

Operating System

Dr. GuoJun LIU

Harbin Institute of Technology

<http://os.guojunhit.cn>

文件系统

Slides-4

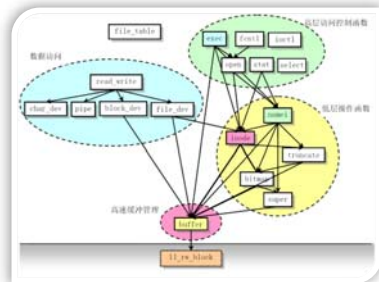
Chapter A3

File System

Linux 0.11

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

Dr. GuoJun LIU

Operating System

Slides-5

Outline

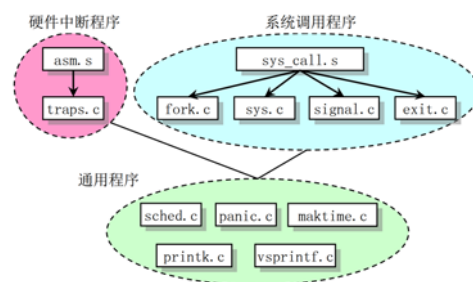
- 文件系统
- 进程打开文件使用的内核数据结构
- 缓冲
- 用户程序读写操作过程

Dr. GuoJun LIU

Operating System

Slides-3

各文件的调用层次关系



Dr. GuoJun LIU

Operating System

Slides-6

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

Dr. GuoJun LIU

Operating System

Slides-7

Page fault

The diagram illustrates the page fault handling process. It shows a memory layout with a 'free_list' and 'Hash 表项指针' (Hash table pointers). A 'Page fault' occurs when a process requests a page not in memory. The system searches the hash table for the page. If found, it's a hit; if not, it's a fault. The diagram shows the process of finding a free page, swapping it with the page on disk, and updating the hash table and memory buffers.

Dr. GuoJun LIU

Operating System

Slides-10

进程打开文件使用的内核数据结构

Slides-8

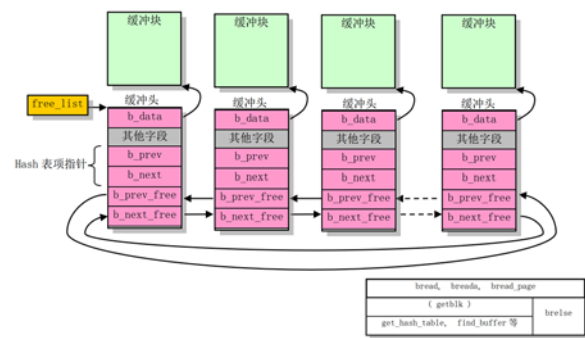
缓冲

Slides-11

The diagram illustrates the kernel data structures for file opening. It shows the 'struct task_struct' containing 'file_table' and 'file' pointers. The 'file' pointer points to a 'struct file' which contains 'f_inode' and 'f_op'. The 'f_inode' points to a 'struct inode' which contains 'i_data' and 'i_op'. The 'i_data' points to a 'struct buffer_head' which contains 'b_data' and 'b_next'. The 'b_next' points to another 'struct buffer_head'.

Slides-9

缓冲块组成的双向循环空闲链表结构



Dr. GuoJun LIU

Operating System

Slides-12

Slides-13

The diagram illustrates the file system architecture, showing the flow of data and control between various components:

- 内核上层程序 (Upper Kernel Program):** Contains a **读写操作 (Read/Write Operation)** block.
- 高速缓冲区 (High-Speed Buffer):** A large blue box representing the buffer.
- 驱动程序 (Driver):** A box that receives data from the buffer and sends it to the device controller.
- 设备控制器 (Device Controller):** A green box that interfaces with the **块设备 (Block Device)**, represented by a disk icon.
- 块设备 (Block Device):** The physical storage device.

Arrows indicate the flow of data: **读写操作** ↔ **高速缓冲区** ↔ **驱动程序** ↔ **设备控制器** ↔ **块设备**.

Below this main flow, a detailed view of the **File management** process is shown:

- User & program commands** and **User access control** feed into the **File management** process.
- The **File management** process involves **Directory management**, **Operation, file name**, and **File manipulation functions**.
- These functions interact with **Records** (represented by horizontal bars).
- The process then moves to **Access method**, which involves **Blocking** and **File allocation**.
- Physical blocks in main memory buffers are managed through **Disk scheduling** and **Disk scheduling**.
- The final step is **Free storage management**, which involves **File allocation** and **Disk scheduling**.

A double-headed arrow at the bottom indicates the **File management context**.

Slides-16

```

1 struct buffer_head *getblk(inode_t *in, blk_t block)
2 {
3     struct buffer_head *new, *bh;
4
5     repeat:
6         if (!in->get_block_table(inode_block))
7             return
8         new = free_list;
9         do {
10             if (!new->b_count)
11                 break;
12             if (! (in->B_BMAPS == new->B_BMAPS)) {
13                 /*
14                  * We want until we find something good */
15                 while (in->B_map_start_free != free_list)
16                     continue;
17                 new = free_list;
18             }
19             get_block(new);
20             /*
21              * We want an unlinked block, commonly also write */
22             if (!in->B_count)
23                 break;
24             while (in->B_count > 0)
25                 continue;
26             new = free_list;
27             if (!new->b_count)
28                 break;
29             get_block(new);
30             /*
31              * We want that this buffer is the only one of its kind.
32              * and that it's unused (b_count=0), unlinked (b_linked=0), and clean */
33             if (!in->B_count || in->B_linked || in->B_dirty || in->B_clean)
34                 continue;
35             in->B_count--;
36             in->B_clean=0;
37             in->B_dirty=0;
38             in->B_map_start_free = free_list;
39             return new;
40         } while (1);
41 }

```

Slides-14

[illegible]

Slides-17

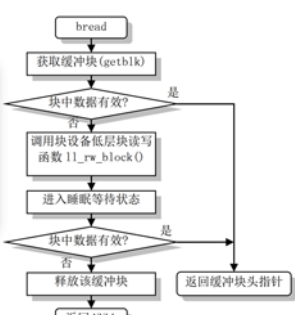


```

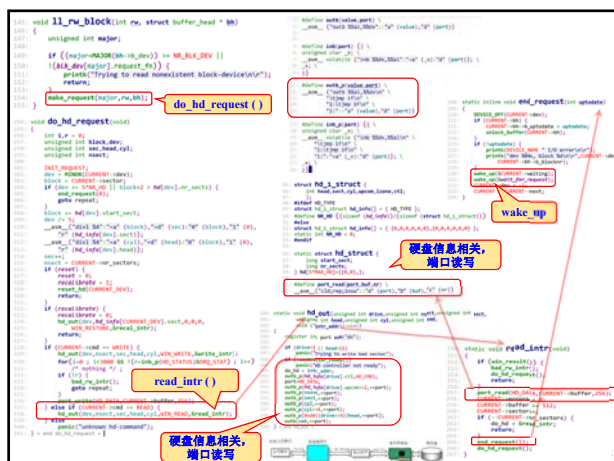
270 struct buffer_head * bread(int dev,int block)
271 {
272     struct buffer_head * bh;
273
274     if (!bh=getblk(dev,block))
275         panic("bread: getblk returned NULL\n");
276     if (!bh->uupdate)
277         return bh;
278     ll_rw_block(READ,bh);
279     wait_on_buffer(bh);
280     if (bh->uupdate)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

```

Slides 15



Slides 18



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