Operating System

Dr. GuoJun LIU

Harbin Institute of Technology

http://os.guojunhit.cn

Chapter A3

File System

Linux 0.11

Outline

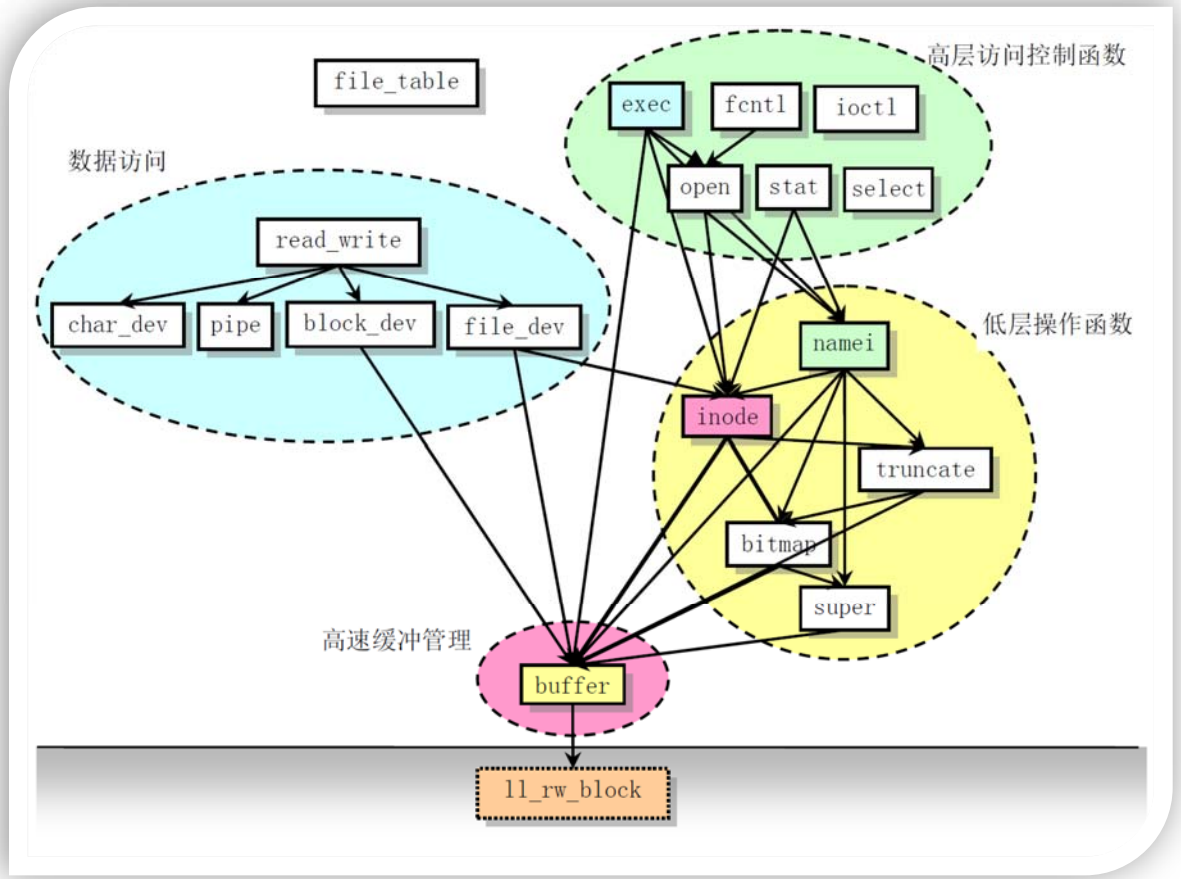
- 文件系统
- 进程打开文件使用的内核数据结构
- 缓冲
- 用户程序读写操作过程

文件系统

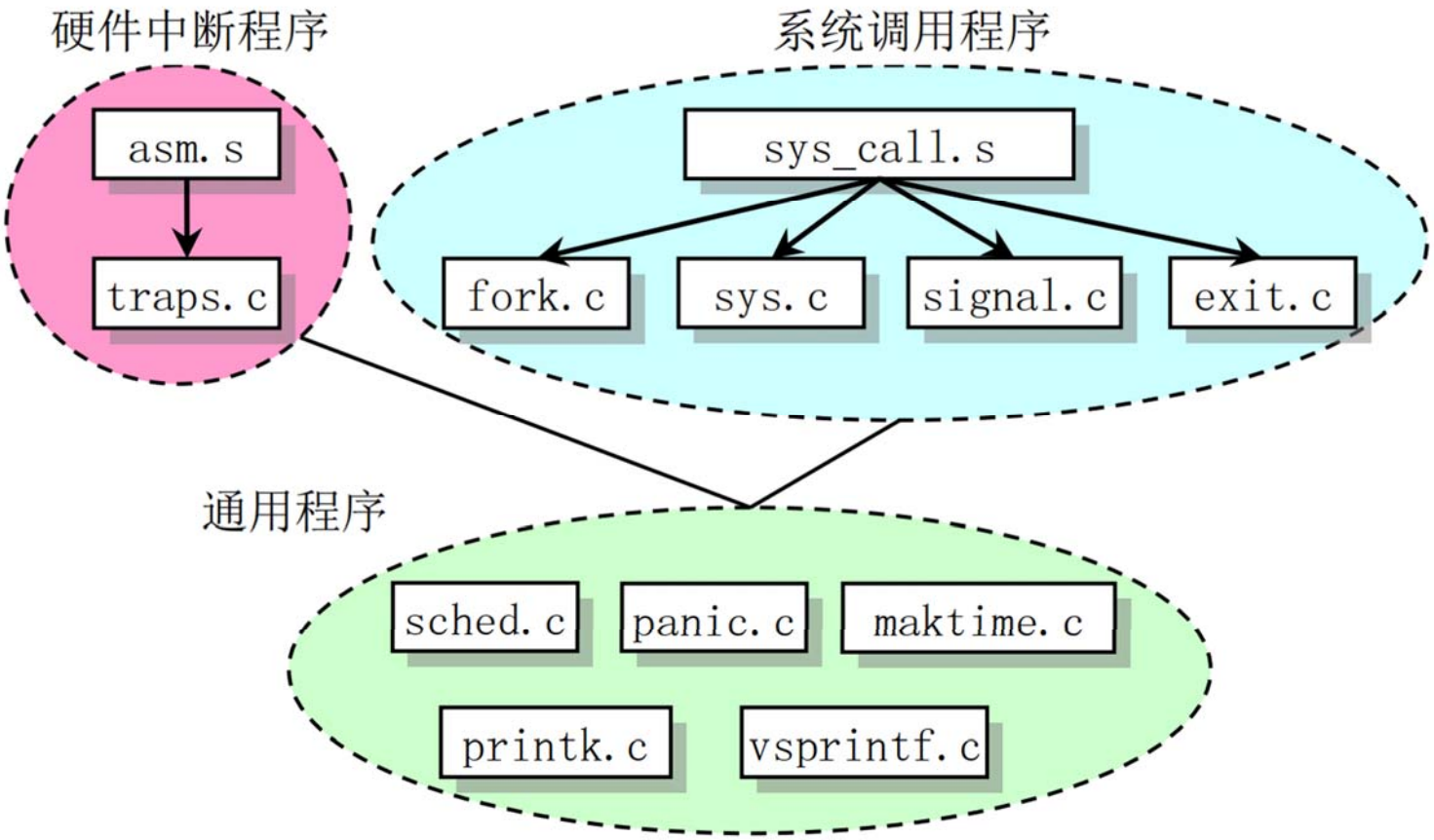
fs 目录中各程序函数之间的引用关系

	名称	大小
	Makefile	7176 bytes
	bitmap.c	4007 bytes
	block_dev.c	1763 bytes
	buffer.c	9072 bytes
	char_dev.c	2103 bytes
	exec.c	9908 bytes
	fcntl.c	1455 bytes
	file_dev.c	1852 bytes
	file_table.c	122 bytes
	inode.c	7166 bytes
	ioctl.c	1136 bytes
	namei.c	18958 bytes
	open.c	4862 bytes
	pipe.c	2834 bytes
	read_write.c	2802 bytes
	select.c	6381 bytes
	stat.c	1875 bytes
	super.c	5603 bytes
	truncate.c	1692 bytes

linux/fs 目录



各文件的调用层次关系


硬件中断程序
asm.s
traps.c
系统调用程序
sys_call.s
fork.c
sys.c
signal.c
exit.c
通用程序
sched.c
panic.c
maktime.c
printk.c
vsprintf.c

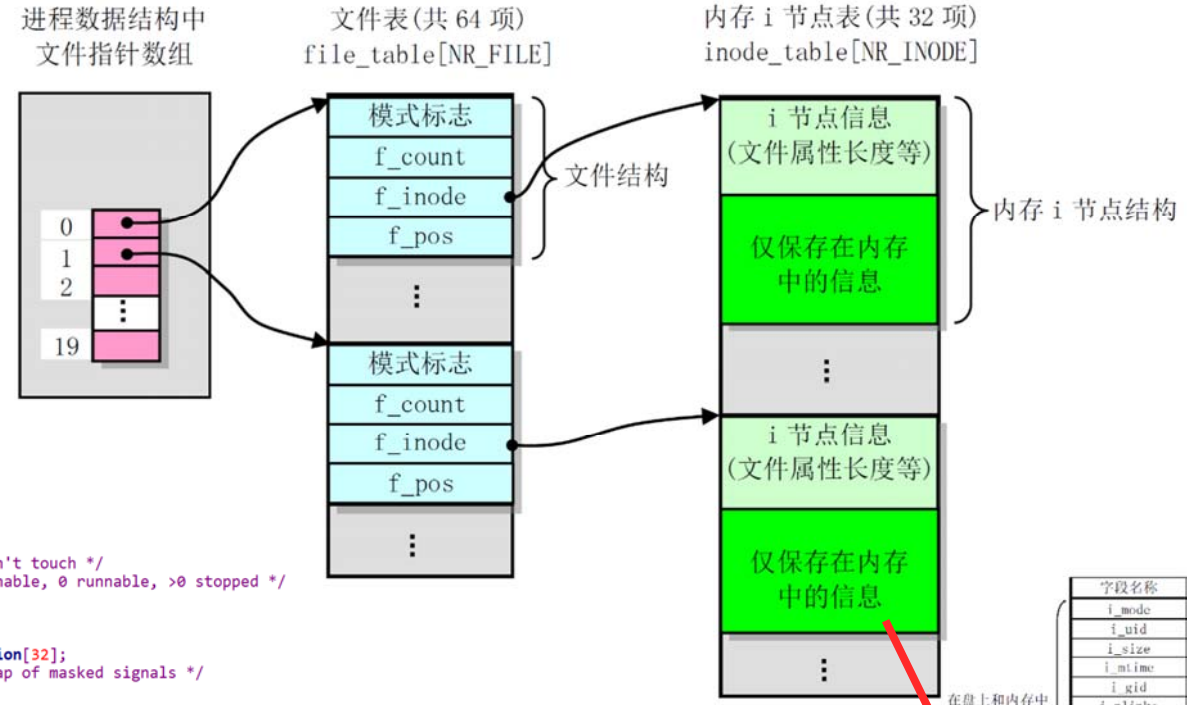
fs 目录中各程序函数功能分类

- 从功能上分为四个部分
  - 有关高速缓冲区的管理程序
    - 主要实现了对硬盘等块设备进行数据高速存取的函数
    - 该部分内容集中在buffer.c 程序中实现
  - 描述了文件系统的低层通用函数
    - 说明了文件索引节点的管理、磁盘数据块的分配和释放以及文件名与 i 节点的转换算法
  - 有关对文件中数据进行读写操作
    - 包括对字符设备、管道、块读写文件中数据的访问
  - 涉及文件的系统调用接口的实现
    - 主要涉及文件打开、关闭、创建以及有关文件目录操作等的系统调用

进程打开文件使用的内核数据结构

```
43: #define NR_OPEN 20
44: #define NR_INODE 32
45: #define NR_FILE 64
46: #define NR_SUPER 8
47: #define NR_HASH 307
48: #define NR_BUFFERS nr_buffers
49: #define BLOCK_SIZE 1024
50: #define BLOCK_SIZE_BITS 10
```

```
162: extern struct m_inode inode_table[NR_INODE];
163: extern struct file file_table[NR_FILE];
164: extern struct super_block super_block[NR_SUPER];
165: extern struct buffer_head * start_buffer;
166: extern int nr_buffers;
```




```
93: struct m_inode {
94:     unsigned short i_mode;
95:     unsigned short i_uid;
96:     unsigned long i_size;
97:     unsigned long i_mtime;
98:     unsigned char i_gid;
99:     unsigned char i_nlinks;
100:    unsigned short i_zone[9];
101: /* these are in memory also */
102:    struct task_struct * i_wait;
103:    unsigned long i_atime;
104:    unsigned long i_ctime;
105:    unsigned short i_dev;
106:    unsigned short i_num;
107:    unsigned short i_count;
108:    unsigned char i_lock;
109:    unsigned char i_dirt;
110:    unsigned char i_pipe;
111:    unsigned char i_mount;
112:    unsigned char i_seek;
113:    unsigned char i_update;
114: } « end m_inode » ;
```

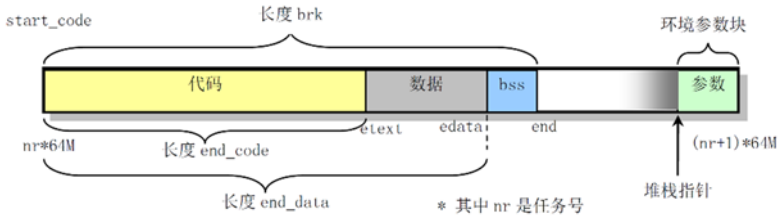
```
80: struct task_struct {
81: /* these are hardcoded - don't touch */
82:     long state; /* -1 unrunnable, 0 runnable, >0 stopped */
83:     long counter;
84:     long priority;
85:     long signal;
86:     struct sigaction sigaction[32];
87:     long blocked;   /* bitmap of masked signals */
88: /* various fields */
89:     int exit_code;
90:     unsigned long start_code,end_code,end_data,brk,start_stack;
91:     long pid,father,pgrp,session,leader;
92:     unsigned short uid,euid,suid;
93:     unsigned short gid,egid,sgid;
94:     long alarm;
95:     long utime,stime,cutime,cstime,start_time;
96:     unsigned short used_math;
97: /* file system info */
98:     int tty;        /* -1 if no tty, so it must be signed */
99:     unsigned short umask;
100:    struct m_inode * pwd;
101:    struct m_inode * root;
102:    struct m_inode * executable;
103:    unsigned long close_on_exec;
104:    struct file * filp[NR_OPEN];
105: /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
106:    struct desc_struct ldt[3];
107: /* tss for this task */
108:    struct tss_struct tss;
109: } « end task_struct » ;
```

```
157: struct dir_entry {
158:     unsigned short inode;
159:     char name[NAME_LEN];
160: };
161:
162: extern struct m_inode inode_table[NR_INODE];
163: extern struct file file_table[NR_FILE];
```

```
116: struct file {
117:     unsigned short f_mode;
118:     unsigned short f_flags;
119:     unsigned short f_count;
120:     struct m_inode * f_inode;
121:     off_t f_pos;
122: };
123:
```

	字段名称	数据类型	说明
在盘上和内存中的字段，共 32 字节	i_mode	short	文件的类型和属性（rwx 位）
	i_uid	short	文件宿主的用户 id
	i_size	long	文件长度（字节）
	i_mtime	long	修改时间（从 1970.1.1:0 时算起，秒）
	i_gid	char	文件宿主的组 id
	i_nlinks	char	链接数（有多少个文件目录项指向该 i 节点）
	i_zone[9]	short	文件所占用的盘上逻辑块号数组。其中： zone[0]-zone[6]是直接块号； zone[7]是一次间接块号； zone[8]是二次（双重）间接块号。 注：zone 是区的意思，可译成区块或逻辑块。 对于设备特殊文件名的 i 节点，其 zone[0]中存放的是该文件名所指设备的设备号。
仅在内存中使用的字段	i_wait	task_struct *	等待该 i 节点的进程。
	i_atime	long	最后访问时间。
	i_ctime	long	i 节点自身被修改时间。
	i_dev	short	i 节点所在的设备号。
	i_num	short	i 节点号。
	i_count	short	i 节点被引用的次数，0 表示空闲。
	i_lock	char	i 节点被锁定标志。
	i_dirt	char	i 节点已被修改（脏）标志。
	i_pipe	char	i 节点用作管道标志。
	i_mount	char	i 节点安装了其他文件系统标志。
	i_seek	char	搜索标志（lseek 操作时）。
	i_update	char	i 节点已更新标志。

Page fault




```
366: void do_no_page(unsigned long error_code,unsigned long address)
367: {
368:     int nr[4];
369:     unsigned long tmp;
370:     unsigned long page;
371:     int block,i;
372:
373:     address &= 0xfffff000;
374:     tmp = address - current->start_code;
375:     if (!current->executable || tmp >= current->end_data) {
376:         get_empty_page(address);
377:         return;
378:     }
379:     if (share_page(tmp))
380:         return;
381:     if (!(page = get_free_page()))
382:         oom();
383: /* remember that 1 block is used for header */
384:     block = 1 + tmp/BLOCK_SIZE;
385:     for (i=0 ; i<4 ; block++,i++)
386:         nr[i] = bmap(current->executable,block);
387:     bread_page(page,current->executable->i_dev,nr);
388:     i = tmp + 4096 - current->end_data;
389:     tmp = page + 4096;
390:     while (i-- > 0) {
391:         tmp--;
392:         *(char *)tmp = 0;
393:     }
394:     if (put_page(page,address))
395:         return;
396:     free_page(page);
397:     oom();
398: } « end do_no_page »
```

```
page_fault:
        xchgl %eax,(%esp)
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%edx
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
        movl %cr2,%edx
        pushl %edx
        pushl %eax
        testl $1,%eax
        jne 1f
        call do_no_page
        jmp 2f
1:      call do_wp_page
2:      addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

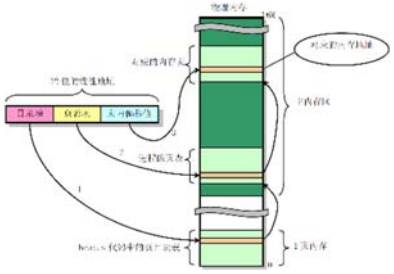
```
140: int bmap(struct m_inode * inode,int block)
141: {
142:     return _bmap(inode,block,0);
143: }
```

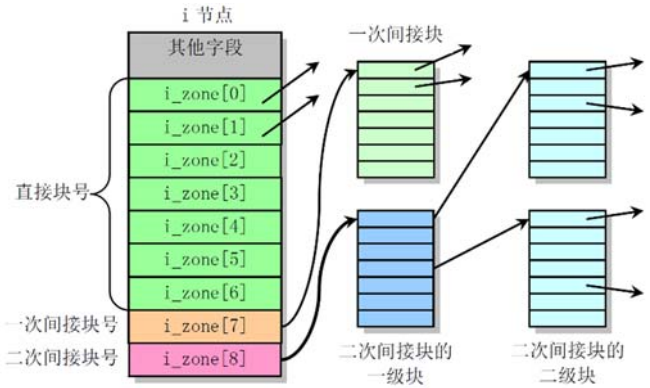
```
72: static int _bmap(struct m_inode * inode,int block,int create)
73: {
74:     struct buffer_head * bh;
75:     int i;
76:
77:     if (block<0)
78:         panic("_bmap: block<0");
79:     if (block >= 7+512+512*512)
80:         panic("_bmap: block>big");
81:     if (block<7) {
82:         if (create && !inode->i_zone[block])
83:             if ((inode->i_zone[block]=new_block(inode->i_dev))) {
84:                 inode->i_ctime=CURRENT_TIME;
85:                 inode->i_dirt=1;
86:             }
87:         return inode->i_zone[block];
88:     }
89:     block -= 7;
```

```
286: #define COPYBLK(from,to) \
287: __asm__("cld\n\t" \
288:     "rep\n\t" \
289:     "movsl\n\t" \
290:     ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
291:     )
292:
293: /*
294:  * bread_page reads four buffers into memory at the desired address. It's
295:  * a function of its own, as there is some speed to be got by reading them
296:  * all at the same time, not waiting for one to be read, and then another
297:  * etc.
298:  */
299: void bread_page(unsigned long address,int dev,int b[4])
300: {
301:     struct buffer_head * bh[4];
302:     int i;
303:
304:     for (i=0 ; i<4 ; i++)
305:         if (b[i]) {
306:             if ((bh[i] = getblk(dev,b[i])))
307:                 if (!bh[i]->b_uptodate)
308:                     ll_rw_block(READ,bh[i]);
309:         } else
310:             bh[i] = NULL;
311:     for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
312:         if (bh[i]) {
313:             wait_on_buffer(bh[i]);
314:             if (bh[i]->b_uptodate)
315:                 COPYBLK((unsigned long) bh[i]->b_data,address);
316:             brelse(bh[i]);
317:         }
318: } « end bread_page »
```

```
39: static inline void wait_on_buffer(struct buffer_head * bh)
40: {
41:     cli();
42:     while (bh->b_lock)
43:         sleep_on(&bh->b_wait);
44:     sti();
45: }
```

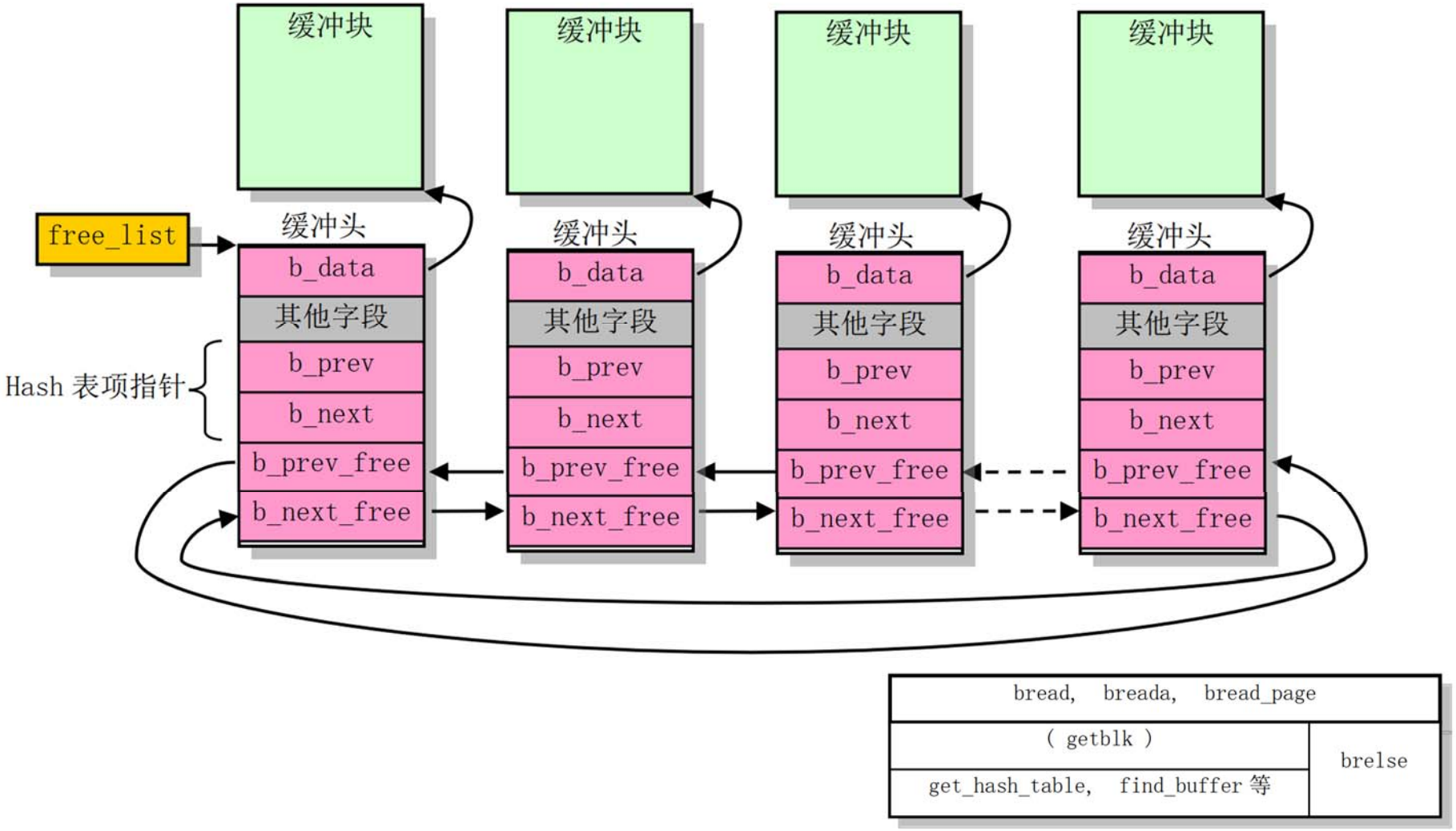
```
151: void sleep_on(struct task_struct **p)
152: {
153:     struct task_struct *tmp;
154:
155:     if (!p)
156:         return;
157:     if (current == &(init_task.task))
158:         panic("task[0] trying to sleep");
159:     tmp = *p;
160:     *p = current;
161:     current->state = TASK_UNINTERRUPTIBLE;
162:     schedule();
163:     if (tmp)
164:         tmp->state=0;
165: }
```



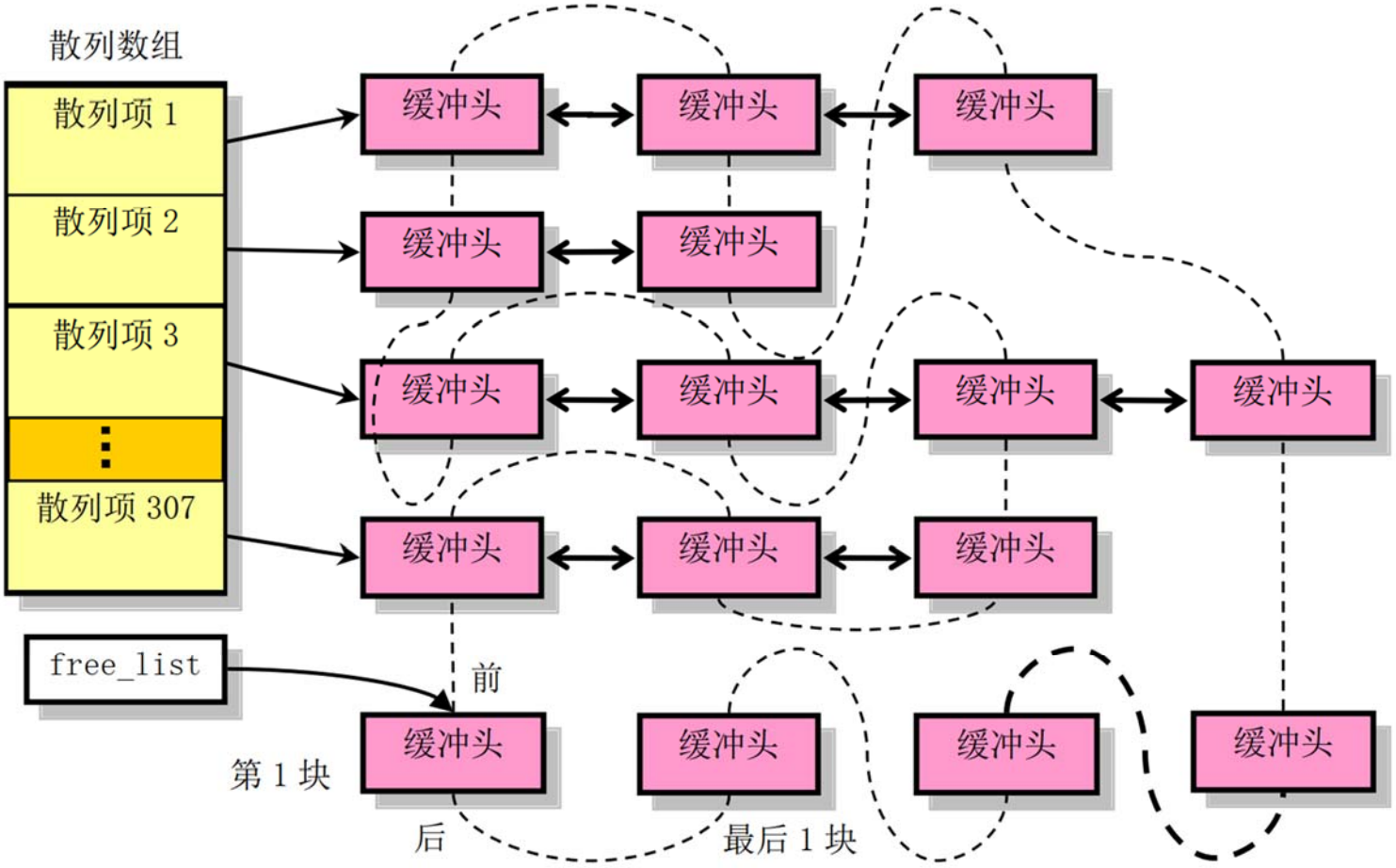


缓冲

缓冲块组成的双向循环空闲链表结构

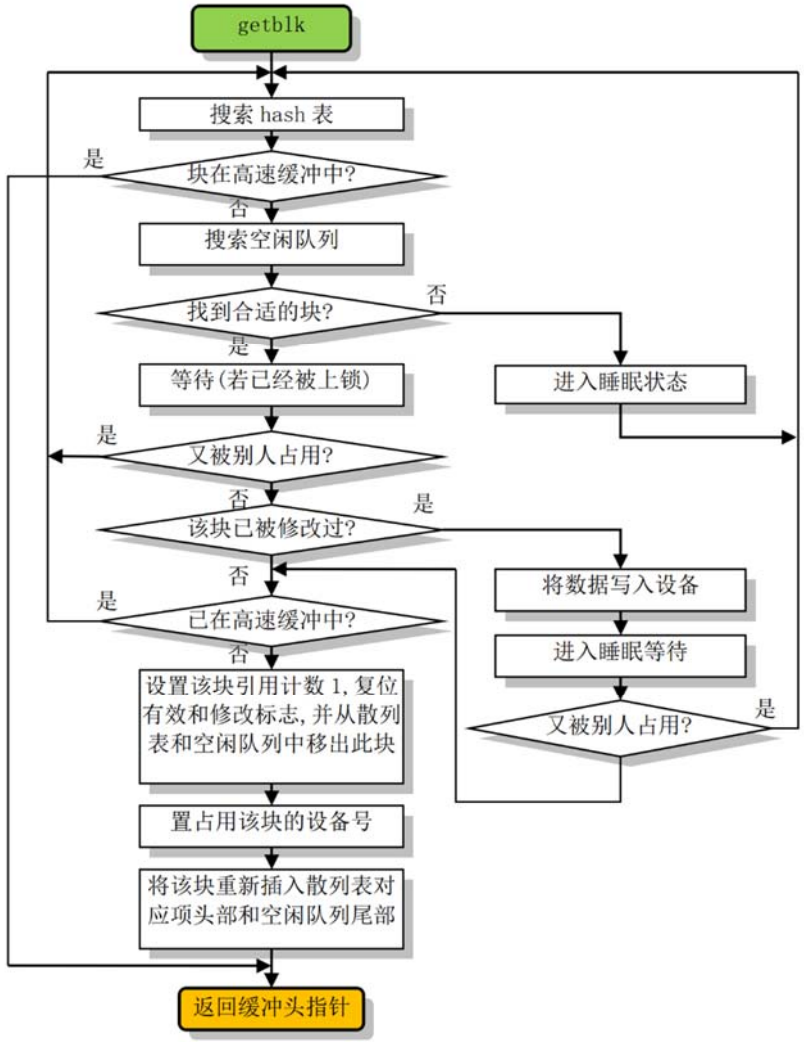


某一时刻内核中缓冲块散列队列



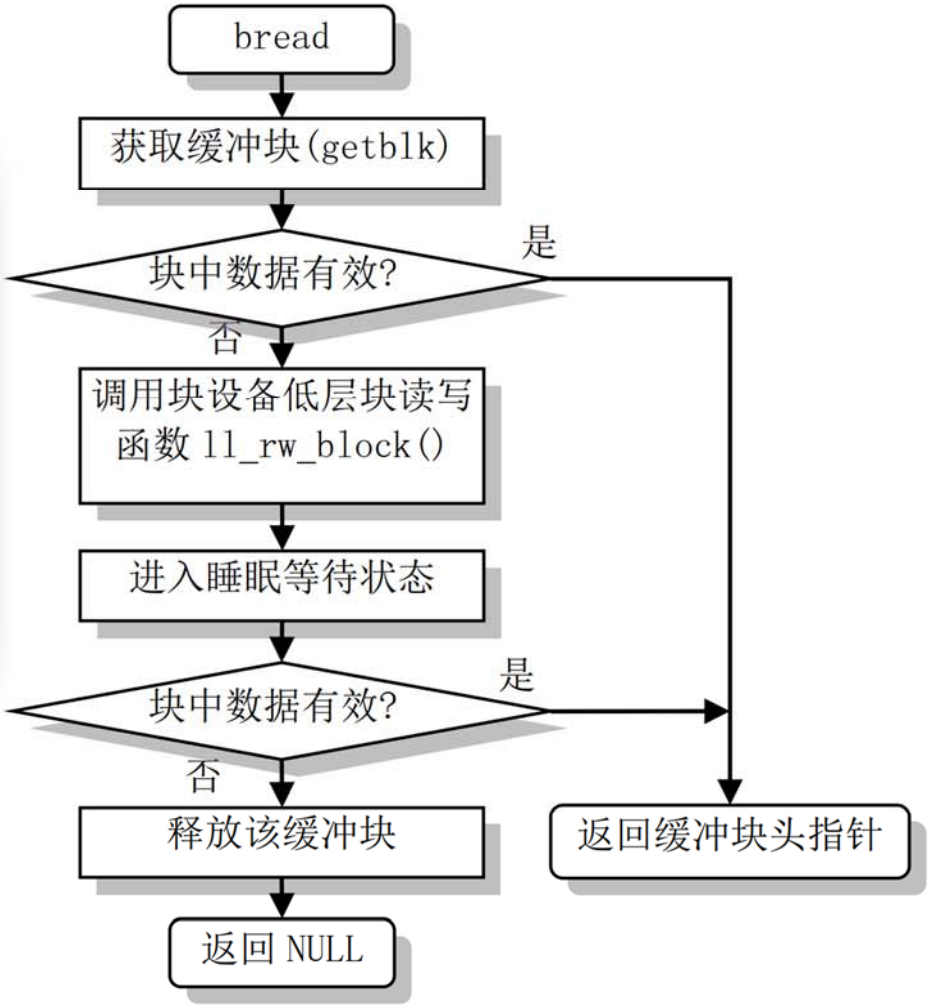
getblk()函数执行流程图

```
209: struct buffer_head * getblk(int dev,int block)
210: {
211:     struct buffer_head * tmp, * bh;
212:
213: repeat:
214:     if ((bh = get_hash_table(dev,block)))
215:         return bh;
216:     tmp = free_list;
217:     do {
218:         if (tmp->b_count)
219:             continue;
220:         if (!bh || BADNESS(tmp)<BADNESS(bh)) {
221:             bh = tmp;
222:             if (!BADNESS(tmp))
223:                 break;
224:         }
225: /* and repeat until we find something good */
226:     } while ((tmp = tmp->b_next_free) != free_list);
227:     if (!bh) {
228:         sleep_on(&buffer_wait);
229:         goto repeat;
230:     }
231:     wait_on_buffer(bh);
232:     if (bh->b_count)
233:         goto repeat;
234:     while (bh->b_dirt) {
235:         sync_dev(bh->b_dev);
236:         wait_on_buffer(bh);
237:         if (bh->b_count)
238:             goto repeat;
239:     }
240: /* NOTE!! While we slept waiting for this block, somebody else might */
241: /* already have added "this" block to the cache. check it */
242:     if (find_buffer(dev,block))
243:         goto repeat;
244: /* OK, FINALLY we know that this buffer is the only one of it's kind, */
245: /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
246:     bh->b_count=1;
247:     bh->b_dirt=0;
248:     bh->b_uptodate=0;
249:     remove_from_queues(bh);
250:     bh->b_dev=dev;
251:     bh->b_blocknr=block;
252:     insert_into_queues(bh);
253:     return bh;
254: } « end getblk »
```

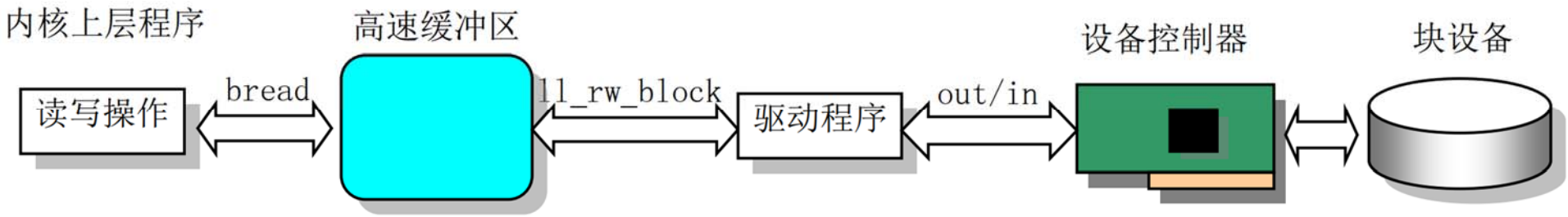


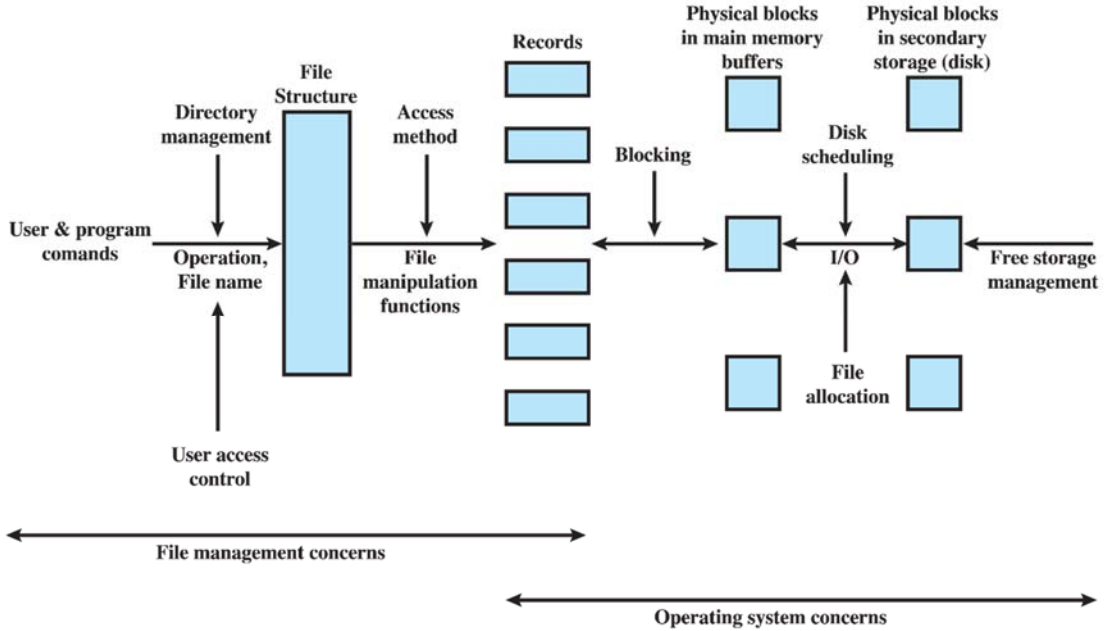
bread()函数

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```



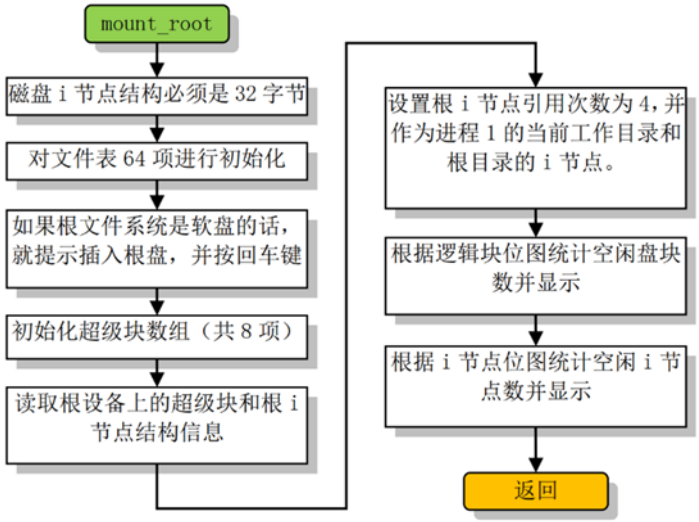
高速缓冲区访问过程和同步操作





mount_root()函数的功能

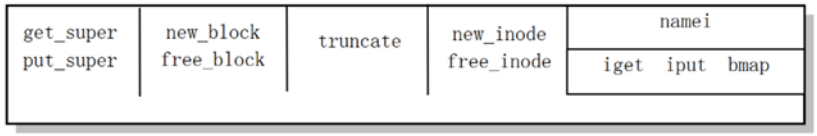
```
242: void mount_root(void)
243: {
244:     int i,free;
245:     struct super_block * p;
246:     struct m_inode * mi;
247:
248:     if (32 != sizeof (struct d_inode))
249:         panic("bad i-node size");
250:     for(i=0;i<NR_FILE;i++)
251:         file_table[i].f_count=0;
252:     if (MAJOR(ROOT_DEV) == 2) {
253:         printk("Insert root floppy and press ENTER");
254:         wait_for_keypress();
255:     }
256:     for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
257:         p->s_dev = 0;
258:         p->s_lock = 0;
259:         p->s_wait = NULL;
260:     }
261:     if (!(p=read_super(ROOT_DEV)))
262:         panic("Unable to mount root");
263:     if (!(mi=iget(ROOT_DEV,ROOT_INO)))
264:         panic("Unable to read root i-node");
265:     mi->i_count += 3 ;  /* NOTE! it is logically used 4 times, not 1 */
266:     p->s_isup = p->s_imount = mi;
267:     current->pwd = mi;
268:     current->root = mi;
269:     free=0;
270:     i=p->s_nzones;
271:     while (-- i >= 0)
272:         if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
273:             free++;
274:     printk("%d/%d free blocks\n\r",free,p->s_nzones);
275:     free=0;
276:     i=p->s_ninodes+1;
277:     while (-- i >= 0)
278:         if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
279:             free++;
280:     printk("%d/%d free inodes\n\r",free,p->s_ninodes);
281: } « end mount_root »
```



用户程序读写操作过程

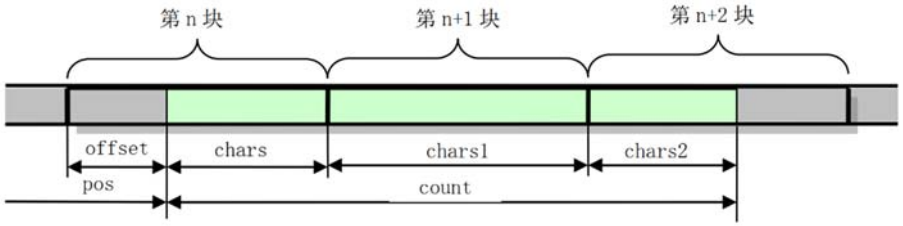
文件系统底层函数

- 包含在以下 5 个文件中
  - bitmap.c
    - 包括对 i 节点位图和逻辑块位图进行释放和占用处理函数
  - truncate.c
  - inode.c
    - 包括分配 i 节点函数 iget()和放回对内存 i 节点存取函数 iput()
    - 根据 i 节点取文件数据块在设备上对应的逻辑块号函数 bmap()
  - namei.c
    - 函数 namei()，该函数使用 iget()、 iput()和 bmap()将给定的文件路径名映射到其 i 节点
  - super.c
    - 专门用于处理文件系统超级块

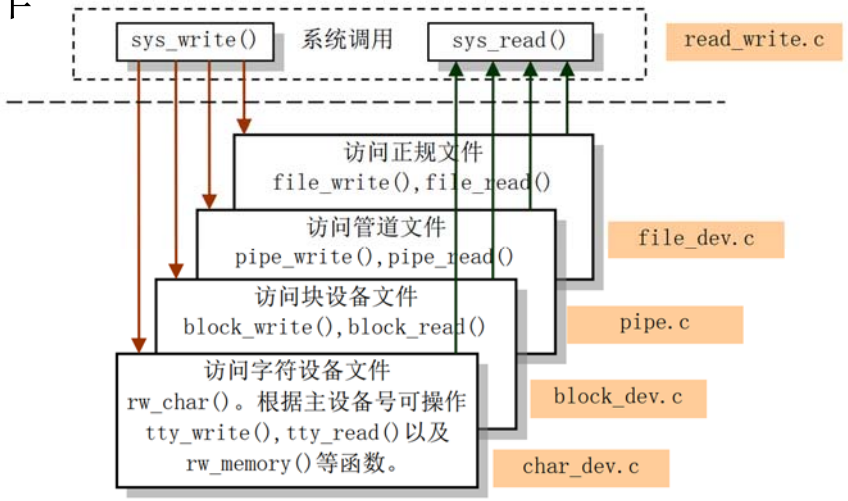


文件中数据的访问操作

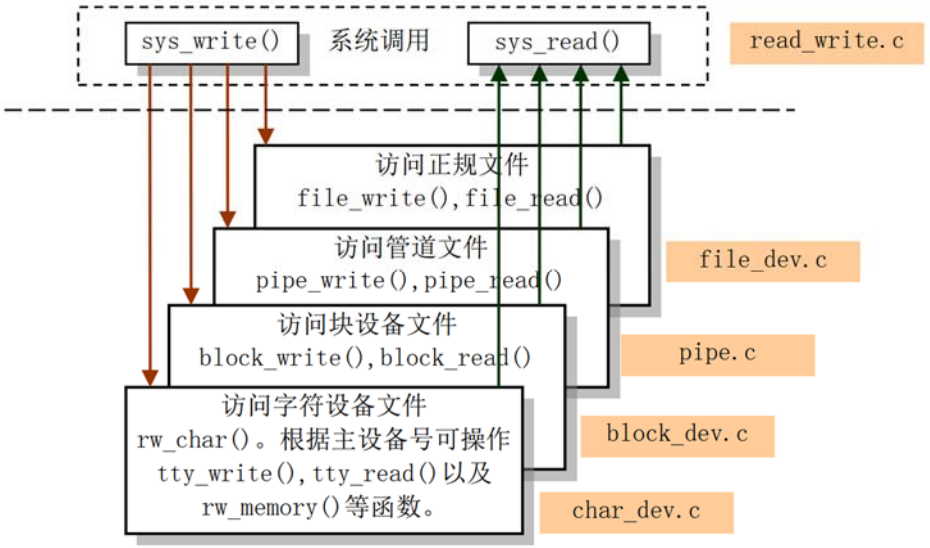
- 文件中数据的访问操作代码，主要涉及 5 个文件
  - block_dev.c、 file_dev.c、char_dev.c、 pipe.c 和 read_write.c
  - 前 4 个文件可以认为是块设备、字符设备、管道设备和普通文件与文件读写系统调用的接口程序
    - 它们共同实现了 read_write.c 中的 read()和 write()系统调用
    - 通过对被操作文件属性的判断，这两个系统调用会分别调用这些文件中的相关处理函数进行操作



块数据读写操作指针位置

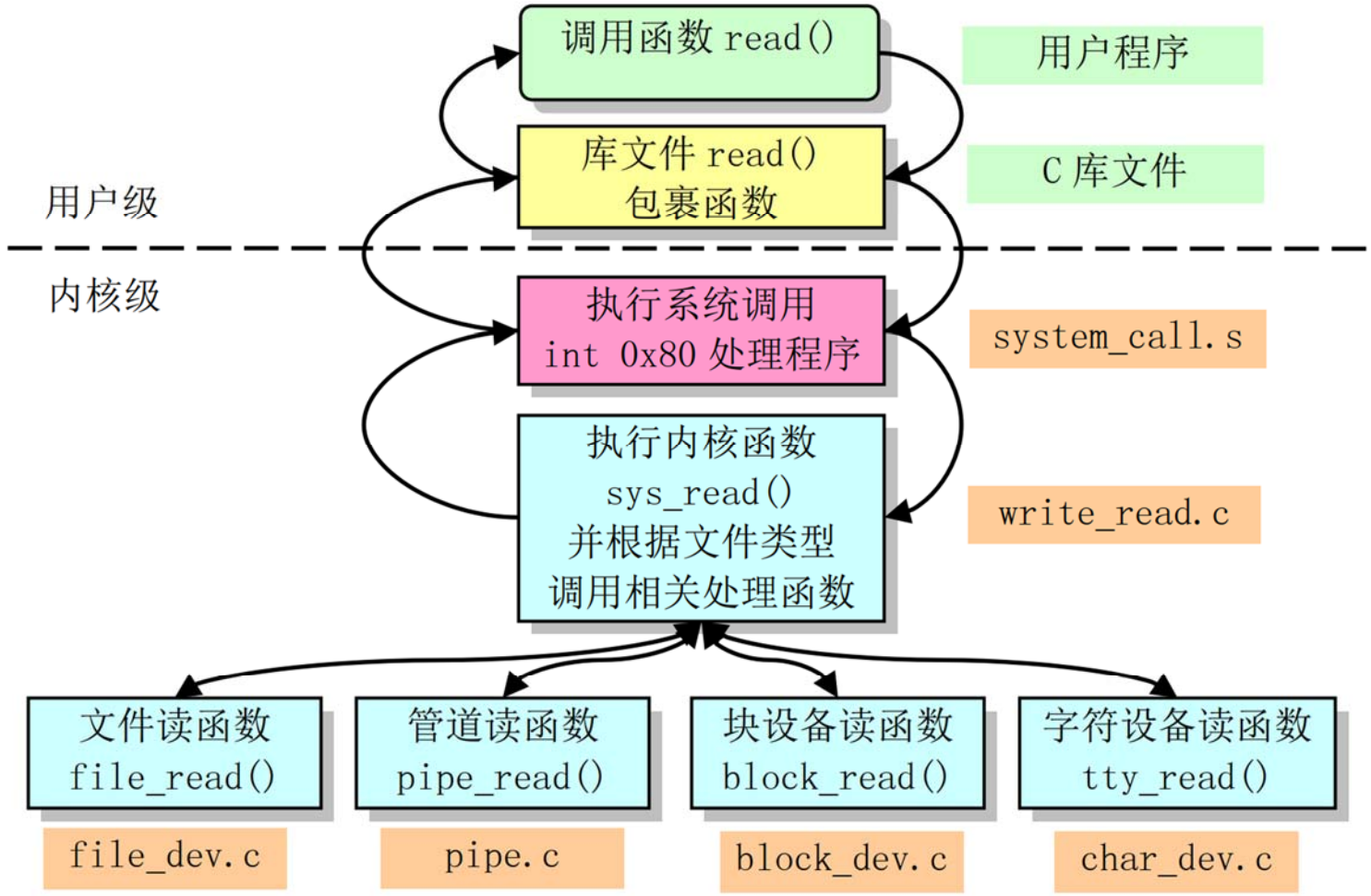


```
83: int sys_write(unsigned int fd,char * buf,int count)
84: {
85:     struct file * file;
86:     struct m_inode * inode;
87:
88:     if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
89:         return -EINVAL;
90:     if (!count)
91:         return 0;
92:     inode=file->f_inode;
93:     if (inode->i_pipe)
94:         return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
95:     if (S_ISCHR(inode->i_mode))
96:         return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
97:     if (S_ISBLK(inode->i_mode))
98:         return block_write(inode->i_zone[0],&file->f_pos,buf,count);
99:     if (S_ISREG(inode->i_mode))
100:        return file_write(inode,file,buf,count);
101:    printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
102:    return -EINVAL;
103: } « end sys_write »
```

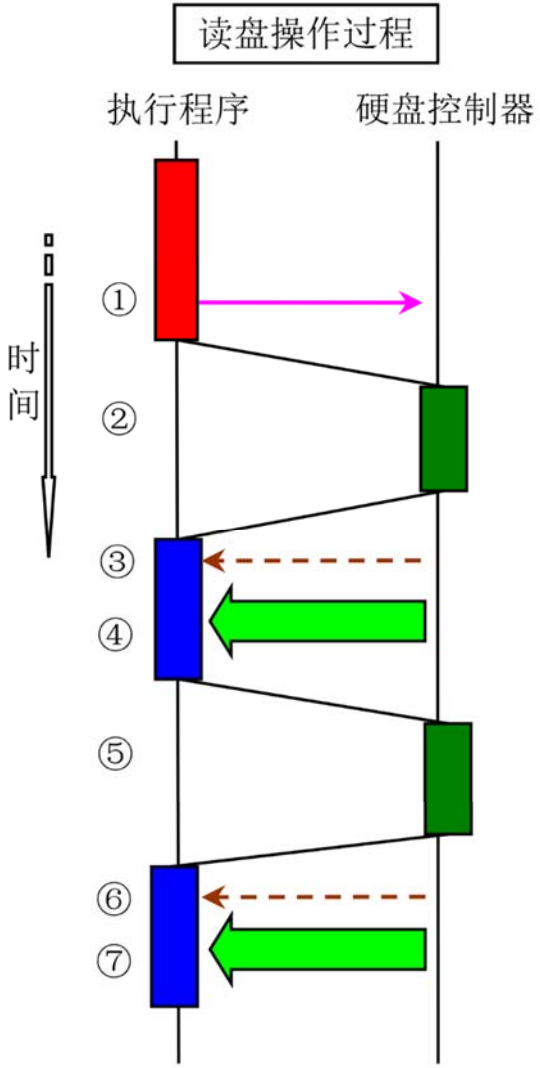


	字段名称	数据类型	说明
在盘上和内存中的字段，共32字节	i_mode	short	文件的类型和属性（rwx 位）
	i_uid	short	文件宿主的用户 id
	i_size	long	文件长度（字节）
	i_mtime	long	修改时间（从 1970.1.1:0 时算起，秒）
	i_gid	char	文件宿主的组 id
	i_nlinks	char	链接数（有多少个文件目录项指向该 i 节点）
	i_zone[9]	short	文件所占用的盘上逻辑块号数组。其中： zone[0]-zone[6]是直接块号； zone[7]是一次间接块号； zone[8]是二次（双重）间接块号。 注：zone 是区的意思，可译成区块或逻辑块。 对于设备特殊文件名的 i 节点，其 zone[0]中存放的是该文件名所指设备的设备号。
仅在内存中使用的字段	i_wait	task_struct *	等待该 i 节点的进程。
	i_atime	long	最后访问时间。
	i_ctime	long	i 节点自身被修改时间。
	i_dev	short	i 节点所在的设备号。
	i_num	short	i 节点号。
	i_count	short	i 节点被引用的次数，0 表示空闲。
	i_lock	char	i 节点被锁定标志。
	i_dirt	char	i 节点已被修改（脏）标志。
	i_pipe	char	i 节点用作管道标志。
	i_mount	char	i 节点安装了其他文件系统标志。
	i_seek	char	搜索标志（lseek 操作时）。
	i_update	char	i 节点已更新标志。

read()函数调用执行过程



读硬盘数据操作的时序关系



处理请求项　　写命令
中断处理过程　　读状态
控制器处理　　传读数据

① 函数 do_hd_request()中用 hd_out()发送读命令，然后退出返回调用程序执行，同时等待控制器发中断；

② 控制器接到读命令后就执行从硬盘读数据到控制器缓冲区中的操作。操作完后就发中断；

③ 在中断处理程序中执行 read_intr()。读取控制器状态；

④ 并从控制器缓冲区读取 1 个扇区数据到请求项缓冲块中；

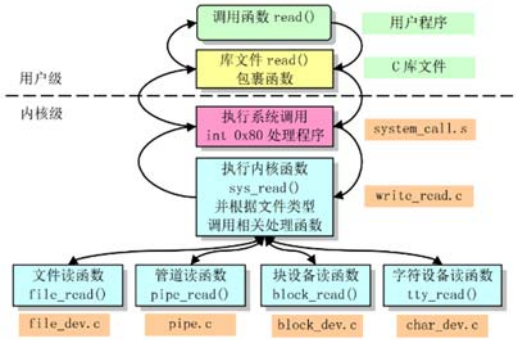
⑤ 控制器随后再执行从硬盘读下一扇区数据到控制器缓冲区中的操作。操作完后就发中断；

⑥ 在中断处理程序中执行 read_intr()。读取控制器状态；

⑦ 并从控制器缓冲区读取 1 扇区数据到请求项缓冲块中，若数据已全部读出则处理本请求项结束事宜，并执行其他硬盘请求操作。

```
74: fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
75: sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
76: sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
77: sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
78: sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
79: sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
80: sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
81: sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
82: sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
83: sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
84: sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
85: sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
86: sys_setreuid,sys_setregid };
```

```
55: int sys_read(unsigned int fd,char * buf,int count)
56: {
57:     struct file * file;
58:     struct m_inode * inode;
59:
60:     if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
61:         return -EINVAL;
62:     if (!count)
63:         return 0;
64:     verify_area(buf,count);
65:     inode = file->f_inode;
66:     if (inode->i_pipe)
67:         return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
68:     if (S_ISCHR(inode->i_mode))
69:         return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
70:     if (S_ISBLK(inode->i_mode))
71:         return block_read(inode->i_zone[0],&file->f_pos,buf,count);
72:     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73:         if (count+file->f_pos > inode->i_size)
74:             count = inode->i_size - file->f_pos;
75:         if (count<=0)
76:             return 0;
77:         return file_read(inode,file,buf,count);
78:     }
79:     printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
80:     return -EINVAL;
81: } « end sys_read »
```




```
17: int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18: {
19:     int left,chars,nr;
20:     struct buffer_head * bh;
21:
22:     if ((left=count)<=0)
23:         return 0;
24:     while (left) {
25:         if ((nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE))) {
26:             if (!(bh=bread(inode->i_dev,nr)))
27:                 break;
28:         } else
29:             bh = NULL;
30:         nr = filp->f_pos % BLOCK_SIZE;
31:         chars = MIN( BLOCK_SIZE-nr , left );
32:         filp->f_pos += chars;
33:         left -= chars;
34:         if (bh) {
35:             char * p = nr + bh->b_data;
36:             while (chars-->0)
37:                 put_fs_byte(*(p++),buf++);
38:             brelse(bh);
39:         } else {
40:             while (chars-->0)
41:                 put_fs_byte(0,buf++);
42:         }
43:     } « end while left »
44:     inode->i_atime = CURRENT_TIME;
45:     return (count-left)?(count-left):-ERROR;
46: } « end file_read »
```

```
11: /* devices are as follows: (same as minix, so we can use the minix
12:  * file system. These are major numbers.)
13:  *
14:  * 0 - unused (nodev)
15:  * 1 - /dev/mem
16:  * 2 - /dev/fd
17:  * 3 - /dev/hd
18:  * 4 - /dev/ttyx
19:  * 5 - /dev/tty
20:  * 6 - /dev/lp
21:  * 7 - unnamed pipes
```

do_hd_request ( )

```
78: #elif (MAJOR_NR == 3)
79: /* harddisk */
80: #define DEVICE_NAME "harddisk"
81: #define DEVICE_INTR do_hd
82: #define DEVICE_REQUEST do_hd_request
83: #define DEVICE_NR(device) (MINOR(device)/5)
84: #define DEVICE_ON(device)
85: #define DEVICE_OFF(device)
86:
87: #elif
88: /* unknown blk device */
89: #error "unknown blk device"
90:
91: #endif
92:
93: #define CURRENT (blk_dev[MAJOR_NR].current_request)
94: #define CURRENT_DEV DEVICE_NR(CURRENT->dev)
```

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```


```
145: void ll_rw_block(int rw, struct buffer_head * bh)
146: {
147:     unsigned int major;
148:
149:     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
150:     !(blk_dev[major].request_fn)) {
151:         printk("Trying to read nonexistent block-device\n\r");
152:         return;
153:     }
154:     make_request(major,rw,bh);
155: }
```

```
88: static void make_request(int major,int rw, struct buffer_head * bh)
89: {
90:     struct request * req;
91:     int rw_ahead;
92:
93: /* WRITEA/READA is special case - it is not really needed, so if the */
94: /* buffer is locked, we just forget about it, else it's a normal read */
95:     if ((rw_ahead = (rw == READA || rw == WRITEA))) {
96:         if (bh->b_lock)
97:             return;
98:         if (rw == READA)
99:             rw = READ;
100:        else
101:            rw = WRITE;
102:    }
103:    if (rw!=READ && rw!=WRITE)
104:        panic("Bad block dev command, must be R/W/RA/WA");
105:    lock_buffer(bh);
106:    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
107:        unlock_buffer(bh);
108:        return;
109:    }
110: repeat:
111: /* we don't allow the write-requests to fill up the queue completely:
112:  * we want some room for reads: they take precedence. The last third
113:  * of the requests are only for reads.
114:  */
115:    if (rw == READ)
116:        req = request+NR_REQUEST;
117:    else
118:        req = request+((NR_REQUEST*2)/3);
119: /* find an empty request */
120:    while (--req >= request)
121:        if (req->dev<0)
122:            break;
123: /* if none found, sleep on new requests: check for rw_ahead */
124:    if (req < request) {
125:        if (rw_ahead) {
126:            unlock_buffer(bh);
127:            return;
128:        }
129:        sleep_on(&wait_for_request);
130:        goto repeat;
131:    }
132: /* fill up the request-info, and add it to the queue */
133:    req->dev = bh->b_dev;
134:    req->cmd = rw;
135:    req->errors=0;
136:    req->sector = bh->b_blocknr<<1;
137:    req->nr_sectors = 2;
138:    req->buffer = bh->b_data;
139:    req->waiting = NULL;
140:    req->bh = bh;
141:    req->next = NULL;
142:    add_request(major+blk_dev,req);
143: } « end make_request »
```

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```

```
39: static inline void wait_on_buffer(struct buffer_head * bh)
40: {
41:     cli();
42:     while (bh->b_lock)
43:         sleep_on(&bh->b_wait);
44:     sti();
45: }
46:
```

sleep_on ( ) -> Schedule( )

```
145: void ll_rw_block(int rw, struct buffer_head * bh)
146: {
147:     unsigned int major;
148:
149:     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
150:     !(blk_dev[major].request_fn)) {
151:         printk("Trying to read nonexistent block-device\n\r");
152:         return;
153:     }
154:     make_request(major,rw,bh);
155: }
```

```
54: static void add_request(struct blk_dev_struct * dev, struct request * req)
55: {
56:     struct request * tmp;
57:
58:     req->next = NULL;
59:     cli();
70:     if (req->bh)
71:         req->bh->b_dirt = 0;
72:     if (!(tmp = dev->current_request)) {
73:         dev->current_request = req;
74:         sti();
75:         (dev->request_fn)();
76:         return;
77:     }
78:     for ( ; tmp->next ; tmp=tmp->next)
79:         if ((IN_ORDER(tmp,req) ||
80:             !IN_ORDER(tmp,tmp->next)) &&
81:             IN_ORDER(req,tmp->next))
82:             break;
83:     req->next=tmp->next;
84:     tmp->next=req;
85:     sti();
} « end add_request »
```

电梯算法

一路返回到
wait_on_buffer

```c
void ll_rw_block(int rw, struct buffer_head * bh)
{
    unsigned int major;

    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);
}
```

do_hd_request ( )

```c
void do_hd_request(void)
{
    int i,r = 0;
    unsigned int block,dev;
    unsigned int sec,head,cyl;
    unsigned int nsect;

    INIT_REQUEST;
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
        end_request(0);
        goto repeat;
    }
    block += hd[dev].start_sect;
    dev /= 5;
    __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++;
    nsect = CURRENT->nr_sectors;
    if (reset) {
        reset = 0;
        recalibrate = 1;
        reset_hd(CURRENT_DEV);
        return;
    }
    if (recalibrate) {
        recalibrate = 0;
        hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
            WIN_RESTORE,&recal_intr);
        return;
    }
    if (CURRENT->cmd == WRITE) {
        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
        for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
            /* nothing */ ;
        if (!r) {
            bad_rw_intr();
            goto repeat;
        }
        port_write(HD_DATA,CURRENT->buffer,256);
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    } else
        panic("unknown hd-command");
} « end do_hd_request »
```

read_intr ( )

硬盘信息相关，
端口读写

```c
#define outb(value,port) \
__asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))

#define inb(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
_v; \
})

#define outb_p(value,port) \
__asm__ ("outb %%al,%%dx\n" \
        "\tjmp 1f\n" \
        "1:\tjmp 1f\n" \
        "1:"::"a" (value),"d" (port))

#define inb_p(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al\n" \
    "\tjmp 1f\n" \
    "1:\tjmp 1f\n" \
    "1:":"=a" (_v):"d" (port)); \
_v; \
})
```

```c
struct hd_i_struct {
    int head,sect,cyl,wpcom,lzone,ctl;
    };
#ifdef HD_TYPE
struct hd_i_struct hd_info[] = { HD_TYPE };
#define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))
#else
struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
static int NR_HD = 0;
#endif

static struct hd_struct {
    long start_sect;
    long nr_sects;
} hd[5*MAX_HD]={{0,0},};

#define port_read(port,buf,nr) \
__asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr))
```

硬盘信息相关，
端口读写

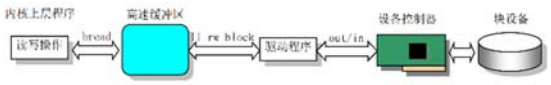
```c
static inline void end_request(int uptodate)
{
    DEVICE_OFF(CURRENT->dev);
    if (CURRENT->bh) {
        CURRENT->bh->b_uptodate = uptodate;
        unlock_buffer(CURRENT->bh);
    }
    if (!uptodate) {
        printk(DEVICE_NAME " I/O error\n\r");
        printk("dev %04x, block %d\n\r",CURRENT->dev,
            CURRENT->bh->b_blocknr);
    }
    wake_up(&CURRENT->waiting);
    wake_up(&wait_for_request);
    CURRENT->dev = -1;
    CURRENT = CURRENT->next;
}
```

wake_up

```c
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
        unsigned int head,unsigned int cyl,unsigned int cmd,
        void (*intr_addr)(void))
{
    register int port asm("dx");

    if (drive>1 || head>15)
        panic("Trying to write bad sector");
    if (!controller_ready())
        panic("HD controller not ready");
    do_hd = intr_addr;
    outb_p(hd_info[drive].ctl,HD_CMD);
    port=HD_DATA;
    outb_p(hd_info[drive].wpcom>>2,++port);
    outb_p(nsect,++port);
    outb_p(sect,++port);
    outb_p(cyl,++port);
    outb_p(cyl>>8,++port);
    outb_p(0xA0|(drive<<4)|head,++port);
    outb(cmd,++port);
} « end hd_out »
```

```c
static void read_intr(void)
{
    if (win_result()) {
        bad_rw_intr();
        do_hd_request();
        return;
    }
    port_read(HD_DATA,CURRENT->buffer,256);
    CURRENT->errors = 0;
    CURRENT->buffer += 512;
    CURRENT->sector++;
    if (--CURRENT->nr_sectors) {
        do_hd = &read_intr;
        return;
    }
    end_request(1);
    do_hd_request();
}
```



理解硬盘驱动程序的操作过程

- 理解硬盘驱动程序的操作过程
  - hd_out()、 do_hd_request()、 read_intr()、write_intr()
- 在使用 **hd_out()**向硬盘控制器发送了读/写命令后
  - hd_out()函数并不会等待所发命令的执行过程，而是立刻返回到调用它的程序中，例如 do_hd_request()
  - 而 do_hd_request()函数也会立刻返回上一级调用它的函数（ add_request()）
  - 最终返回到调用块设备读写函数 ll_rw_block()的其他程序，例如 fs/buffer.c 的 bread()函数中，去等待块设备 IO 的完成