

CS353 Linux Kernel

Chentao Wu 吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

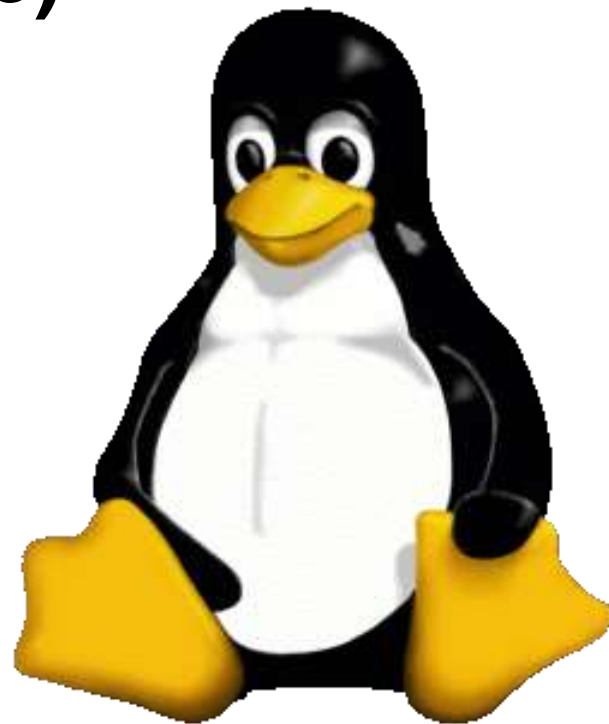


上海交通大学

7B. Linux File System

(Reading Source Code)

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn



上海交通大学

Linux File System

- Minix File System
 - **Linux Kernel 0.11**
- Extended File System (Ext)
- **Second Extended FileSystem (Ext2)**
 - **Linux Kernel 2.4**
- Third Extended FileSystem (Ext3)
- Fourth Extended FileSystem (Ext4)

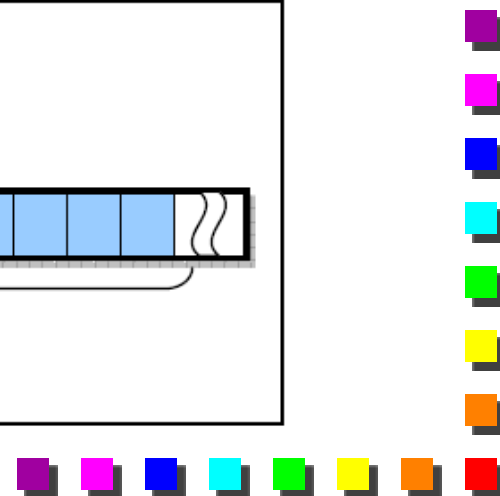
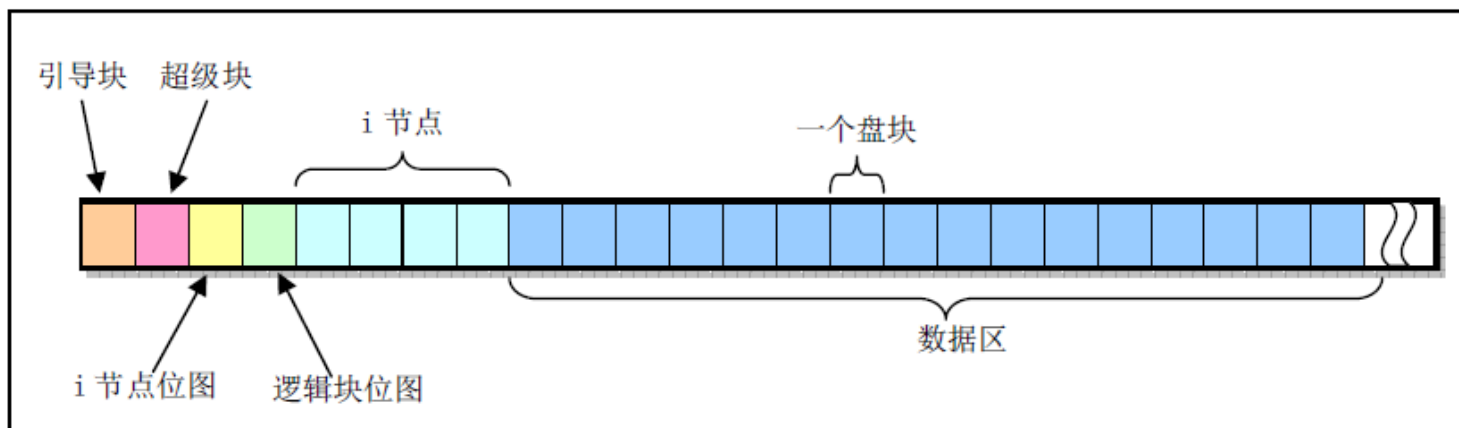


硬盘上的分区和文件系统



















- 图中表示4个分区，分别存放：FAT32、NTFS、MINIX和EXT2文件系统。

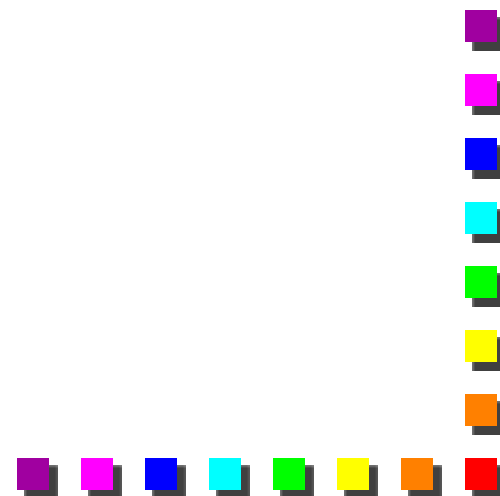


- MINIX文件系统



Linux Kernel 0.11 (Minix File System) Linux/fs 目录

名称	大小	最后修改时间 (GMT)	说明
 Makefile	5053 bytes	1991-12-02 03:21:31	m
 bitmap.c	4042 bytes	1991-11-26 21:31:53	m
 block_dev.c	1422 bytes	1991-10-31 17:19:55	m
 buffer.c	9072 bytes	1991-12-06 20:21:00	m
 char_dev.c	2103 bytes	1991-11-19 09:10:22	m
 exec.c	9134 bytes	1991-12-01 20:01:01	m
 fcntl.c	1455 bytes	1991-10-02 14:16:29	m
 file_dev.c	1852 bytes	1991-12-01 19:02:43	m
 file_table.c	122 bytes	1991-10-02 14:16:29	m
 inode.c	6933 bytes	1991-12-06 20:16:35	m
 ioctl.c	977 bytes	1991-11-19 09:13:05	
 namei.c	16562 bytes	1991-11-25 19:19:59	m
 open.c	4340 bytes	1991-11-25 19:21:01	m
 pipe.c	2385 bytes	1991-10-18 19:02:33	m
 read_write.c	2802 bytes	1991-11-25 15:47:20	m
 stat.c	1175 bytes	1991-10-02 14:16:29	m
 super.c	5628 bytes	1991-12-06 20:10:12	m
 truncate.c	1148 bytes	1991-10-02 14:16:29	m



Super Block Data Structure in Minix File System

	字段名称	数据类型	说明
出现在盘上和内存中的字段	s_ninodes	short	i 节点数
	s_nzones	short	逻辑块数(或称为区块数)
	s_imap_blocks	short	i 节点位图所占块数
	s_zmap_blocks	short	逻辑块位图所占块数
	s_firstdatazone	short	数据区中第一个逻辑块块号
	s_log_zone_size	short	\log_2 (磁盘块数/逻辑块)
	s_max_size	long	最大文件长度
	s_magic	short	文件系统幻数(0x137f)
仅在内存中使用的字段	s_imap[8]	buffer_head *	i 节点位图在高速缓冲块指针数组
	s_zmap[8]	buffer_head *	逻辑块位图在高速缓冲块指针数组
	s_dev	short	超级块所在设备号
	s_isup	m_inode *	被安装文件系统根目录 i 节点
	s_imount	m_inode *	该文件系统被安装到的 i 节点
	s_time	long	修改时间
	s_wait	task_struct *	等待本超级块的进程指针
	s_lock	char	锁定标志
	s_rd_only	char	只读标志
	s_dirt	char	已被修改(脏)标志



Inode Data Structure in Minix File System

字段名称	数据类型	说明
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。
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。

在盘上和内存中的
字段, 共 32
字节

仅在内存中使用的
字段

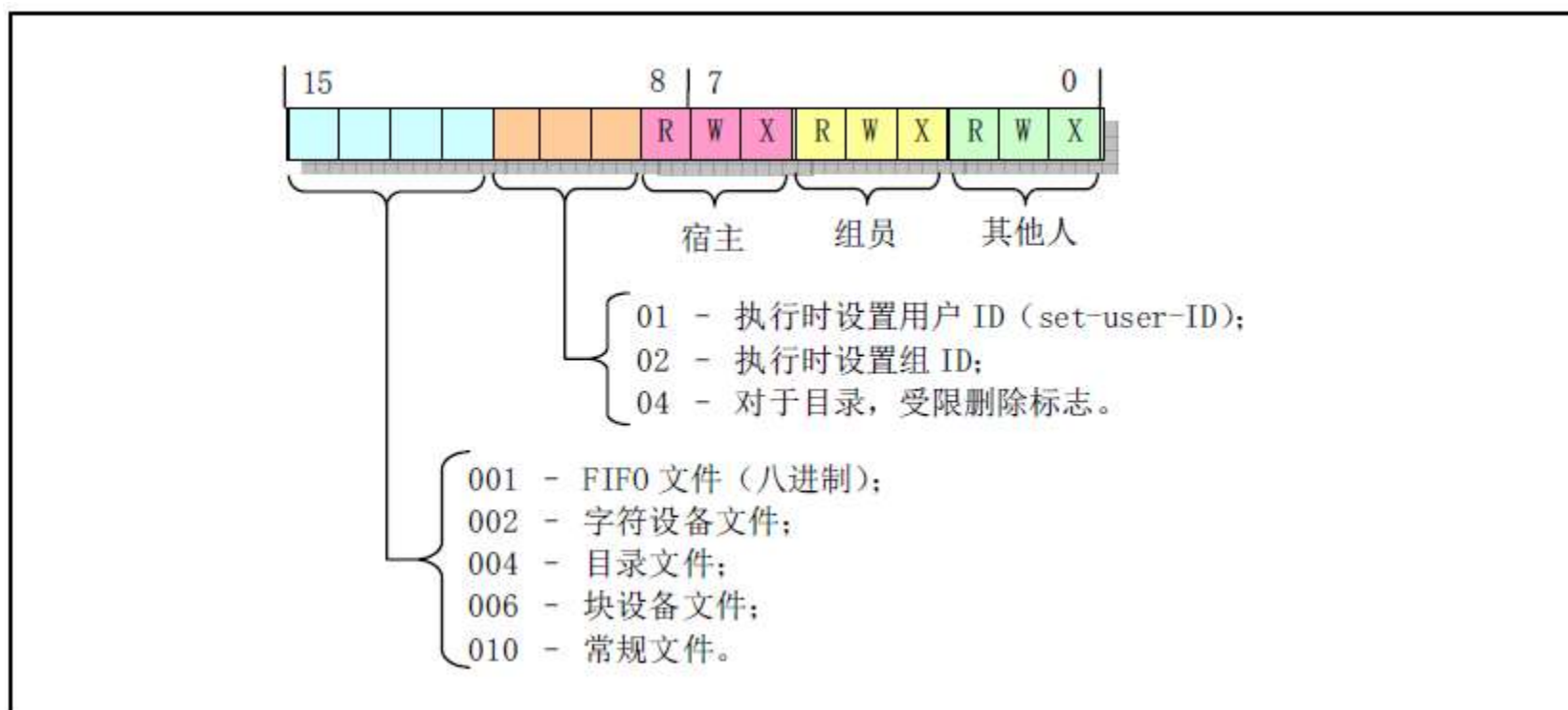


上海交通大学



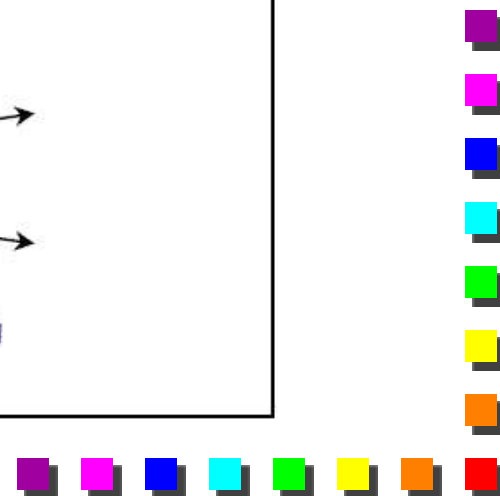
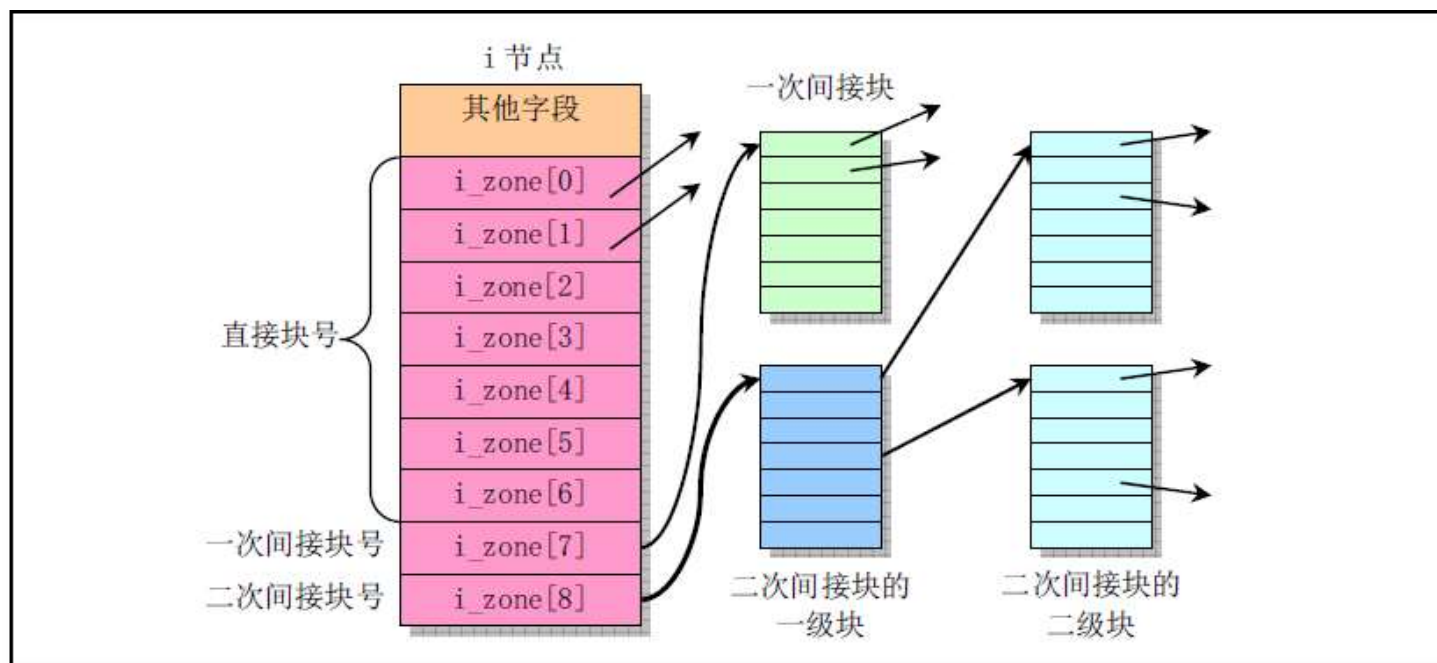
i_mode字段

- 保存文件的类型和访问权限属性

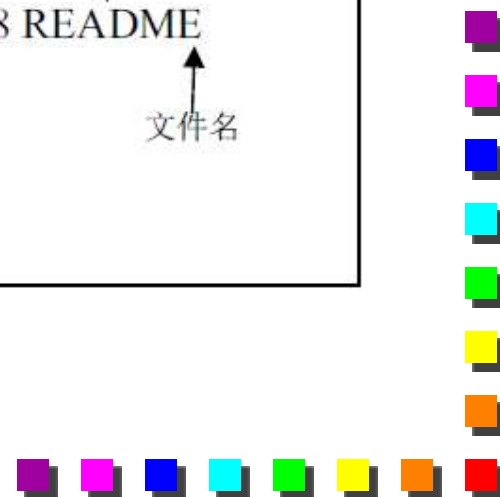
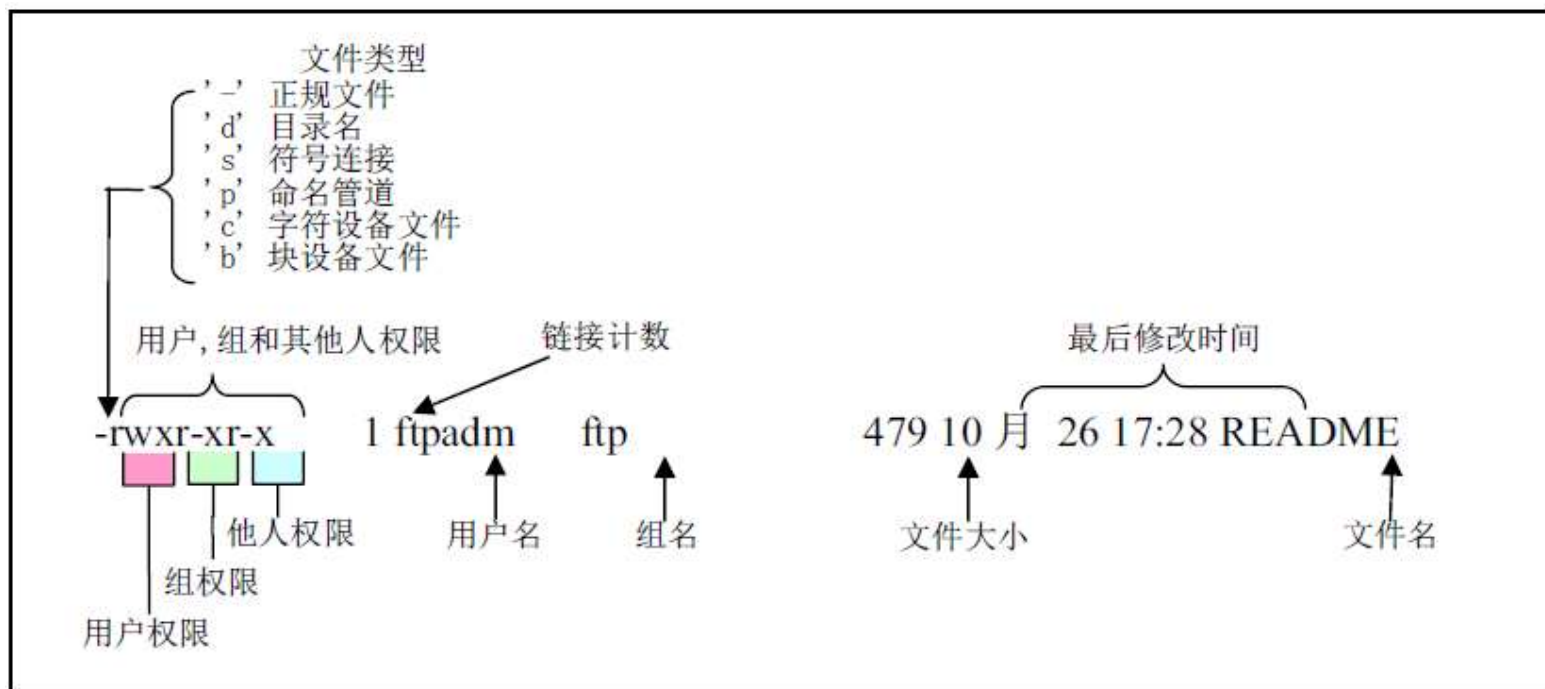


逻辑块数组i_zone

- i_zone[0]- i_zone[6]存放文件开始的7个磁盘块号，称为直接块
- i_zone[7]（一次间接块）存放512个磁盘块号，可以寻址512个磁盘块
- i_zone[8]（二次间接块）存在 512×512 个磁盘块号，可以寻址 512×512 个磁盘块



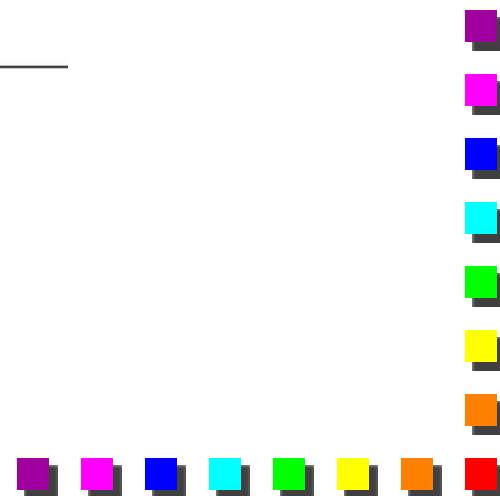
“Ls-l” 显示的文件信息



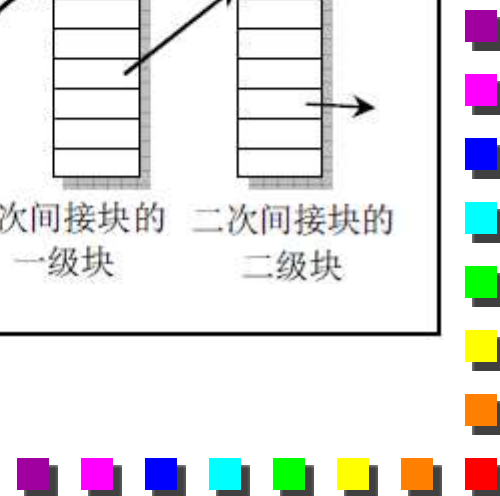
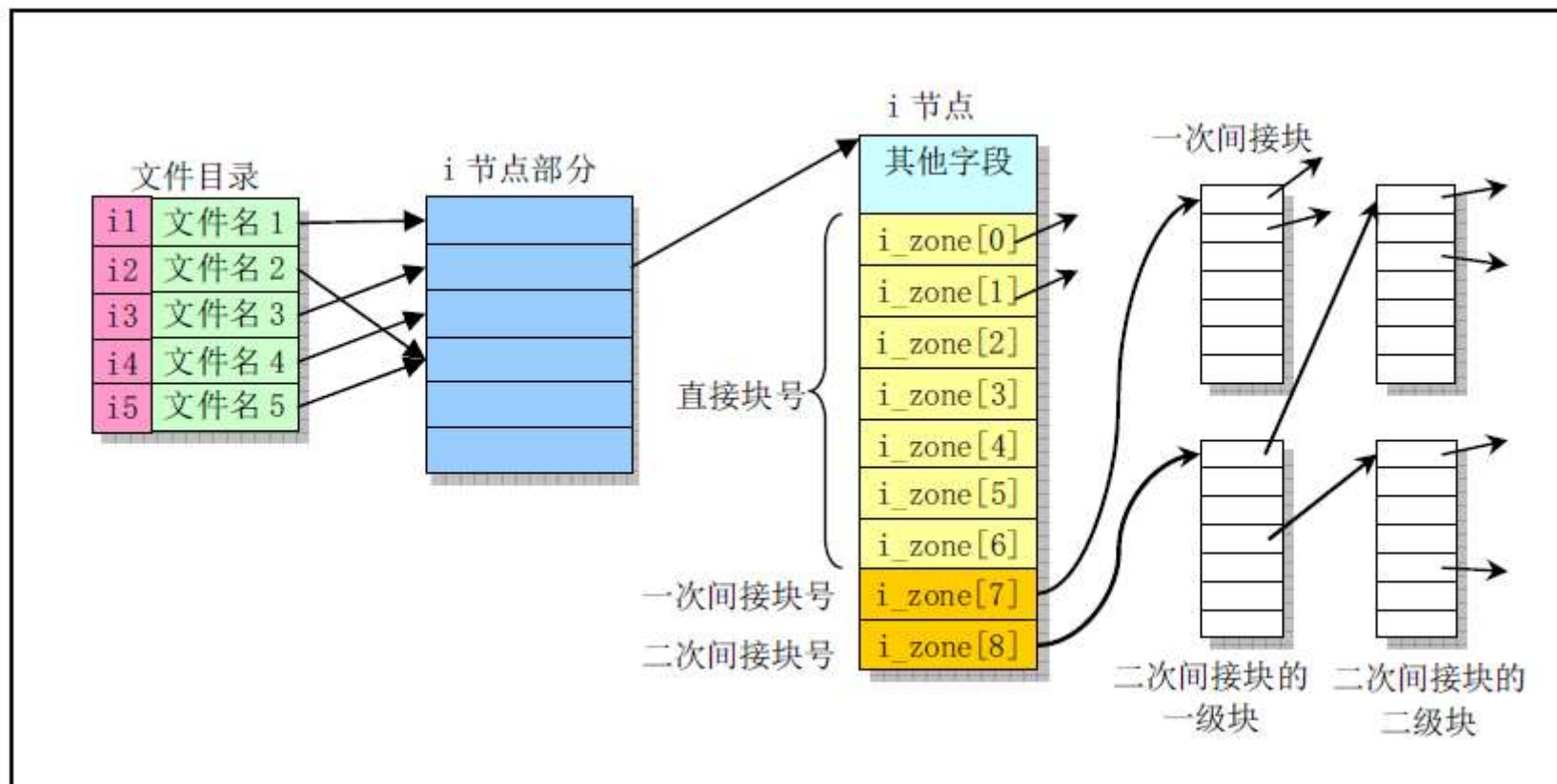
Dentry Data Structure in Minix File System

```
// 定义在 include/linux/fs.h 文件中。
#define NAME_LEN 14                // 名字长度值。
#define ROOT_INO 1                // 根 i 节点。

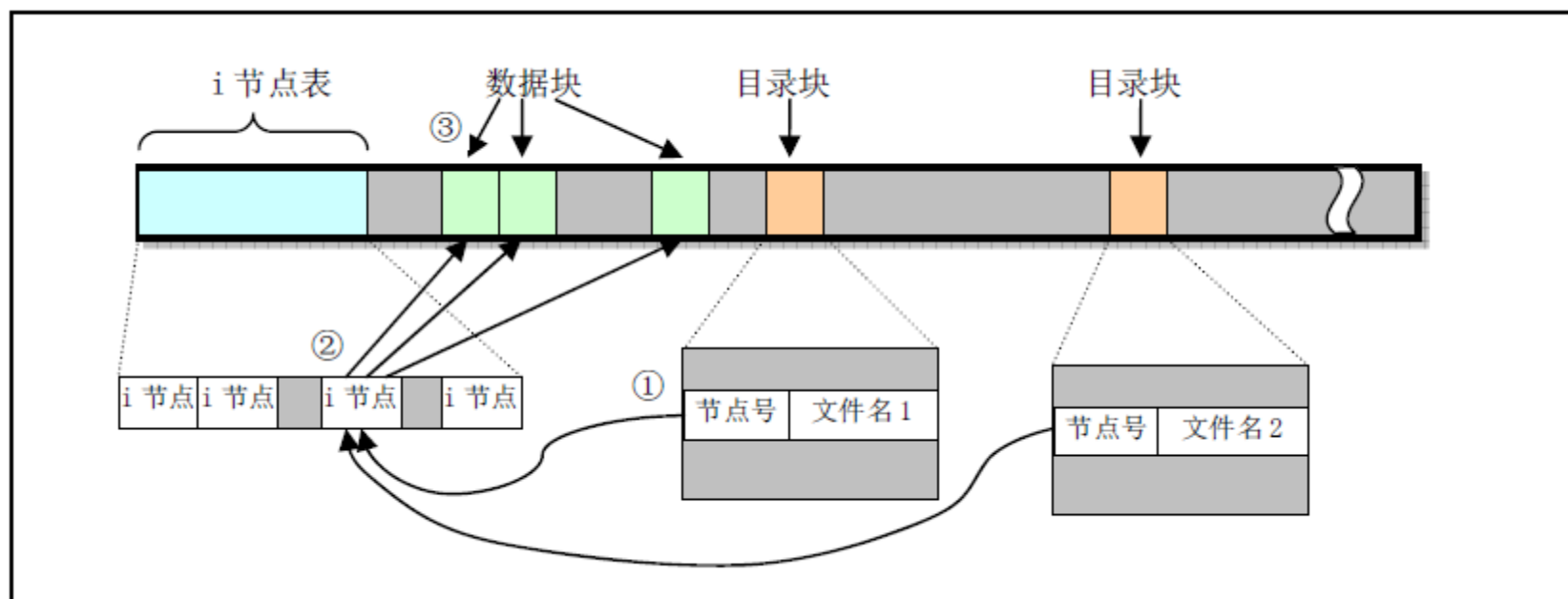
// 文件目录项结构。
struct dir_entry {
    unsigned short inode;           // i 节点号。
    char name[NAME_LEN];           // 文件名。
};
```



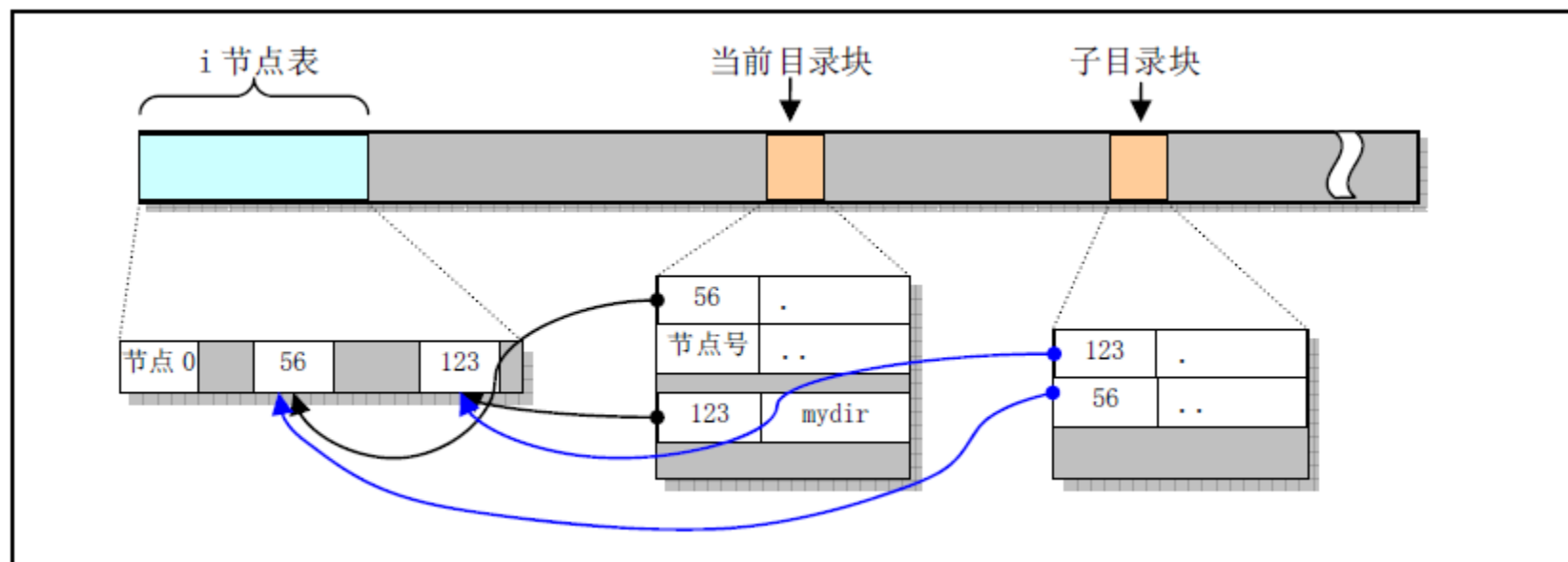
通过文件名查找对应文件磁盘块的位置



从文件名获取其数据块



文件目录项与子目录的链接



Hexdump查看目录项数据块

```
[/usr/root]# cd /
[/]# ls -la
total 10
drwxr-xr-x 10 root    root      176 Mar 21  2004 .
drwxr-xr-x 10 root    4096      176 Mar 21  2004 ..
drwxr-xr-x  2 root    4096      912 Mar 21  2004 bin
drwxr-xr-x  2 root    root      336 Mar 21  2004 dev
drwxr-xr-x  2 root    root      224 Mar 21  2004 etc
drwxr-xr-x  8 root    root      128 Mar 21  2004 image
drwxr-xr-x  2 root    root        32 Mar 21  2004 mnt
drwxr-xr-x  2 root    root        64 Mar 21  2004 tmp
drwxr-xr-x 10 root    root      192 Mar 29  2004 usr
drwxr-xr-x  2 root    root       32 Mar 21  2004 var

[/]# hexdump .
00000000 0001 002e 0000 0000 0000 0000 0000 0000 0000 // .
00000010 0001 2e2e 0000 0000 0000 0000 0000 0000 0000 // ..
00000020 0002 6962 006e 0000 0000 0000 0000 0000 0000 // bin
00000030 0003 6564 0076 0000 0000 0000 0000 0000 0000 // dev
00000040 0004 7465 0063 0000 0000 0000 0000 0000 0000 // etc
00000050 0005 7375 0072 0000 0000 0000 0000 0000 0000 // usr
00000060 0115 6e6d 0074 0000 0000 0000 0000 0000 0000 // mnt
00000070 0036 6d74 0070 0000 0000 0000 0000 0000 0000 // tmp
00000080 0000 6962 2e6e 656e 0077 0000 0000 0000 0000 // 空闲, 未使用。
00000090 0052 6d69 6761 0065 0000 0000 0000 0000 0000 // image
000000a0 007b 6176 0072 0000 0000 0000 0000 0000 0000 // var
000000b0

[/]#
```



上海交通大学



Linux文件系统底层函数

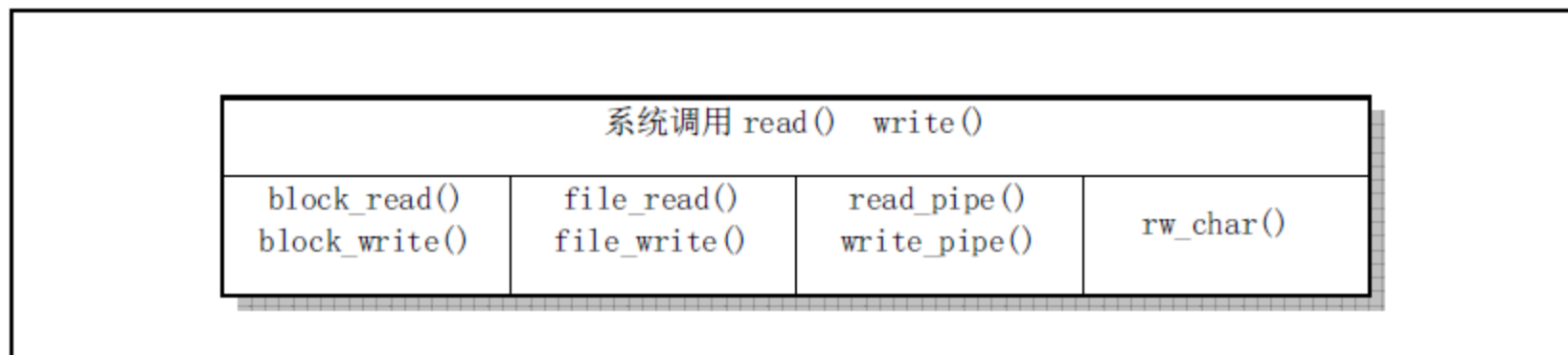
- bitmap.c 程序包括对 i 节点位图和逻辑块位图进行释放和占用处理函数。操作 i 节点位图的函数是 free_inode() 和 new_inode(), 操作逻辑块位图的函数是 free_block() 和 new_block()。
- truncate.c 程序包括对数据文件长度截断为 0 的函数 truncate()。它将 i 节点指定的设备上文件长度截为 0, 并释放文件数据占用的设备逻辑块。
- inode.c 程序包括分配 i 节点函数 iget() 和放回对内存 i 节点存取函数 iput() 以及根据 i 节点信息取文件数据块在设备上对应的逻辑块号函数 bmap()。
- namei.c 程序主要包括函数 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 程序专门用于处理文件系统超级块, 包括函数 get_super()、put_super() 和 free_super() 等。还包括几个文件系统加载/卸载处理函数和系统调用, 如 sys_mount() 等。

get_super put_super	new_block free_block	truncate	new_inode free_inode	namei
				iget iput bmap



Linux文件系统数据访问操作（1）

- 5个文件：block_dev.c（块设备），file_dev.c（普通文件），char_dev.c（字符设备），pipe.c（管道设备）和read_write.c（文件读写系统调用）



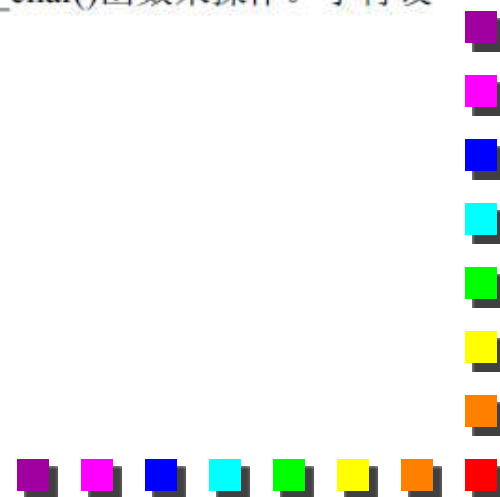
Linux文件系统数据访问操作（2）

block_dev.c 中的函数 `block_read()` 和 `block_write()` 是用于读写块设备特殊文件中的数据。所使用的参数指定了要访问的设备号、读写的起始位置和长度。

file_dev.c 中的 `file_read()` 和 `file_write()` 函数是用于访问一般的正规文件。通过指定文件对应的 i 节点和文件结构，从而可以知道文件所在的设备号和文件当前的读写指针。

pipe.c 文件中实现了管道读写函数 `read_pipe()` 和 `write_pipe()`。另外还实现了创建无名管道的系统调用 `pipe()`。管道主要用于在进程之间按照先进先出的方式传送数据，也可以用于使进程同步执行。有两种类型的管道：有名管道和无名管道。有名管道是使用文件系统的 `open` 调用建立的，而无名管道则使用系统调用 `pipe()` 来创建。在使用管道时，则都用正规文件的 `read()`、`write()` 和 `close()` 函数。只有发出 `pipe` 调用的后代，才能共享对无名管道的存取，而所有进程只要权限许可，都可以访问有名管道。

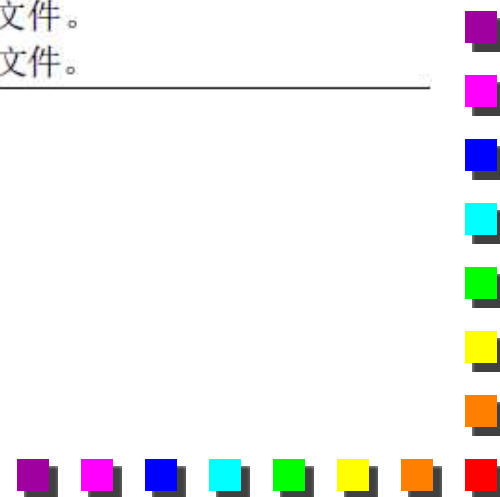
对于字符设备文件，系统调用 `read()` 和 `write()` 会调用 char_dev.c 中的 `rw_char()` 函数来操作。字符设备包括控制台终端（tty）、串口终端（ttyx）和内存字符设备。



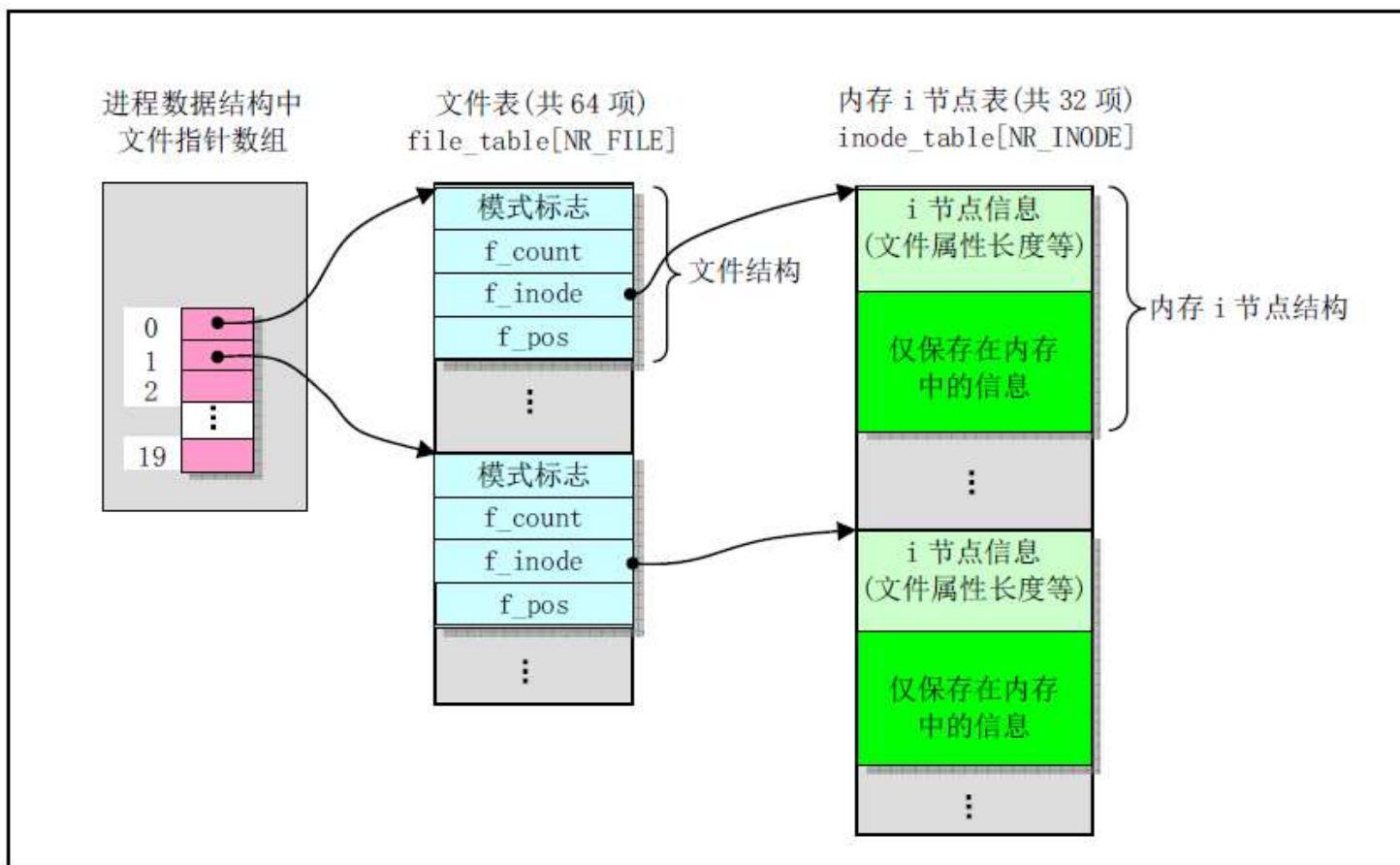
文件访问模式f_mode

// 打开文件 open() 和文件控制函数 fcntl() 使用的文件访问模式。同时只能使用三者之一。

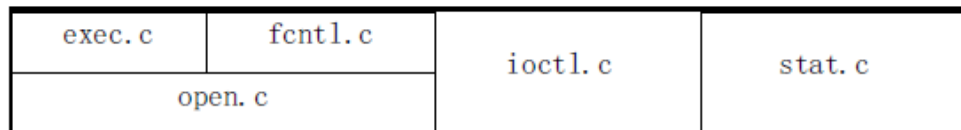
8	#define	<u>O_RDONLY</u>	00	// 以只读方式打开文件。
9	#define	<u>O_WRONLY</u>	01	// 以只写方式打开文件。
10	#define	<u>O_RDWR</u>	02	// 以读写方式打开文件。
// 下面是文件创建和操作标志，用于 open()。可与上面访问模式用'位或'的方式一起使用。				
11	#define	<u>O_CREAT</u>	00100	// 如果文件不存在就创建。fcntl 函数不用。
12	#define	<u>O_EXCL</u>	00200	// 独占使用文件标志。
13	#define	<u>O_NOCTTY</u>	00400	// 不分配控制终端。
14	#define	<u>O_TRUNC</u>	01000	// 若文件已存在且是写操作，则长度截为 0。
15	#define	<u>O_APPEND</u>	02000	// 以添加方式打开，文件指针置为文件尾。
16	#define	<u>O_NONBLOCK</u>	04000	// 非阻塞方式打开和操作文件。
17	#define	<u>O_NDELAY</u>	<u>O_NONBLOCK</u>	// 非阻塞方式打开和操作文件。



进程打开文件使用的内核数据结构



Linux文件系统上层函数 (文件和目录系统调用)



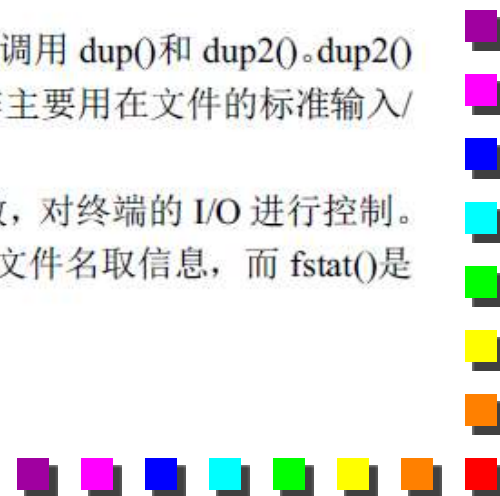
`open.c` 文件用于实现与文件操作相关的系统调用。主要有文件的创建、打开和关闭，文件宿主和属性的修改、文件访问权限的修改、文件操作时间的修改和系统文件系统 `root` 的变动等。

`exec.c` 程序实现对二进制可执行文件和 `shell` 脚本文件的加载与执行。其中主要的函数是函数 `do_execve()`，它是系统中断调用(`int 0x80`)功能号 `__NR_execve()`调用的 C 处理函数，是 `exec()`函数簇的主要实现函数。

`fcntl.c` 实现了文件控制系统调用 `fcntl()`和两个文件句柄(描述符)复制系统调用 `dup()`和 `dup2()`。`dup2()` 指定了新句柄的数值，而 `dup()`则返回当前值最小的未用句柄。句柄复制操作主要用在文件的标准输入/输出重定向和管道操作方面。

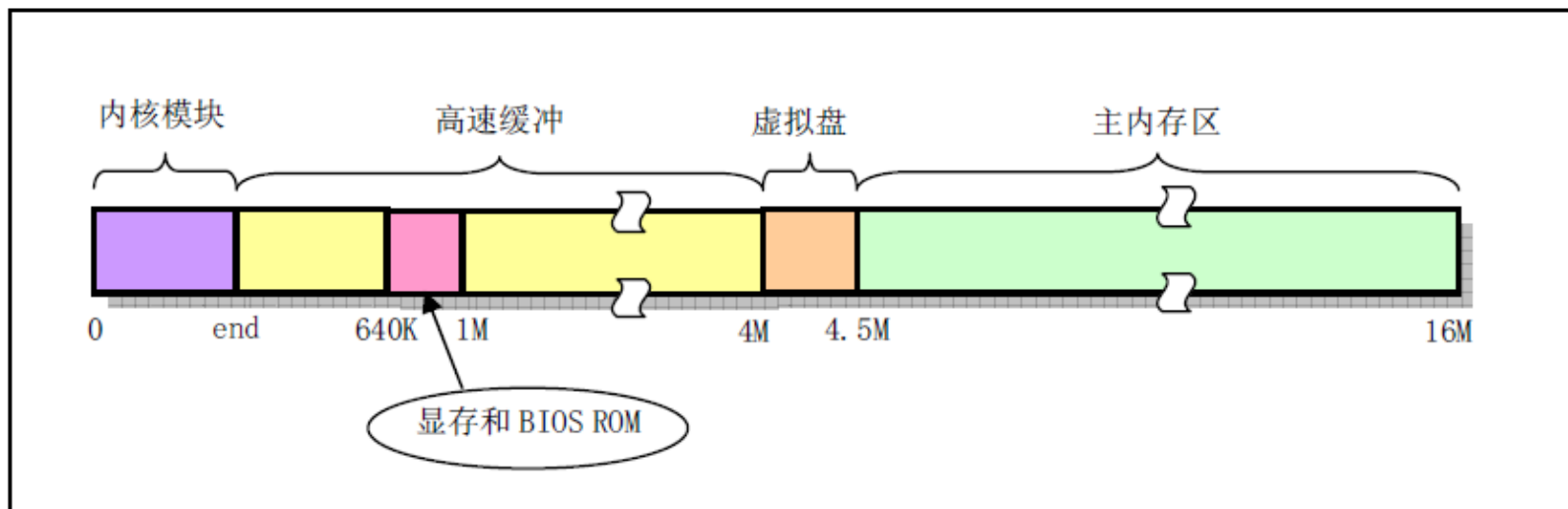
`ioctl.c` 文件实现了输入/输出控制系统调用 `ioctl()`。主要调用 `tty_ioctl()`函数，对终端的 I/O 进行控制。

`stat.c` 文件用于实现取文件状态信息系统调用 `stat()`和 `fstat()`。`stat()`是利用文件名取信息，而 `fstat()`是使用文件句柄(描述符)来取信息。



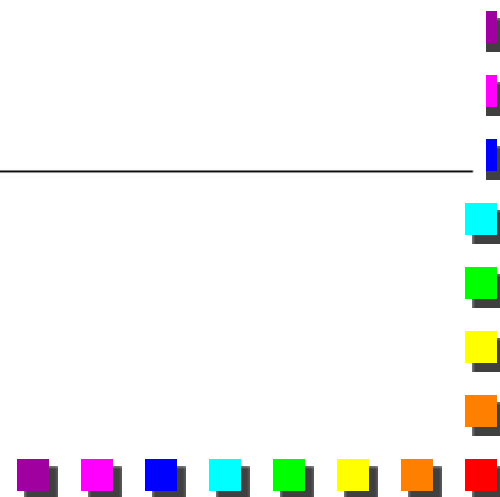
Buffer.c

- Buffer.c用于对高速缓冲区（池）进行操作和管理

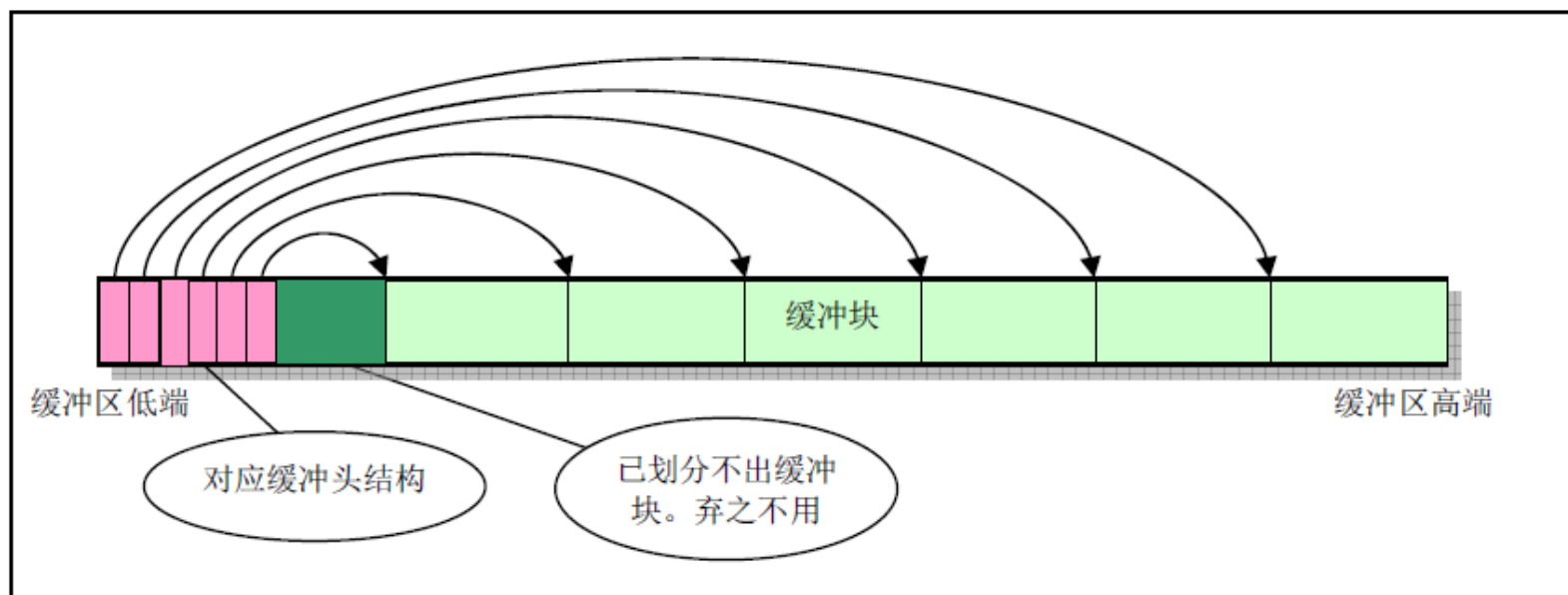


Buffer_head Data Structure

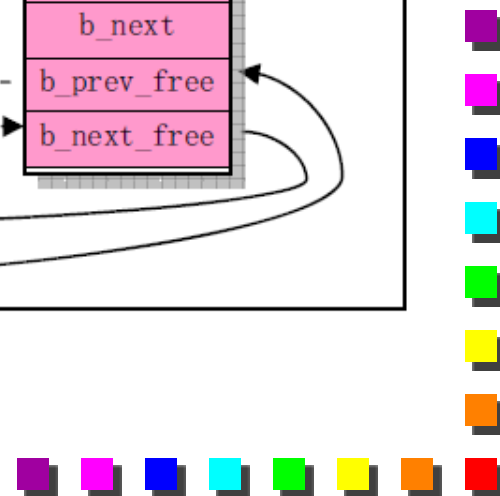
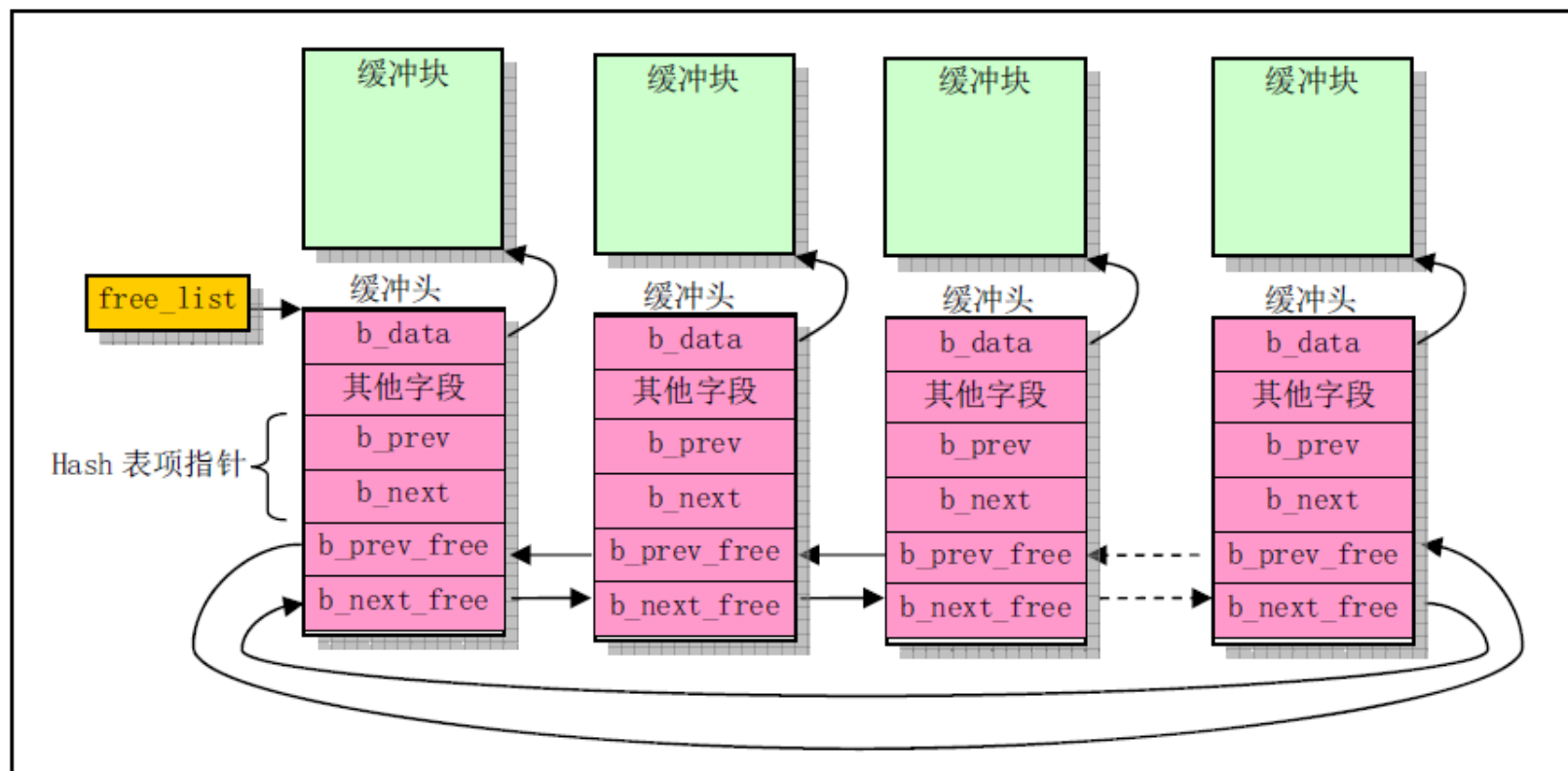
```
struct buffer\_head {  
  
    char * b_data;                // 指向该缓冲块中数据区 (1024 字节) 的指针。  
    unsigned long b_blocknr;      // 块号。  
    unsigned short b_dev;        // 数据源的设备号 (0 = free)。  
    unsigned char b_uptodate;    // 更新标志: 表示数据是否已更新。  
    unsigned char b_dirt;        // 修改标志: 0- 未修改(clean), 1- 已修改(dirty)。  
    unsigned char b_count;       // 使用该块的用户数。  
    unsigned char b_lock;        // 缓冲区是否被锁定。0- ok, 1- locked  
    struct task\_struct * b_wait;  // 指向等待该缓冲区解锁的任务。  
    struct buffer\_head * b_prev;  // hash 队列上前一块 (这四个指针用于缓冲区管理)。  
    struct buffer\_head * b_next;  // hash 队列上下一块。  
    struct buffer\_head * b_prev_free; // 空闲表上前一块。  
    struct buffer\_head * b_next_free; // 空闲表上下一块。  
  
};
```



高速缓冲区的初始化



缓冲块组成的双向链表结构



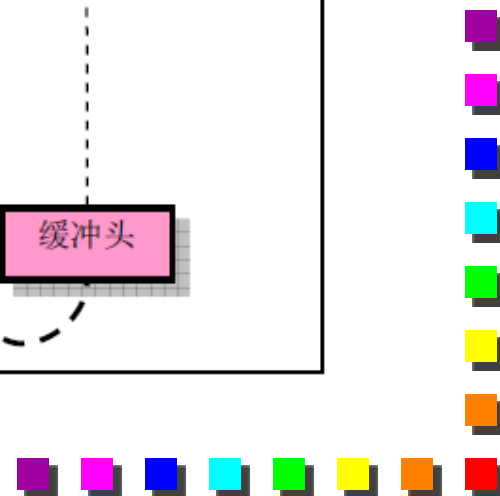
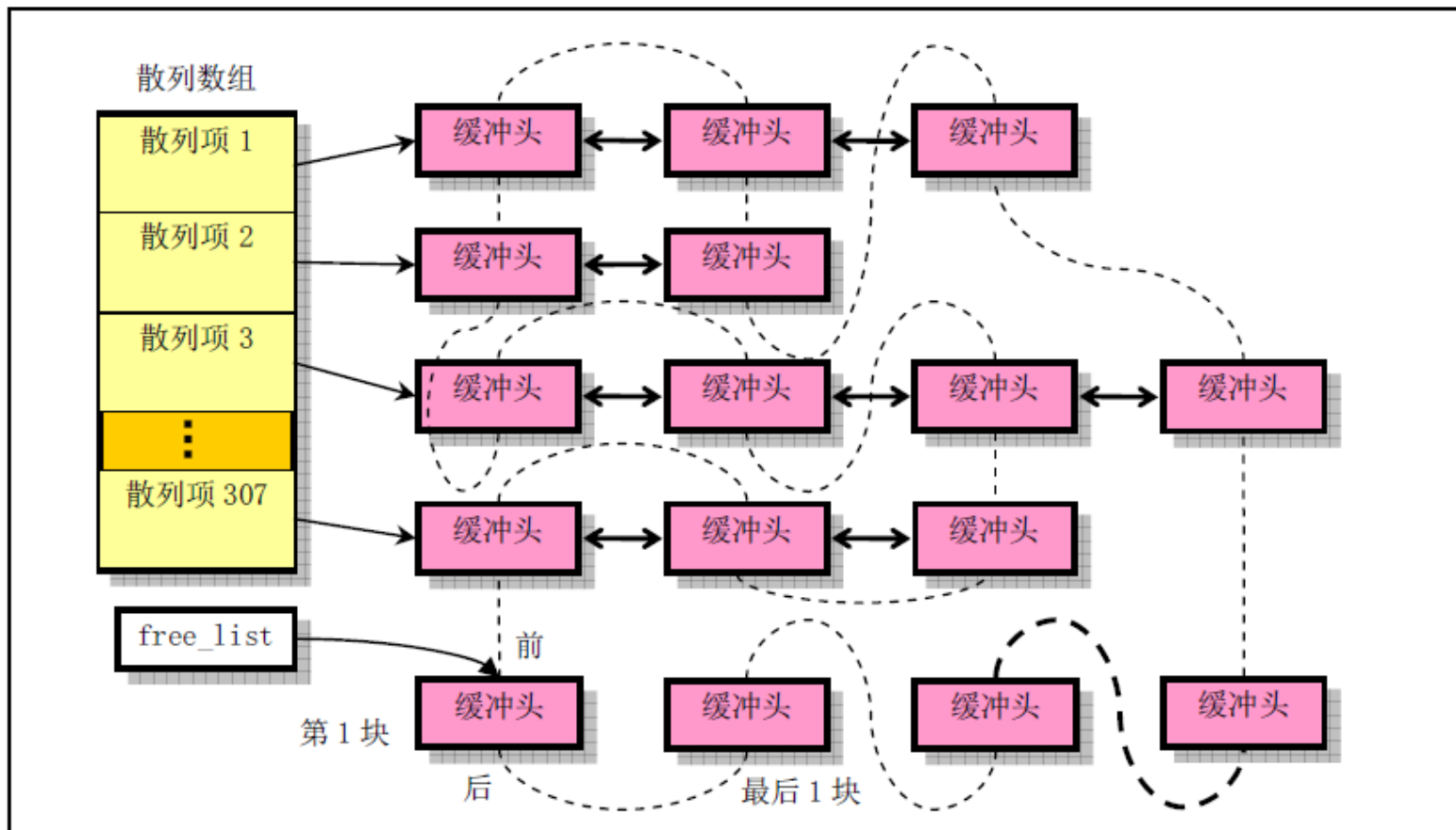
缓冲区管理函数及层次关系

- 缓冲块读取函数**bread()**、**bread_page()**、**breada()**
- 缓冲区搜索管理函数**getblk()**，寻找空闲缓冲块
- 释放缓冲块**brelse()**

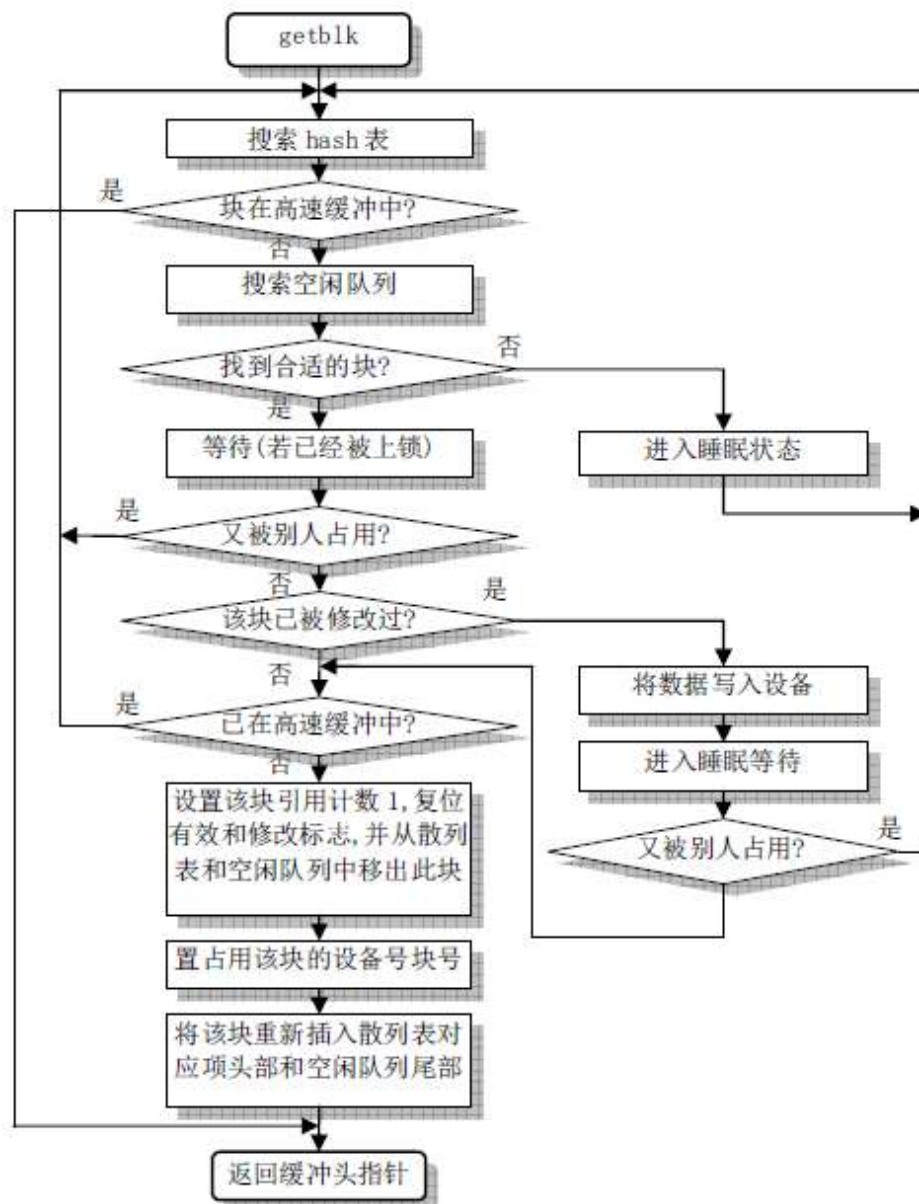
bread, breada, bread_page	
(getblk)	brelse
get_hash_table, find_buffer 等	



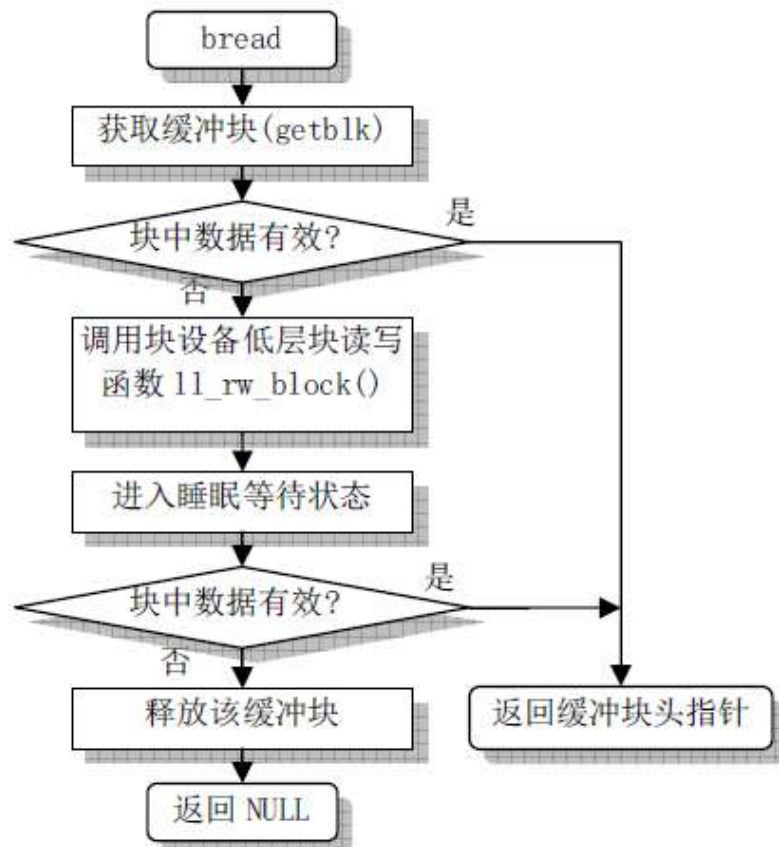
缓冲区Hash队列



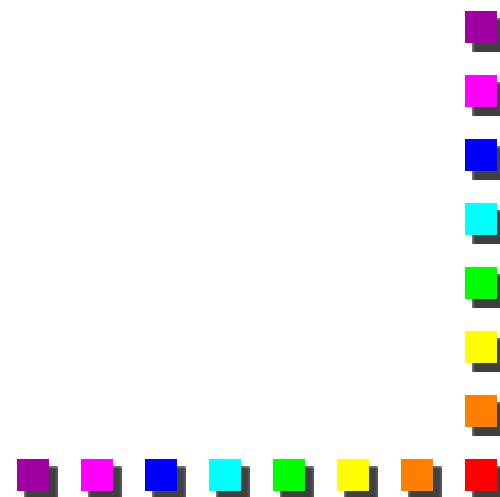
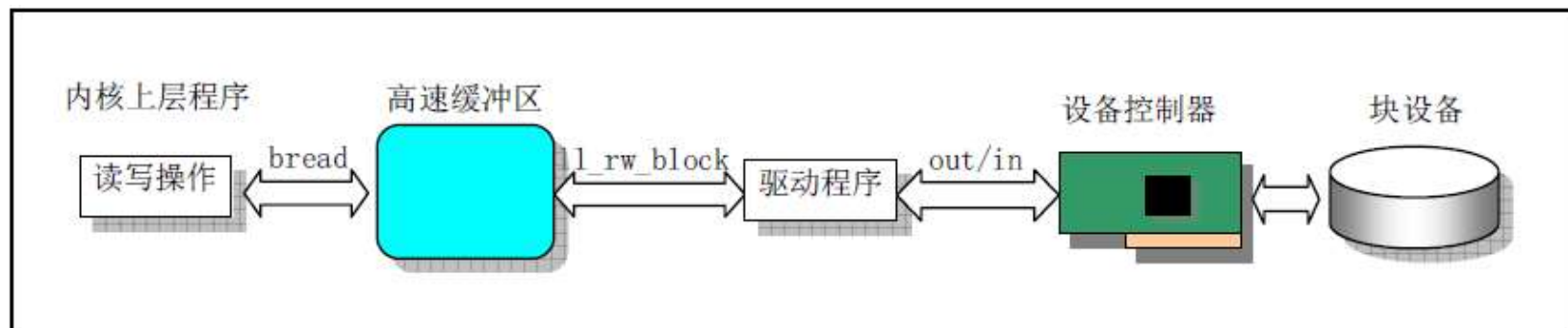
Getblk()



Bread()

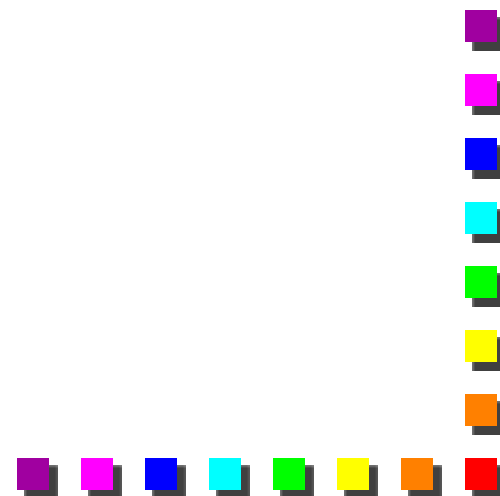


内核程序块设备访问操作



Ext2 File System

- 从路径名到目标节点
 - `_user_walk()` → `path_init()` 和 `path_walk()`



```
===== fs/namei.c 778 803 =====
778 /*
779  *  namei()
780  *
781  *  is used by most simple commands to get the inode of a specified name.
782  *  Open, link etc use their own routines, but this is enough for things
783  *  like 'chmod' etc.
784  *
785  *  namei exists in two versions: namei/lnamei. The only difference is
786  *  that namei follows links, while lnamei does not.
787  *  SMP-safe
788  */
789 int __user_walk(const char *name, unsigned flags, struct nameidata *nd)
790 {
791     char *tmp;
792     int err;
793
794     tmp = getname(name);
795     err = PTR_ERR(tmp);
796     if (!IS_ERR(tmp)) {
797         err = 0;
798         if (path_init(tmp, flags, nd))
799             err = path_walk(tmp, nd);
800         putname(tmp);
801     }
802     return err;
803 }
```



上海交通大學



Path init()

===== fs/namei.c 690 ~702 =====

```
690 /* SMP-safe */
691 int path_init(const char *name, unsigned int flags, struct nameidata *nd)
692 {
693     nd->last_type = LAST_ROOT; /* if there are only slashes... */
694     nd->flags = flags;
695     if (*name=='/')
696         return walk_init_root(name, nd);
697     read_lock(&current->fs->lock);
698     nd->mnt = mntget(current->fs->pwdmnt);
699     nd->dentry = dget(current->fs->pwd);
700     read_unlock(&current->fs->lock);
701     return 1;
702 }
```



上海交通大學

===== fs/namei.c 671 688 =====

[path_init()>walk_init_root()]

```
671  /* SMP-safe */
672  static inline int
673  walk_init_root(const char *name, struct nameidata *nd)
674  {
675      read_lock(&current->fs->lock);
676      if (current->fs->altroot && !(nd->flags & LOOKUP_NOALT)) {
677          nd->mnt = mntget(current->fs->altrootmnt);
678          nd->dentry = dget(current->fs->altroot);
679          read_unlock(&current->fs->lock);
680          if (__emul_lookup_dentry(name, nd))
681              return 0;
682          read_lock(&current->fs->lock);
683      }
684      nd->mnt = mntget(current->fs->rootmnt);
685      nd->dentry = dget(current->fs->root);
686      read_unlock(&current->fs->lock);
687      return 1;
688  }
```

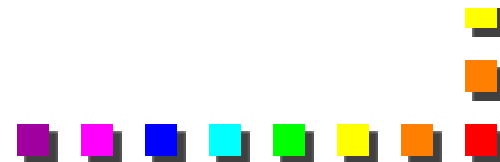


上海交通大學



Path_walk()

```
===== fs/namei.c 414 437 =====  
414 /*  
415  * Name resolution.  
416  *  
417  * This is the basic name resolution function, turning a pathname  
418  * into the final dentry.  
419  *  
420  * We expect 'base' to be positive and a directory.  
421  */  
422 int path_walk(const char * name, struct nameidata *nd)  
423 {  
424     struct dentry *dentry;  
425     struct inode *inode;  
426     int err;  
427     unsigned int lookup_flags = nd->flags;  
428  
429     while (*name=='/')  
430         name++;  
431     if (!*name)  
432         goto return_base;  
433  
434     inode = nd->dentry->d_inode;  
435     if (current->link_count)  
436         lookup_flags = LOOKUP_FOLLOW;  
437
```



===== fs/namei.c 438 467 =====

[path_walk()]

```
438      /* At this point we know we have a real path component. */
439      for(;;) {
440          unsigned long hash;
441          struct qstr this;
442          unsigned int c;
443
444          err = permission(inode, MAY_EXEC);
445          dentry = ERR_PTR(err);
446          if (err)
447              break;
448
449          this.name = name;
450          c = *(const unsigned char *)name;
451
452          hash = init_name_hash();
453          do {
454              name++;
455              hash = partial_name_hash(c, hash);
456              c = *(const unsigned char *)name;
457          } while (c && (c != '/'));
458          this.len = name - (const char *) this.name;
459          this.hash = end_name_hash(hash);
460
461          /* remove trailing slashes? */
462          if (!c)
463              goto last_component;
464          while (*++name == '/');
465          if (!*name)
466              goto last_with_slashes;
467
```



上海交通大學

Path_walk() contd.

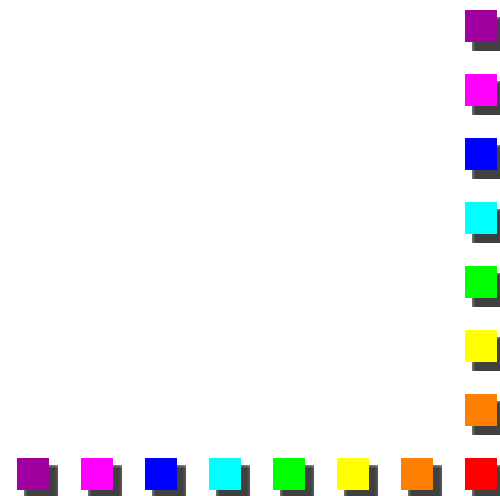
===== fs/namei.c 468 484 =====

[path_walk()]

```
468      /*
469       * "." and ".." are special - ".." especially so because it has
470       * to be able to know about the current root directory and
471       * parent relationships.
472       */
473      if (this.name[0] == '.') switch (this.len) {
474          default:
475              break;
476          case 2:
477              if (this.name[1] != '.')
478                  break;
479              follow_dotdot(nd);
480              inode = nd->dentry->d_inode;
481              /* fallthrough */
482          case 1:
483              continue;
484      }
```



上海交通大學



===== fs/namei.c 380 413 =====

[path_walk()>follow_dotdot()]

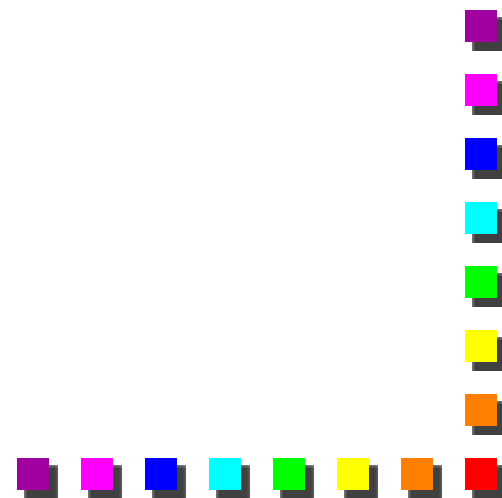
```
380 static inline void follow_dotdot(struct nameidata *nd)
381 {
382     while(1) {
383         struct vfsmount *parent;
384         struct dentry *dentry;
385         read_lock(&current->fs->lock);
386         if (nd->dentry == current->fs->root &&
387             nd->mnt == current->fs->rootmnt) {
388             read_unlock(&current->fs->lock);
389             break;
390         }
391         read_unlock(&current->fs->lock);
392         spin_lock(&dcache_lock);
393         if (nd->dentry != nd->mnt->mnt_root) {
394             dentry = dget(nd->dentry->d_parent);
395             spin_unlock(&dcache_lock);
396             dput(nd->dentry);
397             nd->dentry = dentry;
398             break;
399         }
400         parent=nd->mnt->mnt_parent;
401         if (parent == nd->mnt) {
402             spin_unlock(&dcache_lock);
403             break;
404         }
405         mntget(parent);
406         dentry=dget(nd->mnt->mnt_mountpoint);
407         spin_unlock(&dcache_lock);
408         dput(nd->dentry);
409         nd->dentry = dentry;
410         mntput(nd->mnt);
411         nd->mnt = parent;
412     }
413 }
```



上海交通大學

三种情况

- 已到达节点 $nd \rightarrow dentry$ 就是本进程的根节点：保持 $nd \rightarrow dentry$ 不变
- 已到达节点 $nd \rightarrow dentry$ 与其父节点不在同一个设备上：往上跑一层至 $d \rightarrow parent$
- 已到达节点 $nd \rightarrow dentry$ 就是其所在设备的根节点：通过`vfsmount`检查



Path_walk() contd.

===== fs/namei.c 485 535 =====

[path_walk()]

```
485      /*
486       * See if the low-level filesystem might want
487       * to use its own hash..
488       */
489      if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
490          err = nd->dentry->d_op->d_hash(nd->dentry, &this);
491          if (err < 0)
492              break;
493      }
494      /* This does the actual lookups.. */
495      dentry = cached_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
496      if (!dentry) {
497          dentry = real_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
498          err = PTR_ERR(dentry);
```



```

499         if (IS_ERR(dentry))
500             break;
501     }
502     /* Check mountpoints.. */
503     while (d_mountpoint(dentry) && __follow_down(&nd->mnt, &dentry))
504         ;
505
506     err = -ENOENT;
507     inode = dentry->d_inode;
508     if (!inode)
509         goto out_dput;
510     err = -ENOTDIR;
511     if (!inode->i_op)
512         goto out_dput;
513
514     if (inode->i_op->follow_link) {
515         err = do_follow_link(dentry, nd);
516         dput(dentry);
517         if (err)
518             goto return_err;
519         err = -ENOENT;
520         inode = nd->dentry->d_inode;
521         if (!inode)
522             break;
523         err = -ENOTDIR;
524         if (!inode->i_op)
525             break;
526     } else {
527         dput(nd->dentry);
528         nd->dentry = dentry;
529     }
530     err = -ENOTDIR;
531     if (!inode->i_op->lookup)
532         break;
533     continue;
534     /* here ends the main loop */
535

```



上海交通大学

Cached_lookup()

===== fs/namei.c 243 258 =====

[path_walk()>cached_lookup()]

```
243  /*
244   * Internal lookup() using the new generic dcache.
245   * SMP-safe
246   */
247  static struct dentry * cached_lookup(struct dentry * parent, struct qstr * name, int flags)
248  {
249      struct dentry * dentry = d_lookup(parent, name);
250
251      if (dentry && dentry->d_op && dentry->d_op->d_revalidate) {
252          if (!dentry->d_op->d_revalidate(dentry, flags) && !d_invalidate(dentry)) {
253              dput(dentry);
254              dentry = NULL;
255          }
256      }
257      return dentry;
258  }
```



上海交通大學



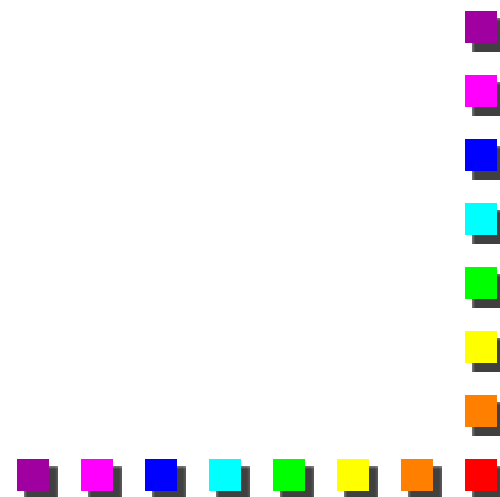
D_lookup()

```
===== fs/dcache.c 703 749 =====  
[path_walk()>cached_lookup()>d_lookup()]  
703 /**  
704  * d_lookup - search for a dentry  
705  * @parent: parent dentry  
706  * @name: qstr of name we wish to find  
707  *  
708  * Searches the children of the parent dentry for the name in question. If  
709  * the dentry is found its reference count is incremented and the dentry  
710  * is returned. The caller must use d_put to free the entry when it has  
711  * finished using it. %NULL is returned on failure.  
712  */  
713  
714 struct dentry * d_lookup(struct dentry * parent, struct qstr * name)  
715 {  
716     unsigned int len = name->len;  
717     unsigned int hash = name->hash;  
718     const unsigned char *str = name->name;  
719     struct list_head *head = d_hash(parent,hash);  
720     struct list_head *tmp;  
721  
722     spin_lock(&dcache_lock);  
723     tmp = head->next;
```

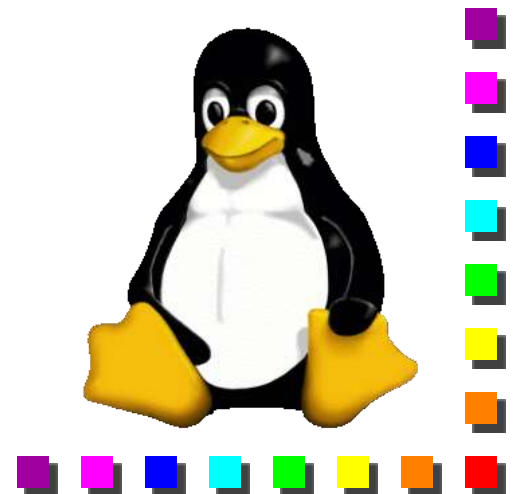
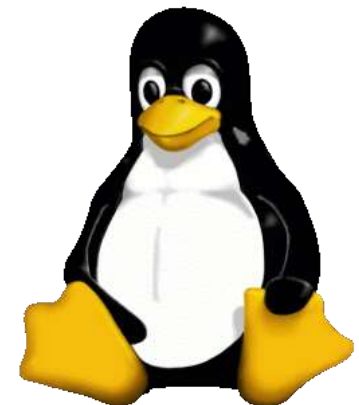


D_lookup() contd.

```
724     for (;;) {
725         struct dentry * dentry = list_entry(tmp, struct dentry, d_hash);
726         if (tmp == head)
727             break;
728         tmp = tmp->next;
729         if (dentry->d_name.hash != hash)
730             continue;
731         if (dentry->d_parent != parent)
732             continue;
733         if (parent->d_op && parent->d_op->d_compare) {
734             if (parent->d_op->d_compare(parent, &dentry->d_name, name))
735                 continue;
736         } else {
737             if (dentry->d_name.len != len)
738                 continue;
739             if (memcmp(dentry->d_name.name, str, len))
740                 continue;
741         }
742         __dget_locked(dentry);
743         dentry->d_flags |= DCACHE_REFERENCED;
744         spin_unlock(&dcache_lock);
745         return dentry;
746     }
747     spin_unlock(&dcache_lock);
748     return NULL;
749 }
```



Project 4: File System



上海交通大學

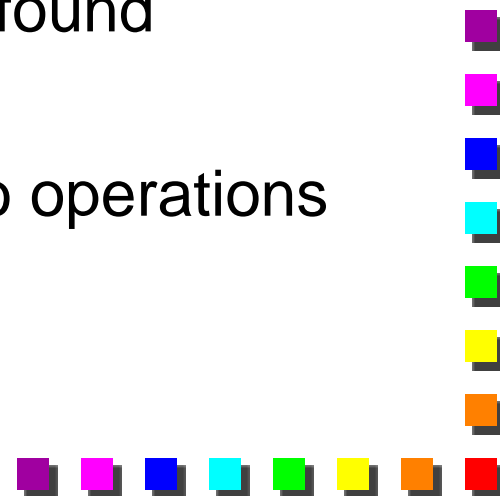
Source

- Inode.c/Makefile (kernel source of romfs)
- Test.img (a romfs image, you can mount it to a dir with 'mount -o loop test.img xxx')
- Say test.img is mounted in t, 'find t' output
 - aa
 - bb
 - ft
 - fo
 - fo/aa



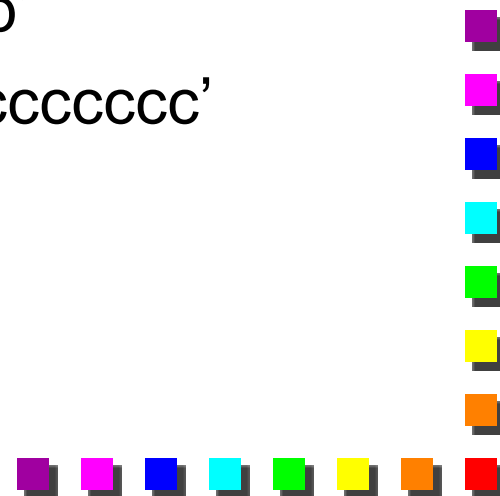
Practice 1

- Change romfs code to hide a file/dir with special name
- Test & result
 - insmod romfs hided_file_name="aa"
 - Mount -o loop test.img t
 - then ls t, ls t/fo, no "aa" and "fo/aa". found
 - ls t/aa, or ls fo/aa, no found
 - Without the code change, above two operations can find file 'aa'



Practice 2

- change the code of romfs to correctly read info of an 'encrypted' romfs
- Test & result
 - insmod romfs hided_file_name="bb"
 - Mount -o loop test.img t
 - Say bb's original content is 'bbbbbbbb'
 - With the change, cat t/bb output 'ccccccccc'



Practice 3

- change the code of romfs to add 'x' (execution) bit for a specific file
- Test & result
 - insmod romfs hided_file_name="bb"
 - Mount -o loop test.img t
 - Without code changes 'ls -l t', output is '-rw-r--r--'
 - With the change, output is '-rwxr-xr-x'

