        // 否则引用计数-1
        bh->b_count--;
    }
}
```

5.7.7 wait_on_buffer 函数

该函数通过 CPU 提供的 cli 指令清除 eflags 寄存器的中断标志位进行开中断，当执行 cli 指令时，处理器会忽略可屏蔽的外部响应中断，此时不再接收程序中断响应，保证此函数执行的原子性。

然后一直等待缓冲区的锁被释放后，将缓冲区中等待的进程置为睡眠状态，再关中断。

当缓冲区未上锁时，此函数仅相当于开一次中断，再关一次中断，无实际意义。

```
// 文件路径：linux-0.11\fs\buffer.c
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

5.7.8　sync_dev 函数

该函数用于将传入设备的缓冲块数据同步到磁盘上。

```
// 文件路径：linux-0.11\fs\buffer.c
int sync_dev(int dev)
{
    int i;
    struct buffer_head * bh;

    bh = start_buffer;
    // 遍历所有的缓冲块，找到传入设备对应的缓冲块
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        // 如果遍历到的缓冲块对应的设备号不是传入的设备号，继续查找下一个缓冲块
        if (bh->b_dev != dev)
            continue;
        // 等待缓冲块解除锁定
        wait_on_buffer(bh);
        // 如果缓冲块的设备号是传入的设备号，并且缓冲块为脏，调用 ll_rw_block 函数，
将缓冲块数据写入磁盘
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    // 将文件的信息也写入磁盘
    sync_inodes();

    ...
    return 0;
}
```

5.7.9　find_buffer 函数

该函数根据指定设备号和块号，在缓冲区中找到对应的缓冲块并返回。

该函数证实了内核通过设备号和逻辑块号绑定唯一缓冲块。

缓冲区通过数组加链表的形式实现。首先根据设备号和逻辑块号计算出 hash 表项（横向确定坐标），再从缓冲区中查找对应的缓冲块（纵向确定坐标），直到缓冲区中某个缓冲块的设备号和块号与用户传入的值相同，则表示找到了对应的缓冲块，如图 5.11 所示。

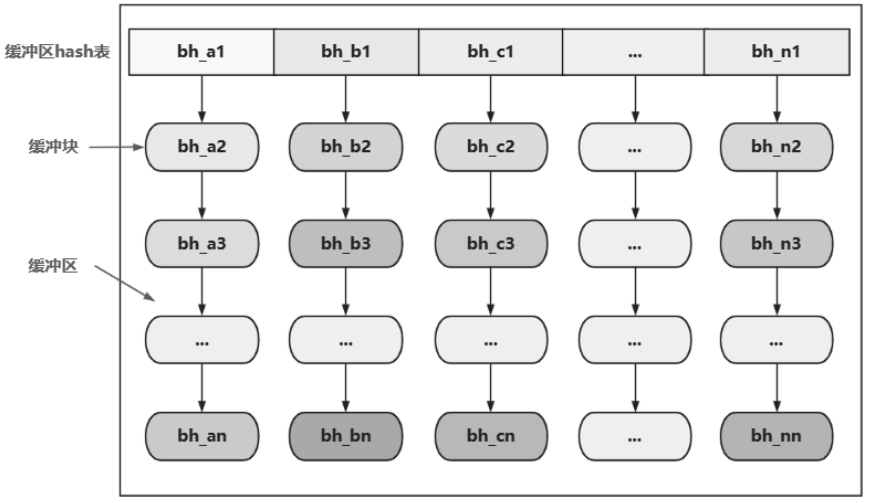

图 5.11　缓冲区 hash 结构

```
// 文件路径：linux-0.11\fs\buffer.c

// 定义 hash 表共有 307 项
#define NR_HASH 307
// 定义 hash 表（hash 数组）
struct buffer_head * hash_table[NR_HASH];
// 将 dev 与 block 进行异或操作（异或表示无进位相加），再跟定义的 NR_HASH 取模，计算出 hash 表的数组下标
#define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
// 通过 hash 表的数组下标，确定 hash 表项
#define hash(dev,block) hash_table[_hashfn(dev,block)]

static struct buffer_head * find_buffer(int dev, int block)
{
    struct buffer_head * tmp;
// 将设备号和逻辑块号计算的 hash 值作为起始值，在缓冲区中一直查找，直到找到对应的缓冲块并返回。如果没找到则返回空
    for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
        if (tmp->b_dev==dev && tmp->b_blocknr==block)
            return tmp;
    return NULL;
}
```

5.7.10　remove_from_queues 函数

该函数用于从 hash 队列和空闲链表中移除缓冲块。

不管是 hash 队列还是空闲链表，移除缓冲块的操作中都是一样的，需要区分的情况是移除缓冲块是否为头节点，如图 5.12 所示。0.11 版本的链表还是写的略显简陋，既没有将移除缓冲块释放，也没有将移除缓冲块断开连接。

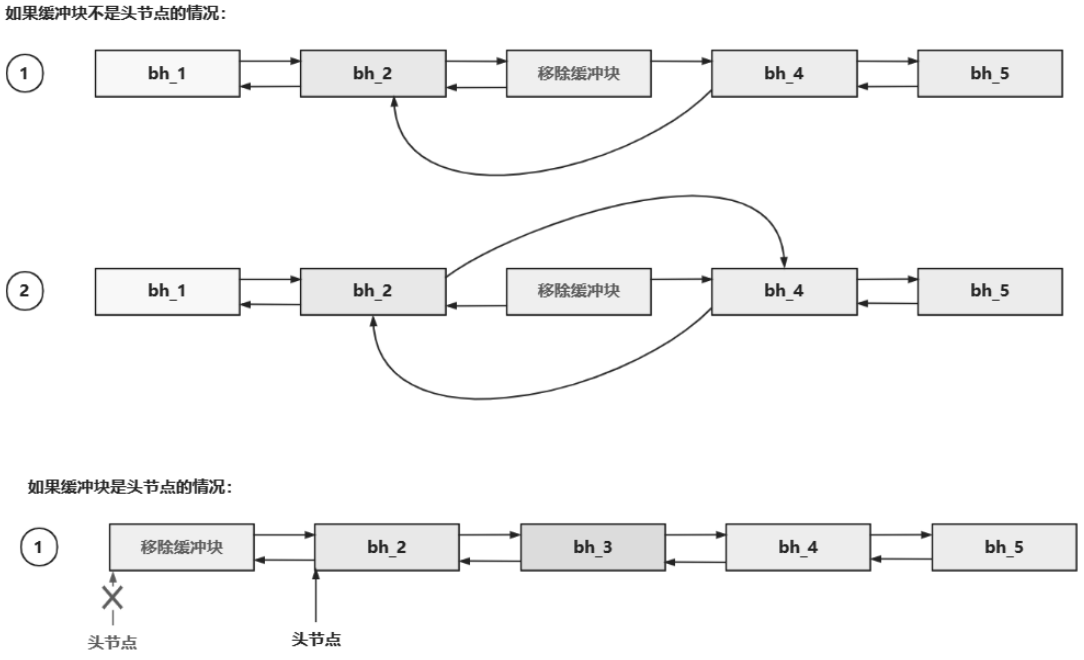

图 5.12　队列移除缓冲块的过程

```
// 文件路径：linux-0.11\fs\buffer.c

static inline void remove_from_queues(struct buffer_head * bh)
{
    // 从 hash 队列中移除缓冲块
// 如果 hash 队列中的缓冲块有后继节点，将缓冲块的后继节点的前驱节点连接到当前缓冲块的前驱节点上
    if (bh->b_next)
        bh->b_next->b_prev = bh->b_prev;
    // 如果 hash 队列中的缓冲块有前驱节点，将缓冲块前驱节点的后继节点连接到当前缓冲块的后继节点上
    if (bh->b_prev)
```

```
        bh->b_prev->b_next = bh->b_next;
    // 根据设备号和块号计算 hash，如果当前缓冲块是 hash 队列的头节点，则将其下一个缓冲块设置为头节点
    if (hash(bh->b_dev,bh->b_blocknr) == bh)
        hash(bh->b_dev,bh->b_blocknr) = bh->b_next;

    // 从空闲链表中移除缓冲块
    // 由于空闲链表是双向循环链表，如果空闲链表中的缓冲块没有前驱节点或后继节点，表示空闲链表错误，死机
    // 因为空闲链表是双向循环链表，所以不判断移除缓冲块是否为头节点
    if (!(bh->b_prev_free) || !(bh->b_next_free))
        panic("Free block list corrupted");
    // 将当前缓冲块前驱节点的后继节点连接到当前缓冲块的后继节点上
    bh->b_prev_free->b_next_free = bh->b_next_free;
    // 将当前缓冲块后继节点的前驱节点连接到当前缓冲块的前驱节点上
    bh->b_next_free->b_prev_free = bh->b_prev_free;
    // 如果当前缓冲块是空闲链表的头节点，则将其下一个缓冲块设置为头节点
    if (free_list == bh)
        free_list = bh->b_next_free;
}
```

5.7.11 insert_into_queues 函数

该函数用于将缓冲块插入空闲链表的尾端，并放入 hash 队列中，如图 5.13 所示。

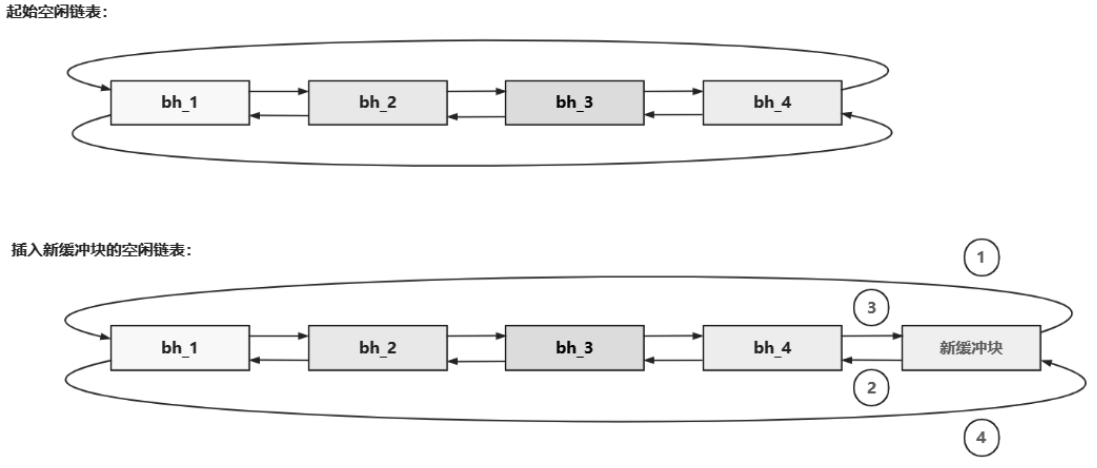

图 5.13　插入队列过程

在 remove_from_queues()函数中，我们以 hash 队列为例，展示了从 hash 队列中删除缓

冲块的图示。与数组加链表来实现的 hash 队列不同的是，空闲链表采用的是双向循环链表，将链表的头尾进行相连形成一个环。双向循环链表的优点在于，从任何一个节点出发跑一圈都可以回到起点，遍历完所有在循环链表上的节点，不需要从头或从尾开始遍历。这种形式更加适合空闲链表中以任意缓冲块出发，查找一个合适的缓冲块。

```
// 文件路径：linux-0.11\fs\buffer.c

static inline void insert_into_queues(struct buffer_head * bh)
{
// 插入空闲链表的尾端
    // 将新缓冲块后继节点连接到空闲链表的头节点
    bh->b_next_free = free_list;
    // 将新缓冲块的前驱节点连接到空闲链表头节点的前驱节点
    bh->b_prev_free = free_list->b_prev_free;
    // 将空闲链表头节点的前驱节点的后继节点连接到新缓冲块
    free_list->b_prev_free->b_next_free = bh;
    // 将空闲链表的前驱节点连接到新缓冲块
    free_list->b_prev_free = bh;

// 如果缓冲块有对应的设备号，那么放入新的 hash 队列中
    // 将缓冲块的前驱节点和后继节点置空
    bh->b_prev = NULL;
    bh->b_next = NULL;
    // 如果缓冲块没有对应的设备号，返回
    if (!bh->b_dev)
        return;
    // 将新缓冲块的后继节点连接到 hash 队列
    bh->b_next = hash(bh->b_dev,bh->b_blocknr);
    // 将 hash 队列的头节点设置为新缓冲块
    hash(bh->b_dev,bh->b_blocknr) = bh;
    // 将新缓冲块的后继节点的前驱节点连接到新缓冲块
    bh->b_next->b_prev = bh;
}
```

5.8　块设备驱动

块设备驱动是操作系统中的一个组件，用于管理和控制块设备的访问和操作。块设备是以固定大小的块（通常是几千字节）为单位进行读写的设备，如硬盘、固态硬盘（SSD）或闪存设备等。

5.8.1 块设备定义

blk 头文件用于定义块在不同设备上的定义。之前提到过，块是内核的概念，如果要将块中的数据读写到设备上，还需要将块转换成不同设备的磁道、扇区、柱面等信息。

```
// 代码路径：linux-0.11\kernel\blk_drv\blk.h

#define NR_BLK_DEV  7                    // 块设备的数量

// 系统所含设备的定义
  0 - unused (nodev)                     // 未使用
  1 - /dev/mem                           // 内存设备
  2 - /dev/fd                            // 软盘设备
  3 - /dev/hd                            // 硬盘设备
  4 - /dev/ttyx                          // ttyx 串行终端设备
  5 - /dev/tty                           // tty 终端设备
  6 - /dev/lp                            // lp 打印设备
  7 - unnamed pipes                      // 未命名的管道



// NR_REQUEST 用于表示 request-queue 请求队列的项数，写操作会使用到请求队列中 2/3 部分的请求项，并且读操作会被优先处理。32 项是一个合理的数组项，这样可以从电梯算法中获得最大的效率，当请求进入队列的时候，不会锁住大量的缓冲区。而如果设定为 64 项，就显得太大了（当大量的读操作或同步操作被执行时，很容易导致长时间的暂停）
#define NR_REQUEST  32

// request 结构体是一种扩展形式，因此我们从分页请求中使用相同的 request 结构。在分页过程中，'bh'是 NULL，而 waiting 用于读操作或写操作的实现
struct request {
    int dev;                             // 设备号
    int cmd;                             // READ 或 WRITE 命令
    int errors;                          // 产生错误的次数
    unsigned long sector;                // 起始扇区（1 个缓冲块 = 2 个扇区）
    unsigned long nr_sectors;            // 读或写的扇区数
    char * buffer;                       // 读或写的缓冲区
    struct task_struct * waiting;        // 等待执行的进程
    struct buffer_head * bh;             // 缓冲头
    struct request * next;               // 下一个请求项
};

// 块设备结构
```

```
struct blk_dev_struct {
    void (*request_fn)(void);                    // 指向处理请求的函数
    struct request * current_request;            // 指向当前请求
};

#ifdef MAJOR_NR                                  // 主设备号

// 需要时加入以下项，当前的块设备仅支持硬盘和软盘
// 根据不同的 MAJOR_NR 进行宏定义。包括定义设备名称、设备请求函数、设备数量、开启设备和关闭设备
// 如果 MAJOR_NR 主设备号为 1，表示的是 RAM 盘，进行 RAM 盘的宏定义
#if (MAJOR_NR == 1)
#define DEVICE_NAME "ramdisk"
#define DEVICE_REQUEST do_rd_request
#define DEVICE_NR(device) ((device) & 7)
#define DEVICE_ON(device)
#define DEVICE_OFF(device)

// 如果 MAJOR_NR 主设备号为 2，表示的是软盘，进行软盘的宏定义
#elif (MAJOR_NR == 2)
#define DEVICE_NAME "floppy"
#define DEVICE_INTR do_floppy
#define DEVICE_REQUEST do_fd_request
#define DEVICE_NR(device) ((device) & 3)
#define DEVICE_ON(device) floppy_on(DEVICE_NR(device))
#define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))

// 如果 MAJOR_NR 主设备号为 3，表示的是硬盘，进行硬盘的宏定义
#elif (MAJOR_NR == 3)
#define DEVICE_NAME "harddisk"
#define DEVICE_INTR do_hd
#define DEVICE_REQUEST do_hd_request
#define DEVICE_NR(device) (MINOR(device)/5)
#define DEVICE_ON(device)
#define DEVICE_OFF(device)
```

5.8.2　ll_rw_block 函数

该函数是缓冲区数据向硬盘发起读写请求的起始处，ll_rw_block 表示低等级读写块设备，这里面仅调用了 make_request()函数，一旦进入 make_request()函数，就到了硬件设备层。

```
// 宏定义主设备号，将传入的设备号右移，取设备号的高端地址
#define MAJOR(a) (((unsigned)(a))>>8)
```

```
// 宏定义次设备号，截取传入设备号的低 8 位，取设备号的低端地址
#define MINOR(a) ((a)&0xff)

// 请求结构体中包含所有加载扇区数据到内存中的信息
// 定义请求数组，包含 32 个请求项，后面称作请求队列
struct request request[NR_REQUEST];

// 当请求队列中没有空闲的请求项可以放入时，使用 wait_for_request 等待队列存放等待的请求项
struct task_struct * wait_for_request = NULL;

// blk_dev_struct 块设备结构中定义了 2 个属性，一个是处理请求的函数，一个是当前请求的 request
// 这里定义了一个 blk_dev 数组，内核会在不同设备初始化时，根据定义的数组下标，设定不同设备的请求处理函数。因此后续函数可以传入数组下标来查找相应的执行函数
struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
    { NULL, NULL },             // 无设备
    { NULL, NULL },             // 内存设备
    { NULL, NULL },             // 软盘设备
    { NULL, NULL },             // 硬盘设备
    { NULL, NULL },             // ttyx 设备
    { NULL, NULL },             // tty 设备
    { NULL, NULL }              // lp 打印设备
};

// 文件路径：linux-0.11\kernel\blk_drv\ll_rw_blk.c
void ll_rw_block(int rw, struct buffer_head * bh)
{
    unsigned int major;         // 根据设备号获取设备标识

    // 如果设备标识超出了已定义的设备数或者处理请求的函数为空，打印“尝试读取不存在的块设备”，返回
    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
    !(blk_dev[major].request_fn)) {
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    make_request(major,rw,bh);
}
```

5.8.3 make_request 函数

该函数用于处理缓冲区对于硬盘的读写请求。预读和预写都不是必要操作，仅是对于

性能的优化，首先保证必要的读写操作完成，再考虑预读和预写。请求队列中不能全部是读操作或写操作，内核分配前 2/3 处用于写操作，后 1/3 处用于读操作，由于请求队列是从后往前遍历，因此读操作总会被优先处理。

```
// 文件路径：linux-0.11\kernel\blk_drv\ll_rw_blk.c
static void make_request(int major,int rw, struct buffer_head * bh)
{
    struct request * req;
    int rw_ahead;

    // WRITEA/READA 预写或预读操作是一种特殊情况，它们并不是必要操作，如果缓冲区已经被上锁，我们就忽略预写或预读操作，否则就执行一般的读写操作
    if (rw_ahead = (rw == READA || rw == WRITEA)) {
        if (bh->b_lock)
            return;
        if (rw == READA)
            rw = READ;
        else
            rw = WRITE;
    }
    // 如果传入的读写标识不是 READ/WRITE，死机
    if (rw!=READ && rw!=WRITE)
        panic("Bad block dev command, must be R/W/RA/WA");
    // 将缓冲区上锁
    lock_buffer(bh);
    // 如果是写标识并且缓冲区不为脏，或是读标识且设备已经更新，对缓冲区解锁后返回
    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
        unlock_buffer(bh);
        return;
    }
repeat:
    // 内核不允许写请求全部充满队列，还应该为读请求保留一些空间，并且读操作还应该是优先执行的。请求队列最后 1/3 部分的请求项是为读请求保留的
    // 如果是读请求，从请求队列的起始地址开始，加上 32 个请求项，指针指向请求队列尾部，表示从尾部向前添加读请求项
    if (rw == READ)
        req = request+NR_REQUEST;
    // 如果是写请求，从请求队列的起始地址开始，加上 2/3 个请求项，指针指向请求队列的 2/3 处，表示从请求队列的 2/3 处向前添加写请求项
    else
        req = request+((NR_REQUEST*2)/3);
// 从后向前搜索空闲的请求项
    while (--req >= request)
        if (req->dev<0)
```

```
            break;

    // 如果没有找到空闲的请求项，让新的请求项睡眠。需要检查是否为预读或预写
    // 如果请求队列的头部地址大于请求项的地址，表示请求项的指针已经越过请求队列
    if (req < request) {
        // 如果当前是预读或预写操作，解锁缓冲块，返回
        if (rw_ahead) {
            unlock_buffer(bh);
            return;
        }
        // 否则将当前请求项睡眠，直到请求队列中有空闲项后再插入
        sleep_on(&wait_for_request);
        goto repeat;
    }
    // 在新的请求项中填写请求信息，然后添加到请求队列中
    // 填写请求项指向的缓冲块设备号
    req->dev = bh->b_dev;
    // 填写请求项的命令（READ/WRITE）
    req->cmd = rw;
    // 填写请求项操作失败的次数
    req->errors=0;
     // 填写请求项的起始扇区，根据缓冲块的块号换算成扇区号，左移表示乘 2，即 1 块=2 扇区
    req->sector = bh->b_blocknr<<1;
    // 填写请求项的扇区数
    req->nr_sectors = 2;
    // 填写请求项指向的缓冲块数据区
    req->buffer = bh->b_data;
    // 填写请求项指向等待操作的进程
    req->waiting = NULL;
    // 填写请求项指向的缓冲块
    req->bh = bh;
    // 填写请求项指向的下一个请求项
    req->next = NULL;
    // 将请求项添加到请求队列中
    add_request(major+blk_dev,req);
}
```

5.8.4 lock_buffer 函数

该函数用于将传入的缓冲块上锁，如果缓冲块已经被别的进程上锁，让缓冲块的进程陷入睡眠，直到缓冲块被解锁。

```
// 文件路径：linux-0.11\kernel\blk_drv\ll_rw_blk.c
```

```
static inline void lock_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    bh->b_lock=1;
    sti();
}
```

5.8.5　unlock_buffer 函数

该函数用于将传入的缓冲块解锁，并唤醒缓冲块中等待的进程。

```
// 文件路径：linux-0.11\kernel\blk_drv\ll_rw_blk.c

static inline void unlock_buffer(struct buffer_head * bh)
{
    if (!bh->b_lock)
        printk("ll_rw_block.c: buffer not locked\n\r");
    bh->b_lock = 0;
    wake_up(&bh->b_wait);
}
```

5.8.6　add_request 函数

该函数会调用 cli()函数关闭中断，这样就可以安全地将请求项加入请求队列中。

如果请求项中存在缓冲块，就说明它已经在调度队列里了，那么将缓冲块里的 b_dirt 置 0，代表不脏了，可以写入了。

设备只有一个队列，该队列缓存了待执行的读写请求。

- ☑ 若队列为空，表示当前进程是第一个操作这个设备的进程，它得负责处理该队列。
- ☑ 若队列不为空，表示已经有进程正在处理这个请求队列，只需要将请求放入其中即可。

```
// 文件路径：linux-0.11\kernel\blk_drv\ll_rw_blk.c

static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    struct request * tmp;

    req->next = NULL;
    cli();
```

```
    // 如果请求项中存在缓冲块，就说明它已经在调度队列里了，所以将缓冲块的 b_dirt 属性
置为 0，表示不脏了，可以写入了
    if (req->bh)
        req->bh->b_dirt = 0;
    // 如果设备里的请求项为空，将传入的请求项置为设备的当前请求项
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        // 调用硬盘的请求处理函数 do_hd_request()
        (dev->request_fn)();
        return;
    }

    // 使用梯度算法
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp,req) ||
            !IN_ORDER(tmp,tmp->next)) &&
            IN_ORDER(req,tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
}

// 电梯算法
// IN_ORDER 用于电梯算法里排序的定义，读操作总是在写操作之前进行，读操作相对写操作而
言，对时间有更严格的要求
#define IN_ORDER(s1,s2)
((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd &&
((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev &&
(s1)->sector < (s2)->sector)))
```

5.8.7 do_hd_request 函数

该函数用于处理缓冲区读写硬盘数据的请求。通过将逻辑块转换为磁道、柱面和扇区，进行缓冲块与硬盘间的数据映射。内核与硬盘交互需要建立连接，当检测硬盘状态置位时，才可以通过端口向硬盘发送读写请求。

```
// 代码路径：linux-0.11\kernel\blk_drv\hd.c
// 宏定义当前请求项
#define CURRENT (blk_dev[MAJOR_NR].current_request)

// 宏定义当前设备号
```

```
#define CURRENT_DEV DEVICE_NR(CURRENT->dev)

// 计算硬盘数量
#define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))

// 宏定义初始化请求
#define INIT_REQUEST
repeat:
// 如果设备的当前请求项为空，返回
    if (!CURRENT)
        return;
// 如果当前请求项的设备不是硬盘，死机
    if (MAJOR(CURRENT->dev) != MAJOR_NR)
        panic(DEVICE_NAME ": request list destroyed");
// 如果当前请求项的缓冲块存在，但是没有被上锁，死机
    if (CURRENT->bh) {
        if (!CURRENT->bh->b_lock)
            panic(DEVICE_NAME ": block not locked");
    }


// 代码路径：linux-0.11\kernel\blk_drv\hd.c
void do_hd_request(void)
{
    int i,r;
    unsigned int block,dev;
    unsigned int sec,head,cyl;
    unsigned int nsect;

    // 初始化请求
    INIT_REQUEST;
    // 从请求中获取次设备号和块号
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    // 如果次设备号超出硬盘数量或者当前请求项的扇区号+2大于硬盘设备的扇区数,结束请求,
返回 repeat 标识，重新初始化请求
    // 1 个逻辑块（1024B）= 2 个扇区（2 * 512B）
    if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
        end_request(0);
        goto repeat;        // repeat 标识在 INIT_REQUEST 中，表示重新初始化请求
    }
    // 起始扇区号 + 偏移扇区号，获取要读写的块对应的扇区
    block += hd[dev].start_sect;
    dev /= 5;
```

```
    // 通过逻辑块号，换算成磁盘中的磁盘、扇区和柱面
    __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
        "r" (hd_info[dev].sect));
    __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
        "r" (hd_info[dev].head));
    sec++;
    // 获取要读或写的扇区数
    nsect = CURRENT->nr_sectors;
    // 如果 reset 重置标志存在，进行复位操作
    if (reset) {
        reset = 0;
        recalibrate = 1;
        reset_hd(CURRENT_DEV);
        return;
    }
    // 如果 recalibrate 重新校正标志存在，将磁盘的磁头移动到第 0 柱面
    if (recalibrate) {
        recalibrate = 0;
        hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
            WIN_RESTORE,&recal_intr);
        return;
    }
    // 如果当前请求是写操作，调用 hd_out()函数，根据磁头、柱面、扇区等信息，向硬盘控
制器发送写命令
    if (CURRENT->cmd == WRITE) {
        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
        // 循环判断硬盘状态，如果磁盘请求服务状态置位，表示可以连接磁盘写入，退出循环。
否则循环结束，硬盘连接失败
        for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
            ;
        // 如果硬盘连接失败,无法写入,调用 bad_rw_intr()函数结束本次请求,回到 repeat
标识，重新初始化请求
        if (!r) {
            bad_rw_intr();
            goto repeat;
        }
        // 将缓冲区中的数据通过 HD_DATA 端口写入磁盘
        port_write(HD_DATA,CURRENT->buffer,256);
        // 如果当前请求是读操作，向硬盘控制器发送读命令
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    } else
        panic("unknown hd-command");
}
```

5.9　高版本文件写入原理

2.6 版本内核的写入流程与 0.11 版本内核的写入流程差别不大。本节的主要目的是帮助读者从低版本过渡到高版本的内核代码中，如果读者对于 0.11 版本的内核代码已经有了较为深刻的理解，那么本节的内容阅读起来将会十分轻松。如果读者对于文件写入还有些许疑惑，那么以下流程也能帮助大家梳理高版本中文件写入的逻辑。

5.9.1　sys_open 函数

该函数用于打开指定路径的文件，首先根据用户空间传入的文件路径获取 fd 文件描述符，调用 filp_open()函数根据 fd 获取文件 file，然后将 fd 与 file 关联。

```
// 代码路径：linux-2.6.0\fs\open.c

asmlinkage long sys_open(const char __user * filename, int flags, int mode)
{
    char * tmp;
    int fd, error;
    tmp = getname(filename); // 将用户空间的 filename 复制到内核空间，此时 tmp 指向的内存为内核内存
    fd = PTR_ERR(tmp);   // 检查 tmp 是否包含了错误码，因为错误码范围将会在（0xffff f000, 0xffff ffff)之间（注意：在内核中，将最高地址的最后一页保留，用作错误码表示）
    if (!IS_ERR(tmp)) {      // 返回值为正常指针
        fd = get_unused_fd();
        if (fd >= 0) {       // 成功获取 fd，那么调用 filp_open 打开文件 file
            struct file *f = filp_open(tmp, flags, mode);
            error = PTR_ERR(f);
            if (IS_ERR(f))
                goto out_error;
            fd_install(fd, f);    // 随后将 fd 与 file 文件关联
        }
        out:
        putname(tmp);
    }
    return fd;
    out_error:
    put_unused_fd(fd);
    fd = error;
```

```
    goto out;
}
```

5.9.2 filp_open 函数

该函数用于打开文件。

```
// 代码路径：linux-2.6.0\fs\open.c
/*
 * 在 sys_open 函数中，flag 值的低两位表示如下：
 *  00：read-only，只读
 *  01：write-only，只写
 *  10：read-write，读写
 *  11：special，特殊值
 * 将会被改变为如下含义：
 *  00：no permissions needed，不需要权限
 *  01：read-permission，只读权限
 *  10：write-permission，只写权限
 *  11：read-write，读写权限
 */
struct file *filp_open(const char * filename, int flags, int mode)
{
    ...
    error = open_namei(filename, namei_flags, mode, &nd); // 尝试直接通过文
件路径获取 file 目录
    if (!error)                         // 目录获取成功，那么通过目录对象打开文件
        return dentry_open(nd.dentry, nd.mnt, flags);
    return ERR_PTR(error);
}
```

5.9.3 open_namei 函数

该函数用于根据 pathname 来查找文件，注意：有时候我们传入的 filename 可能携带路径，也可能不携带路径，该函数主要调用 path_lookup()函数来完成查找。Linux 允许在文件不存在时传入 O_CREAT 标志，以自动创建文件，但是为了保证主流程顺畅，这里读者只需要观察文件存在情况即可。

```
// 代码路径：linux-2.6.0\fs\namei.c

int open_namei(const char * pathname,int flag,int mode,struct nameidata *nd)
{
    int acc_mode, error = 0;
```

```
    struct dentry *dentry;      // 文件目录
    struct dentry *dir;
    int count = 0;
    ...
        if (!(flag & O_CREAT)) { // 不需要创建文件，我们主要观察文件存在情况
            error = path_lookup(pathname,lookup_flags(flag)|LOOKUP_OPEN,nd);
                                // 根据路径查找
            if (error)
                return error;
            // 查找成功
            dentry = nd->dentry;
            goto ok;
        }
    // 不存在该文件，那么创建（这里不考虑）
    error=path_lookup(pathname,LOOKUP_PARENT|LOOKUP_OPEN|LOOKUP_CREATE,nd);
    if (error)
        return error;
    ...
        ok:
    error = may_open(nd, acc_mode, flag);   // 检擦文件是否可以打开
    if (error)
        goto exit;
    return 0;
    ...
}

// 根据 name 查找文件，将文件信息放入 nameidata *nd 中
int path_lookup(const char *name, unsigned int flags, struct nameidata *nd)
{
    ...
    if (*name=='/') {           // 以路径符传入，说明此时打开的文件以绝对路径查找
        ...
        // 设置查找路径为根目录
        nd->mnt = mntget(current->fs->rootmnt);
        nd->dentry = dget(current->fs->root);
    }
    else{                       // 否则设置当前进程所处的路径（pwd）为查找路径
        nd->mnt = mntget(current->fs->pwdmnt);
        nd->dentry = dget(current->fs->pwd);
    }
    ...
    return link_path_walk(name, nd);        // 开始查找
}
```

```
// 根据当前 nd 设置的 mnt 和 dentry 查找 name 文件的目录
int link_path_walk(const char * name, struct nameidata *nd)
{
    struct path next;
    struct inode *inode;
    int err;
    unsigned int lookup_flags = nd->flags;

    while (*name=='/')// 移动 name 指针到最后一个'/'分隔符，此时 name 指针指向文件名
        name++;
    if (!*name)                                  // 不存在文件名，直接退出
        goto return_reval;

    inode = nd->dentry->d_inode;                 // 获取目录的 inode
    ...
    for(;;) {                                    // 循环查找
        unsigned long hash;
        struct qstr this;                        // 保存文件名信息
        unsigned int c;
        ...
        this.name = name;                        // 保存文件名
         // 根据文件名计算 hash 值
        c = *(const unsigned char *)name;
        hash = init_name_hash();
        do {
            name++;
            hash = partial_name_hash(c, hash);
            c = *(const unsigned char *)name;
        } while (c && (c != '/'));
         // 保存文件名长度与 hash 值
        this.len = name - (const char *) this.name;
        this.hash = end_name_hash(hash);
        ...
        // 修正"."和".."分别表示当前目录和上一级目录
        if (this.name[0] == '.') switch (this.len) {
            default:
                break;
            case 2:
                if (this.name[1] != '.')
                    break;
                follow_dotdot(&nd->mnt, &nd->dentry);
                inode = nd->dentry->d_inode;
            case 1:
                continue;
```

```
        }
        ...
        err = do_lookup(nd, &this, &next);  // 执行实际搜索
        if (err)
            break;
        ...
        inode = next.dentry->d_inode;
        if (!inode)          // 找到目录（最后一个目录便包含了所查找文件的 inode）
            goto out_dput;
        err = -ENOTDIR;
        if (!inode->i_op)    // 查找到最后一个 inode，但不是目录，所以没有 i_op 操
作，直接退出
            goto out_dput;
        ...
         // 继续查找下一个目录
        dput(nd->dentry);
        nd->mnt = next.mnt;
        nd->dentry = next.dentry;
        err = -ENOTDIR;
        ...
    }
    ...
}

// 实际查找过程
static int do_lookup(struct nameidata *nd, struct qstr *name,
           struct path *path)
{
    struct vfsmount *mnt = nd->mnt;                          // 获取挂载点
    struct dentry *dentry = __d_lookup(nd->dentry, name); // 从设置的目录
dentry 处查找
    if (!dentry)
        goto need_lookup;
    ...
done:                                                        // 查找成功
    path->mnt = mnt;
    path->dentry = dentry;
    return 0;

need_lookup:    // 注意：当内存中的 dentry 不存在时，因为初始化时内存中将不会存在
dentry，dentry 只是对磁盘上的数据进行缓存优化，所以此时将会陷入 FS 文件系统中读取磁盘
数据进行查找
    dentry = real_lookup(nd->dentry, name, nd);
    if (IS_ERR(dentry))
```

```
        goto fail;
    goto done;
...
fail:
    return PTR_ERR(dentry);
}

// 代码路径：linux-2.6.0\fs\dcache.c
// 从 parent 目录处查找 name 所述的 dentry
struct dentry * __d_lookup(struct dentry * parent, struct qstr * name)
{
    unsigned int len = name->len;
    unsigned int hash = name->hash;
    const unsigned char *str = name->name;
    struct hlist_head *head = d_hash(parent,hash); // 根据目录和文件名在
hash 表找到目录下的文件头部指针
    struct dentry *found = NULL;
    struct hlist_node *node;
    ...
    hlist_for_each (node, head) {                    // 遍历该链表
        struct dentry *dentry;
        unsigned long move_count;
        struct qstr * qstr;
        dentry = hlist_entry(node, struct dentry, d_hash); // 获取当前 node
处的 dentry 目录信息
        ...
        if (dentry->d_name.hash != hash)             // hash 值不同则继续查找
            continue;
        if (dentry->d_parent != parent)              // 父目录不同则继续查找
            continue;
        qstr = dentry->d_qstr;
        // 进行名字比较
        if (parent->d_op && parent->d_op->d_compare) {
            if (parent->d_op->d_compare(parent, qstr, name))
                continue;
        } else {
            if (qstr->len != len)
                continue;
            if (memcmp(qstr->name, str, len))
                continue;
        }
        ...                                          // 成功查找
        break;
    }
```

```
    return found;
}
```

5.9.4　dentry_open 函数

dentry 的作用是当创建或第一次访问 inode 时，将文件经过的 inode 全部维护在一个 hash 表或一棵目录树上，这样再次查找 inode 时就不需要从头遍历到尾。dentry 主要起到缓存的作用。

```
struct dentry {
    atomic_t d_count;                    // 目录项引用计数
    unsigned long d_vfs_flags;
    spinlock_t d_lock;                   // 目录项的自旋锁
    struct inode  * d_inode;             // 与目录项关联的 inode
    struct list_head d_lru;              // 长时间未使用的目录项链表
    struct list_head d_child;            // 同一父目录下的子目录链表
    struct list_head d_subdirs;          // 子目录链表
    struct list_head d_alias;            // 目录别名链表
    unsigned long d_time;                // 由 d_revalidate 调用
    struct dentry_operations  *d_op;     // 目录项操作符
    struct super_block * d_sb;           // 目录项根节点
    unsigned int d_flags;                // 目录项标志
    int d_mounted;                       // 目录项挂载点
    void * d_fsdata;                     // 文件依赖数据
    struct rcu_head d_rcu;               // 回收目录项对象时，由 RCU 描述符使用
    struct dcookie_struct * d_cookie;    // 保存 kmem_cache 信息
    unsigned long d_move_count;          // 在无锁状态下查找移动的目录数量
    struct qstr * d_qstr;                // 在无锁状态下快速查找指向的元数据信息
    struct dentry * d_parent;            // 父目录
    struct qstr d_name;                  // 快速查找目录名
    struct hlist_node d_hash;            // 查找 dentry hash 表
    struct hlist_head * d_bucket;        // 查找 hash 桶
    unsigned char d_iname[DNAME_INLINE_LEN_MIN]; // 存放短文件名
} ____cacheline_aligned;
```

在 open_namei 中，通过 dentry 遍历找到文件所述的目录 dentry 信息，那么 dentry_open 函数将会根据 dentry 找到该目录下的 name 文件。

```
// 代码路径：linux-2.6.0\fs\open.c

struct file *dentry_open(struct dentry *dentry,struct vfsmount *mnt,int flags)
{
    struct file * f;
    struct inode *inode;
```

```
    int error;
    error = -ENFILE;
    f = get_empty_filp();    // 分配一个新的 file 结构，注意：在该函数中将会检查打开的文件的计数，读者可以自行查看
    ...
    inode = dentry->d_inode;              // 获取目录 inode 信息
    ...
    // 保存文件的元数据信息
    f->f_dentry = dentry;
    f->f_vfsmnt = mnt;
    f->f_pos = 0; // 初始文件操作点从 offset 偏移量为 0 处开始
    f->f_op = fops_get(inode->i_fop);   // 从 inode 中获取操作文件的函数指针
    file_move(f, &inode->i_sb->s_files);// 将打开文件信息绑定到 super_block 中，此时可以根据超级块看到该文件系统打开的所有文件
    if (f->f_op && f->f_op->open) {     // 若 inode 操作存在 open 函数，那么进行回调（对于 ext2 来说，这里对文件长度进行了校验）
        error = f->f_op->open(inode,f);
        if (error)
            goto cleanup_all;
    }
    ...
}
```

5.9.5 fd_install 函数

该函数用于将 fd 与打开文件 file 进行关联，这里就是将其放入当前进程 task_struct 的 files 数组对应 fd 下标处。源码描述如下。

```
// 代码路径：linux-2.6.0\fs\open.c
void fd_install(unsigned int fd, struct file * file)
{
    struct files_struct *files = current->files;
    spin_lock(&files->file_lock);
    if (unlikely(files->fd[fd] != NULL))
        BUG();
    files->fd[fd] = file;        // 在对应 fd 下标处放置 file，将 fd 返回给用户
    spin_unlock(&files->file_lock);
}
```

5.9.6 sys_write 函数

该系统调用将根据传入的 fd 下标，找到对应的 file 结构，然后执行写入操作。

```
// 代码路径：linux-2.6.0\fs\read_write.c
asmlinkage ssize_t sys_write(unsigned int fd, const char __user * buf, size_t
count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;
    file = fget_light(fd, &fput_needed);    // 根据 fd 找到 file
    if (file) {                            // 文件存在，那么开始调用 vfs 接口写入
        ret = vfs_write(file, buf, count, &file->f_pos);
        fput_light(file, fput_needed);
    }

    return ret;
}

// 代码路径：linux-2.6.0\fs\file_table.c
struct file *fget_light(unsigned int fd, int *fput_needed)
{
    struct file *file;
    struct files_struct *files = current->files;    // 获取进程打开文件结构
    *fput_needed = 0;
    if (likely((atomic_read(&files->count) == 1))) { // 原子性增加 file 引用
计数
        file = fcheck(fd);                    // 从 fd 数组下标处获取 file 结构
    } else {                                  // 否则失败，那么上锁获取
        spin_lock(&files->file_lock);
        file = fcheck(fd);
        if (file) {
            get_file(file);
            *fput_needed = 1;
        }
        spin_unlock(&files->file_lock);
    }
    return file;
}

// 代码路径：linux-2.6.0\include\linux\file.h
#define fcheck(fd)  fcheck_files(current->files, fd)
static inline struct file * fcheck_files(struct files_struct *files,
unsigned int fd)
{
    struct file * file = NULL;
    if (fd < files->max_fds)                  // 直接从下标处获取即可
        file = files->fd[fd];
```

```
    return file;
}
```

5.9.7 vfs_write 函数

该函数将会调用 vfs 接口向 file 文件中写入数据。首先根据 file 获取到 inode，然后执行写入。

```
// 代码路径：linux-2.6.0\fs\read_write.c
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count,
loff_t *pos)
{
    struct inode *inode = file->f_dentry->d_inode; // 通过 file 结构获取到文件 inode 节点
    ssize_t ret;
    ...
    if (!ret) {
        ret = security_file_permission (file, MAY_WRITE); // 检测权限
        if (!ret) {
            if (file->f_op->write)
                // 若文件存在 write 函数，那么直接调用
                ret = file->f_op->write(file, buf, count, pos);
            else
                // 否则尝试使用 aio_write 并等待完成，因为有些设备文件只支持异步写入
                ret = do_sync_write(file, buf, count, pos);
            ...
        }
    }

    return ret;
}
```

5.9.8 generic_file_write 函数

generic_file_write 函数将会作为 write 函数在 ext2 文件系统的实现，所以我们关注该函数的实现过程即可。

```
// 代码路径：linux-2.6.0\fs\ext2\file.c

// ext2 文件系统注册的文件写入操作
struct file_operations ext2_file_operations = {
    ...
```

```
    .write      = generic_file_write,
    ...
};

// 代码路径：linux-2.6.0\mm\filemap.c

ssize_t generic_file_write(struct file *file, const char __user *buf, size_t
count, loff_t *ppos)
{
    struct inode    *inode = file->f_dentry->d_inode->i_mapping->host;
// 获取文件 inode
    ssize_t     err;
    struct iovec local_iov = { .iov_base = (void __user *)buf, .iov_len =
count };                    // 构建写入向量信息
    down(&inode->i_sem);    // 上锁并开始写入
    err = generic_file_write_nolock(file, &local_iov, 1, ppos);
    up(&inode->i_sem);
    return err;
}

// 代码路径：linux-2.6.0\mm\filemap.c
ssize_t generic_file_write_nolock(struct file *file,const struct iovec *iov,
              unsigned long nr_segs, loff_t *ppos)
{
    struct kiocb kiocb;      // I/O 控制块，用于表示一次 I/O 操作
    ssize_t ret;
    init_sync_kiocb(&kiocb, file);
    ret = generic_file_aio_write_nolock(&kiocb, iov, nr_segs, ppos); // 调
用该函数完成写入
    if (-EIOCBQUEUED == ret) // IOCB 控制块已经进入调度队列，那么等待其执行完成
        ret = wait_on_sync_kiocb(&kiocb);
    return ret;
}

// 代码路径：linux-2.6.0\include\linux\aio.h

// 初始化 I/O 控制块
#define init_sync_kiocb(x, filp)
    do {
        struct task_struct *tsk = current;
        (x)->ki_flags = 0;
        (x)->ki_users = 1;
        (x)->ki_key = KIOCB_SYNC_KEY;
        (x)->ki_filp = (filp);
```

```
        (x)->ki_ctx = &tsk->active_mm->default_kioctx;
        (x)->ki_cancel = NULL;
        (x)->ki_user_obj = tsk;
    } while (0)
```

5.9.9 generic_file_aio_write_nolock 函数

该函数将实现真正文件数据写入流程。我们看到将会根据文件打开的类型来选择写入方式，如果是 O_DIRECT，那么直接写入调度层；如果是 O_SYNC，那么等待数据落盘，否则我们写入 page cache，并且由于 buffer_head 的数据区存在于 page 中，当函数 filemap_copy_from_user 复制成功时，表明写入完成。

```
// 代码路径：linux-2.6.0\mm\filemap.c
ssize_t generic_file_aio_write_nolock(struct kiocb *iocb, const struct
iovec *iov, unsigned long nr_segs, loff_t *ppos)
{
    ...
    for (seg = 0; seg < nr_segs; seg++) { // 循环计算写入数量，注意：这里由于是
单次写入，所以在上述函数中 nr_segs 为 1
        const struct iovec *iv = &iov[seg];
        ocount += iv->iov_len;           // 增加计数
        if (unlikely((ssize_t)(ocount|iv->iov_len) < 0))
            return -EINVAL;
        if (access_ok(VERIFY_READ, iv->iov_base, iv->iov_len)) // 检测地址
是否可读
            continue;
        if (seg == 0)
            return -EFAULT;
        nr_segs = seg;
        ocount -= iv->iov_len;           // 当前写入段有误，那么减少增加的计数
        break;
    }
    ...
    if (unlikely(file->f_flags & O_DIRECT)) { // 文件打开类型为 O_DIRECT，那
么不经过 page cache，直接写入
        ...
        written = generic_file_direct_IO(WRITE, iocb,
                                  iov, pos, nr_segs);      // 直接写入
        ...
        if (written >= 0 && file->f_flags & O_SYNC)         // 文件打开类型
为 O_SYNC，那么将写入的文件数据落盘
            status = generic_osync_inode(inode, OSYNC_METADATA);
        ...
```

```
    }
    // 否则执行数据写入 page cache 中
    buf = iov->iov_base;
    do { // 循环写入
        ...
        page = __grab_cache_page(mapping,index,&cached_page,&lru_pvec);
// 获取当前写入文件的 page 页（过程暂时省略，这里了解流程即可）
        ...
        status = a_ops->prepare_write(file, page, offset, offset+bytes);
        ...
        if (likely(nr_segs == 1))    // 从用户空间 buf 中将数据复制到 page 页中，
由于 buffer_head 的数据便存在于 page 中，当该函数执行完成，那么就完成了写入操作
            copied = filemap_copy_from_user(page, offset,
                                    buf, bytes);
        else
            copied = filemap_copy_from_user_iovec(page, offset,
                                        cur_iov, iov_base, bytes);
        ...
        status = a_ops->commit_write(file, page, offset, offset+bytes);
                                    // 提交写入结果
        ....
    } while (count);
    ...
}
```

5.9.10　generic_commit_write 函数

在 ext2 中可以看到是该函数完成了提交写操作，这里将会把映射到 page 中的 buffer_head 高速缓冲区设置为 dirty（脏）。

```
// 代码路径：linux-2.6.0\fs\ext2\inode.c

struct address space operations ext2 aops = {
    ...
    .commit write      = generic commit write,
    ...
};

// 代码路径：linux-2.6.0\fs\buffer.c
int generic_commit_write(struct file *file, struct page *page, unsigned from,
unsigned to)
{
    ...
```

```
        // 提交写
    __block_commit_write(inode,page,from,to);
    // 写入成功，那么标记 inode 为脏，表明内存中的数据尚未落盘
    if (pos > inode->i_size) {
        i_size_write(inode, pos);
        mark_inode_dirty(inode);
    }
    return 0;
}

// 代码路径：linux-2.6.0\fs\buffer.c
static int __block_commit_write(struct inode *inode, struct page *page,
        unsigned from, unsigned to)
{
    ...
        // 从页结构中获取 buffer_head 高速缓冲区块，然后遍历写入
    for(bh = head = page_buffers(page), block_start = 0;
        bh != head || !block_start;
        block_start=block_end, bh = bh->b_this_page) {
        block_end = block_start + blocksize;
        // 部分写入
        if (block_end <= from || block_start >= to) {
            if (!buffer_uptodate(bh))
                partial = 1;
            // 写入整个高速缓冲区块，并标记为脏
        } else {
            set_buffer_uptodate(bh);
            mark_buffer_dirty(bh);
        }
    }
    ...
    return 0;
}
```

5.10 小　　结

本章基于如何提升 I/O 性能入手，从内存到缓存、从缓存到磁盘间进行了拆解分析。

为了适配更多的文件系统，内核将文件系统模块化，通过虚拟文件系统达到解耦合和高扩展的目的。用户层定义统一接口函数，当函数经过系统调用到达内核态时，内核函数 sys_xxx 根据系统所加载的不同文件系统执行不同的函数实现。

0.11 文件系统中最重要的概念包括 super block、inode 位图、逻辑块位图、inode 和缓冲块，而到了 2.6 版本的内核则扩展了 vfs_mount 文件树（用于挂载文件系统），但是实际文件的查找过程还是不变的，通过文件路径获取 fd 文件描述符，通过 fd 查找 dentry，如果 dentry 中有缓存的 inode 则直接获取，否则从根节点开始查找文件。获取文件通常是为了读写文件，因此根据文件的 inode 和逻辑块找到对应的缓冲区，进行读写。当缓冲区写完后，并不会立即同步到磁盘，而是需要等待程序调用 sync 同步函数，才会将数据刷新到磁盘中。当缓冲区数据刷新到磁盘时，需要先将逻辑块转换成对应扇区，然后对磁盘发送请求，尝试与磁盘进行连接，如果连接成功，则通过磁盘端口将数据写入扇区。

以下为文件系统布局的推理过程。

要研究 I/O，就需要有文件系统的前置知识，我们先从磁盘入手，了解磁盘布局后，再推理到内存。磁盘由一组连续的空间组成，为了方便管理这一组连续的空间（避免外碎片问题，有利于快速分配），就将一系列的连续空间划分为一个个的数据块（盘块）。

如何找到空数据块？我们知道位图是以 bit 为单位存储数据并建立映射关系来查找位置，而 bit 的表现形式就是 0 和 1，因此可以大大减少存储空间，缩短在大量数据中查询的时间。磁盘正好可以利用这一特征，使用 0 表示当前块为空闲块，使用 1 表示当前块已分配。这样一来，当数据想要插入磁盘某个数据块时，在扫描磁盘的过程中一旦发现标识为 0 的数据块，即可立即插入，因此使用逻辑块位图的方式表示是否为空数据块。

如何找到所需要的文件？文件由元数据和数据块组成，描述数据块的元数据称之为 inode，因此可以通过 inode 索引节点找到文件。

如何找到 inode？提前将 inode 生成一张表，通过 inode 位图找到空的 inode 即可。那么又如何找到这张 inode 表、数据块起始地址和数据块位图地址呢？超级块中保存了这些信息，可以通过超级块来查找 inode 表、数据块起始地址和数据块位图地址。因此通过 inode 位图找到 inode 索引节点，通过逻辑块位图找到空的数据块。

如何找到操作系统引导信息？约定将 OS 的引导信息放在固定位置：磁盘的第 0 个柱面第 0 个扇区第 1 个磁道的 512 字节处，后面跟着的就是超级块信息。

如果数据区无限大，那么 inode 节点就无限大。解决方法：引入块组，类似页表查找过程（多级分页），实现无限嵌套。

数据区是为了把盘块分割，为什么要分成一块块的？减少碎片，方便管理。

内存内碎片怎么解决？磁盘空间不值钱，想怎么用就怎么用。所以使用避免外碎片的方式规整化管理，允许有内碎片，因为空间特别大，带来的收益高。如果要找到的那些块是空的，就需要一个位图。逻辑块位图用于标识哪一个块不是空闲的。

文件有元数据信息和数据块信息，元数据信息描述了这个文件由谁来创建、在什么时

间创建、被引用了几次。这个元数据块就称为 inode。预先构建一张表，用 inode 号（类似索引）找到 inode，用 inode 找到数据块信息。

为什么需要 inode 位图？因为需要从 inode 表里找到没有被占用的数据块，位图只有 0 和 1 两个状态，用于标识有或无，所以遍历位图即可。

那么如何知道位图在哪？inode 节点在哪？就需要有描述 inode 位图、逻辑块位图、inode 节点的元数据信息。约定第二个块为超级块，超级块可以拥有 inode 位图、逻辑块位图、inode 节点的信息。

如果数据区无限大，那么 inode 表、位图会很大。此时将此结构分为块组，只需要知道这个文件在那个块组，块组保存少量数据区、节点、位图即可。

文件如何写入磁盘？用户空间传入一个 fd 文件描述符和要写的数据，然后放入内核态，内核通过 fd 下标找到对应的 file，通过 file 找到 inode，通过 inode 找到对应的缓冲块，把数据写入缓冲块即可。

文件如何读取磁盘数据？通过 fd 文件描述符和一个缓冲区，放入要读取的数据。流程依旧是通过 fd 下标找到 file，通过 file 找到 inode，通过 inode 找到对应的缓冲块。如果缓冲块存在，将数据放入缓冲区；如果不存在，分配一个缓冲块到磁盘里读取，读取到缓冲块后，把缓冲块数据放入用户缓冲区，然后返回。