第13章 操作系统实例分析

孙承杰

E-mail：sunchengjie@hit.edu.cn

哈工大计算学部人工智能教研室

2023年秋季学期

第13章 操作系统实例分析

进程-内存-文件系统-IO系统

基于一个简单linux0.11
分析操作系统核心模块间交互和结构设计

纸上得来终觉浅，绝知此事要躬行

思考几个小问题

- 使用fork创建一个进程的过程？
- 如何将一个可执行文件变成进程？
- 基于虚拟内存部分加载、按需调页？
- 进程如何操作文件？
- 磁盘操作很慢，IO子系统如何解决？

大纲

- 进程控制块结构分析
- 父进程创建子进程的过程（**fork**）
- 进程与文件的连接
- 进程操作文件时内存中数据结构
- 文件的操作的核心过程
- **a.out**可执行文件头
- 创建一个与父进程不同的子进程
- 虚拟内存的缺页调页过程
- 基于缓冲区的磁盘访问（**IO**缓冲区和调度）

PCB中的关键元素

- 操作系统通过PCB感知和管理进程
- 进程运行在内存中
- 进程与文件系统的关系如何建立？
  - 进程的可执行映像是文件；
  - 进程对辅存读写也主要基于文件
- 内存与辅存间交换信息需要建立内存缓冲区（IO内核子系统）

几大核心模块能够有机地建立联系

重新分析进程控制块PCB

```
struct task_struct {
/*-----------------------------------------------*/
    long state;    // 进程运行状态（-1 不可运行，0 可运行，>0 以停止）
    long counter; // 任务运行时间片，递减到 0 是说明时间片用完
    long priority;     // 任务运行优先数，刚开始是 counter＝priority
    long signal;  // 任务的信号位图，信号值=偏移+1
    struct sigaction sigaction[32];  //信号执行属性结构，对应信号将要执行的操作和标志信息
    long blocked; // 信号屏蔽码
/*--------------- various fields---------------------- */
    int exit_code;     // 任务退出码，当任务结束时其父进程会读取
    unsigned long start_code, end_code, end_data, brk, start_stack;
        // start_code      代码段起始的线性地址
        // end_code    代码段长度
        // end_data    代码段长度+数据段长度
        // brk         代码段长度+数据段长度+bss 段长度
        // start_stack     堆栈段起始线性地址
    long pid, father, pgrp, session, leader;
        // pid 进程号，father 父进程号，pgrp 父进程组号，session 会话号，leader 会话首领
    unsigned short uid, euid, suid;
        // uid 用户标 id，euid 有效用户 id，suid 保存的用户 id
    unsigned short gid, egid, sgid;
        // gid 组 id，egid 有效组 id，sgid 保存组 id
    long alarm;    // 报警定时值
```

创建进程时设置

重新分析进程控制块PCB

```
    long alarm;    // 报警定时值
    long utime,stime,cutime,cstime,start_time;
        // utime 用户态运行时间
        // stime 内核态运行时间
        // cutime 子进程用户态运行时间
        // cstime 子进程内核态运行时间
        // start_time 进程开始运行时刻
    unsigned short used_math;    // 标志，是否使用了 387 协处理器
/* ------------------------file system info--------------------- */
    int tty;        // 进程使用 tty 的子设备号，-1 表示没有使用
    unsigned short umask;   //文件创建属性屏蔽码
    struct m_inode * pwd;
    struct m_inode * root;
    struct m_inode * execut
    struct file * filp[NR_O
    unsigned long close_on_

/*-------- ldt for this task
    struct desc_struct ldt[
/* ----------------------tss for this task ---------------------*/
    struct tss_struct tss;       // 本任务的 tss 段
};
```

- **TSS 全称为task state segment。**
- **X86体系从硬件上支持任务间的切换。**
- **任务状态段(TSS)，它和数据段、代码段一样也是一种段。**
- **任务切换的时候，CPU会将原寄存器的内容写出到相应的TSS，同时将新TSS的内容填到寄存器中，这样就实现了任务的切换。**

重新分析进程控制块PCB

进程与文件连接的关键结构？

```
struct m_inode * pwd;
struct m_inode * root;
struct m_inode * executable; //进程可执行文件
struct file * filp[NR_OPEN]; //进程操作的其他文件
```

m_inode、file这两个数据结构是关键！

fork创建一个进程的过程

- 使用fork创建一个子进程核心过程包括：创建进程控制块、分配内存、IO系统、文件访问。
- 回顾用fork创建子进程的情形

- **这个过程没有通过文件系统访问块设备！**
  fork创造子进程与父进程完全相同

MINIX文件系统中针对inode的结构

	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
	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 对于设备特殊文件名的 i 节点，其 zon 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。

Linux\fs.h

```
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 };
```

MINIX文件系统1.0的m_node数据结构

Linux0.11进程与文件的连接

Linux/fs.h

```
struct file {
  unsigned short f_mode;          //文件操作模式
  unsigned short f_flags;         //文件打开控制标志
  unsigned short f_count;         //当前文件打开的次数
  struct m_inode * f_inode;       //指向文件对应的i节点(inode_table中)
  off_t f_pos;                    //文件位置（读写偏移位置）
};
#define NR_OPEN 20   //一个进程可以打开文件的最大数（包括相同的）
#define NR_FILE 64    //操作系统可以打开文件总数（包括相同的）
#define NR_INODE 32   //操作系统中可以打开不同文件总数
```

Linux/sched.h

```
struct file * filp[NR_OPEN];  //管理进程使用文件的指针数组
```

Fs/file_table.c

```
struct file file_table[NR_FILE];//file数组，记录操作系统打开文件信息
```

fs/inode.c

```
struct m_inode inode_table[NR_INODE]={{0,},};
```

思考一下这个顺序的含义？

进程打开文件时数据结构的建立

- 打开文件
- 1）将用户进程打开的文件（句柄fd）在filp[20]中进行登记
- 2）建立filp[fd]与内核中的file_table[64]进行连接
- 3）将要打开文件的i节点在file_table[64]中进行登记
- 4）将新打开的i节点登记到inode_table[32]

- 通过inode_table[32]掌控正在使用的文件i节点数
- 打开文件的本质就是要建立*filp[20]、file_table[64]、inode_table[32]三者之间的关系

进程打开文件时数据结构的建立

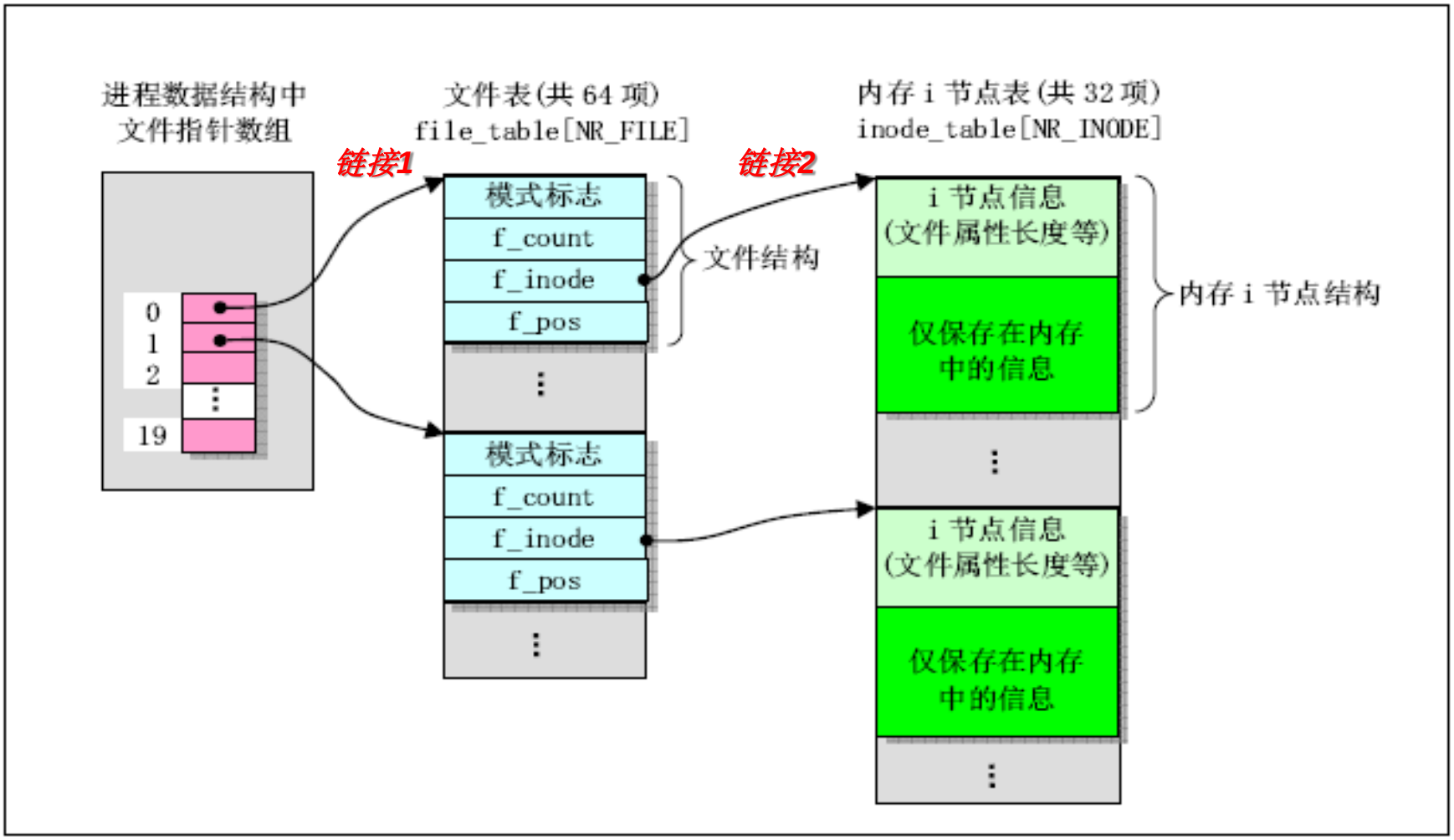
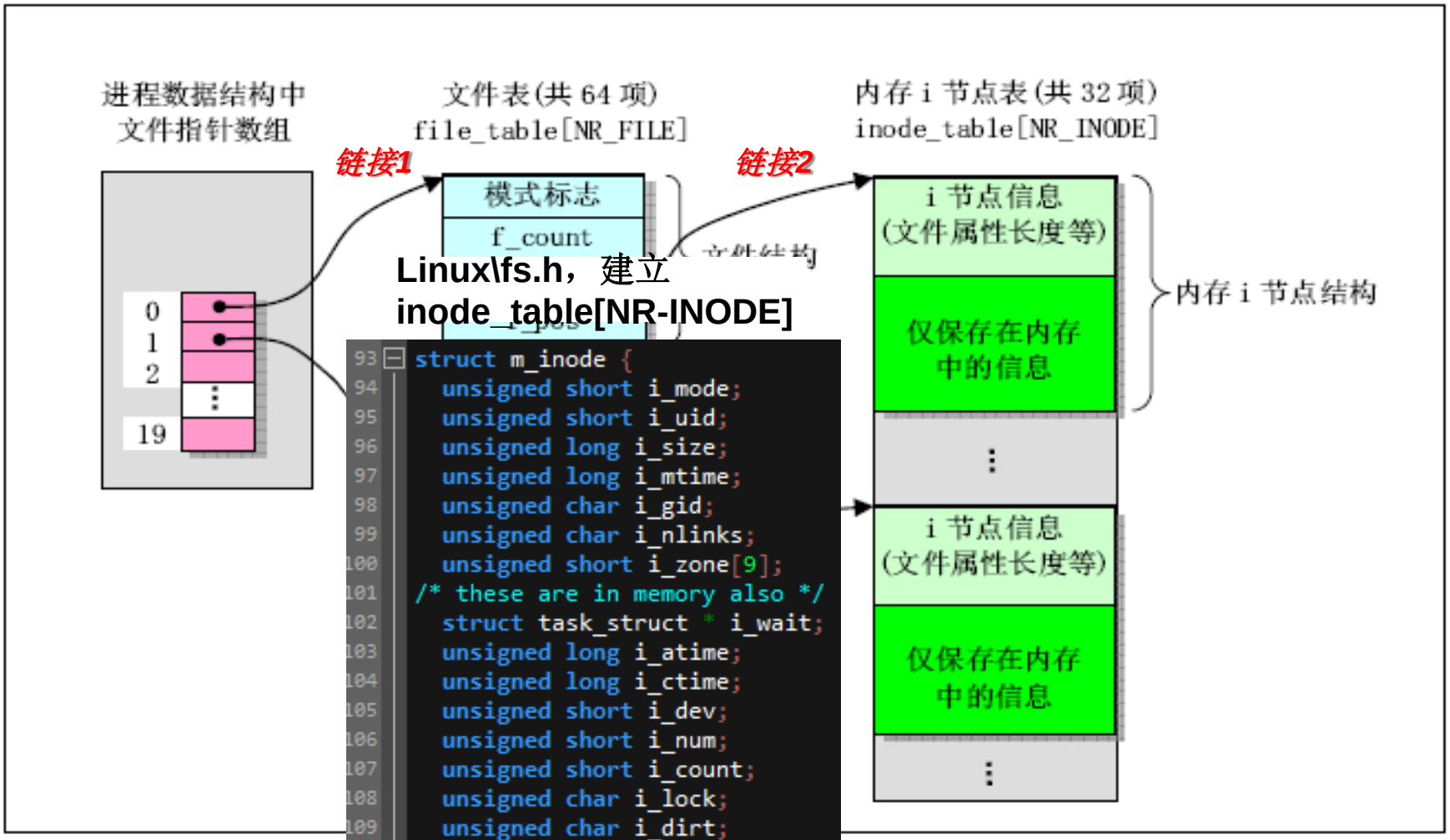


图 12-13 进程打开文件使用的内核数据结构

filp[20]

filp[20]

进程打开文件时数据结构的建立



进程打开文件时数据结构的建立

```
int sys_open(const char * filename,int flag,int mode)
{
        struct m_inode * inode; // inode ？
        struct file * f;  //f是什么？
        int i,fd;  //fd是什么？
……
        for(fd=0 ; fd<NR_OPEN ; fd++)   //fd从当前进程flip[20]中寻找空闲项
                if (!current->filp[fd])
                        break;
……
        f=0+file_table;   //f为文件指针，其指向了filetable文件数组起始地址
        for (i=0 ; i<NR_FILE ; i++,f++)     //从file_table[64]中寻找空闲项
                if (!f->f_count) break;    // f_count等于0时找到
……
        //将进程filp与file_table对应项挂接，增加文件句柄计数（链接1）
        (current->filp[fd]=f)->f_count++;
… …
}
```

进程打开文件时数据结构

文件存储的绝对路径
/mnt/test/my.c

```
int sys_open(const char * filename,int flag,int mod
{
……
        //循环路径解析，找到i节点，存储到inode_table中
        if ((i=open_namei(filename,flag,mode,&inode))<0) {
                current->filp[fd]=NULL;
                f->f_count=0;
                return i;
        }
……
        f->f_mode = inode->i_mode;  //用i节点
        f->f_flags = flag;   //设置文件操作方式
        f->f_count = 1;    //将文件引用计数加1
        f->f_inode = inode;  //将文件与inode_table 中i 节点建立关系（链接2）
        f->f_pos = 0;      //将文件读写指针设为0
        return (fd);      //把文件句柄返回用户空间
}
```

传地址，找到后inode就变成inode_table中的项

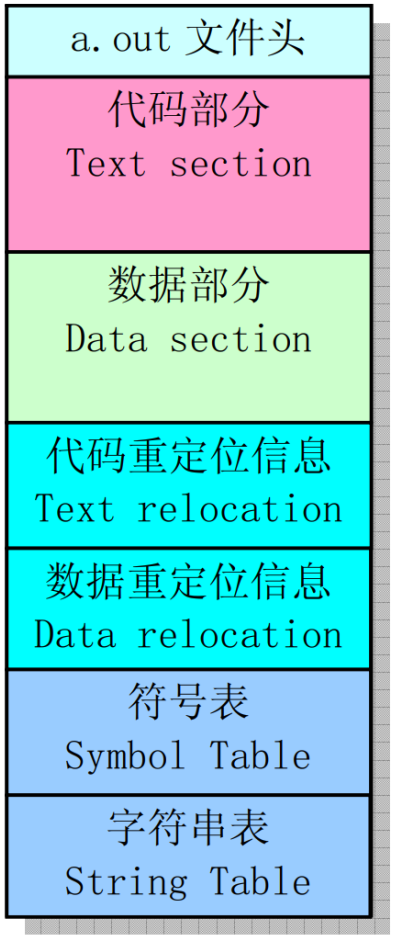
进程打开文件时数据结构的建立

```
    long alarm;    // 报警定时值
    long utime,stime,cutime,cstime,start_time;
        // utime 用户态运行时间
        // stime 内核态运行时间
        // cutime 子进程用户态运行时间
        // cstime 子进程内核态运行时间
        // start_time 进程开始运行时刻
    unsigned short used_math;    // 标志，是否使用了 387 协处
/* ------------------------file system info-------------
    int tty;        // 进程使用 tty 的子设备号，-1 表示没有使
    unsigned short umask;   //文件创建属性屏蔽码
    struct m_inode * pwd;   // 当前工作目录的 i 节点
    struct m_inode * root;  // 根目录的 i 节点
    struct m_inode * executable;     // 可执行文件的 i 节点
    struct file * filp[NR_OPEN];     // 进程使用的文件
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志

/*-------- ldt for this task 0 - zero 1 - cs 2 - ds&ss ----------*/
    struct desc_struct ldt[3];//本任务的 ldt 表，0-空，1-代码段，2-数据和堆栈段
/* ----------------------tss for this task ----------------------*/
    struct tss_struct tss;       // 本任务的 tss 段
};
```

均要查找和使用i节点。建立过程？

a.out可执行文件及文件头分析

- **UNIX/LINUX平台下三种主要的可执行文件格式：**
  - **a.out**（assembler and link editor output汇编器和链接编辑器的输出）：**linux0.11支持的**
  - COFF（Common Object File Format 通用对象文件格式）
  - **ELF（Executable and Linking Format 可执行和链接格式）**



a.out 格式的目标文件

Linux0.11中a.out可执行文件及文件头分析

- a.out 目标文件包含 7 个 section，格式如下：
  - **exec header（文件头部）**
  - text segment（文本段）
    data segment(数据段)
  - text relocations(文本重定位段（表））
    data relocations（数据重定位段（表））
    symbol table（符号表）
  - string table（字符串表）

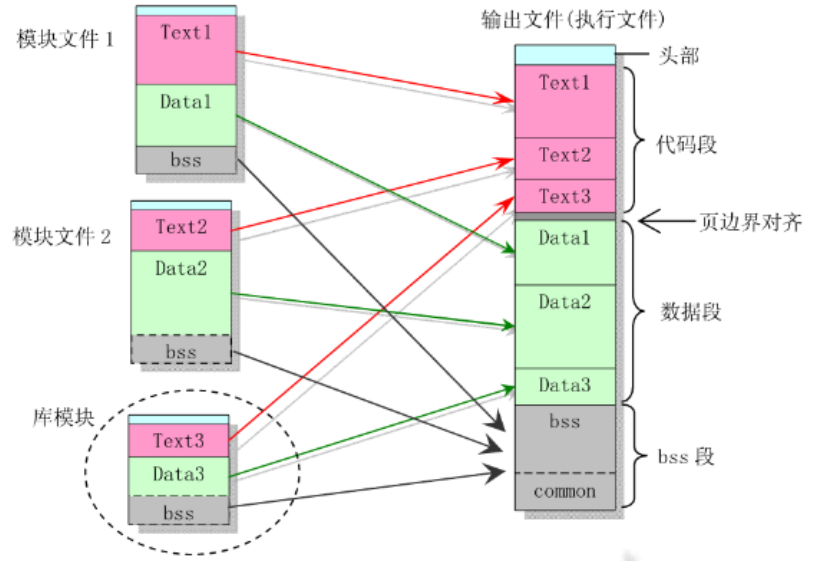


图 3 目标文件的链接操作

a.out可执行文件及文件头分析

- a.out 目标文件中包含符号表和两个重定位表，这三部分的内容在链接目标文件以生成可执行文件时起作用
- 最终可执行的a.out文件中，这三个部分的长度都为0。a.out文件在链接时就把所有外部定义静态链接在可执行文件中。不支持动态链接！

a.out可执行文件及文件头分析

- 可执行文件头的数据结构：

```
struct exec {
        unsigned long   a_magic;    /* 魔数和其它信息 */
        unsigned long   a_text;     /* 文本段的长度 */
        unsigned long   a_data;     /* 数据段的长度 */
        unsigned long   a_bss;      /* BSS段的长度 */
        unsigned long   a_syms;     /* 符号表的长度 */
        unsigned long   a_entry;    /* 程序进入点 */
        unsigned long   a_trsize;   /* 文本重定位表的长度 */
        unsigned long   a_drsize;   /* 数据重定位表的长度 */
};
```

a.out可执行文件及文件头分析

- 文件头部主要描述了各个 section 的长度，a_entry（程序进入点），代表了系统在加载程序并初试化各种环境后开始执行程序代码的入口
- 由 a.out 格式和头部数据结构我们可以看出，a.out的格式非常紧凑，只包含了程序运行所必须的信息（文本、数据、BSS），而且每个 section的顺序是固定的
- 这种结构缺乏扩展性，如不能包含"现代"可执行文件中常见的调试信息

a.out可执行文件及文件头分析

- linux0.11中，a.out的文件头单独存放在一个block（盘块）中
- 如何创建一个进程并执行一个新程序？
- 首先需要加载可执行文件的inode中的i_zone[0]指向的盘块中的内容（文件头）

在盘上和内存中的字段，共 32 字节

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。

创建一个与父进程不同的子进程

- Fork创建一个子进程，返回pid(=0子进程中）
- 在子进程中执行shell终端程序(/bin/sh)
- sh也只是一个正常的具有main函数的可执行程序，其读取用户在命令行输入的程序名以及相应参数，然后fork出一个子进程去execve这个程序，子程序执行完成后又回到命令行等待输入，
- execve(“/bin/sh”,argv,envp)函数是关键
- execve→ sys_execve → do_execve (exec.c)

函数的参数是通过堆栈传递的，返回地址是在call语句时被压入堆栈的，返回值通过eax传递。

```
01.  static char * argv[] = { "-/bin/sh",NULL };
02.  static char * envp[] = { "HOME=/usr/root", NULL };
03.
04.  void init(void)
05.  {
06.  /*... 省略前面代码...*/
07.
08.      while (1) {
09.          if ((pid=fork())<0) {
10.              printf("Fork failed in init\r\n");
11.              continue;
12.          }
13.          if (!pid) {
14.              close(0);close(1);close(2);
15.              setsid();
16.              (void) open("/dev/tty0",O_RDWR,0);
17.              (void) dup(0);
18.              (void) dup(0);
19.              _exit(execve("/bin/sh",argv,envp));
20.          }
21.          while (1)
22.              if (pid == wait(&i))
23.                  break;
24.          printf("\n\rchild %d died with code %04x\n\r",pid,i);
25.          sync();
26.      }
27.
28.      _exit(0);   /* NOTE! _exit, not exit() */
29.  }
```

do_execve过程分析

- execve(“/bin/sh”,argv,envp)
- linux/lib/execve.c

  _syscall3(int,execve,const char *,file,char **,argv,char **,envp)
- include/unistd.h

```
#define _syscall3(type,name,atype,a,btype,b,ctype,c)    \
type name(atype a,btype b,ctype c)   \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
        return (type) __res; \
errno=-__res; \
return -1; \
}
```

_NR_execve

“b”-%ebx, 保存filename

“c”-%ecx,“d”-%edx

系统调用号存入%eax

do_execve过程分析

```
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
        return (type) __res; \
errno=-__res; \
return -1; \
}
```

然后产生中断，硬件会做什么？？

硬件自动保存**ss,esp,eflags,cs,eip**到内核栈中。

do_execve分析

- **int $0x80 导致进入系统调用过程**
- **入栈顺序（内核栈）：ds,es,fs,edx,ecx,ebx**
- **入栈顺序（内核栈）：ds,es,fs,edx,ecx,ebx,call返回地址**

```
system_call:
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx          # push %ebx,%ecx,%edx as parameters
        pushl %ebx          # to the system call
        movl $0x10,%edx             # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
        movl $0x17,%edx             # fs points to local data space
        mov %dx,%fs
        call sys_call_table(,%eax,4)
```

```
sys_execve:
        lea EIP(%esp),%eax
        pushl %eax
        call do_execve
        addl $4,%esp
        ret
```

call (_sys_call_table+%eax*4)
同时：pushl call下条指令地址

do_execve过程分析--参数传递

- 入栈顺序（内核栈）：
- **ds,es,fs,edx,ecx,ebx, call返回地址,eax(eip地址）, call返回地址**

```
sys_execve:
        lea EIP(%esp),%eax
        pushl %eax
        call do_execve
        addl $4,%esp
        ret
```

栈中%eip所在地址

```
EAX             = 0x00
EBX             = 0x04
ECX             = 0x08
EDX             = 0x0C
FS              = 0x10
ES              = 0x14
DS              = 0x18
EIP             = 0x1C
CS              = 0x20
EFLAGS          = 0x24
OLDESP          = 0x28
OLDSS           = 0x2C
```

do_execve过程分析--参数传递

- 入栈顺序（内核栈）：
- **ds,es,fs,edx,ecx,ebx, call返回地址,eax(eip地址）, call返回地址**

中断返回后的eip

```
5  // 该函数系统中断调用(int 0x80)功能号__NR_execve 调用的函数。
6  // 参数：eip - 指向堆栈中调用系统中断的程序代码指针eip 处，参见kernel/system_call.s 程序
7  // 开始部分的说明；tmp - 系统中断调用本函数时的返回地址，无用；
8  // filename - 被执行程序文件名；argv - 命令行参数指针数组；envp - 环境变量指针数组。
9  // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。
10 int
11 do_execve (unsigned long *eip, long tmp, char *filename,
12         char **argv, char **envp)
```

“b”-%ebx, 保存filename

do_execve过程分析—i节点

```
14    struct m_inode *inode;     // 内存中I 节点指针结构变量。
15    struct buffer_head *bh;    // 高速缓存块头指针。
16    struct exec ex;            // 执行文件头部数据结构变量。
17    unsigned long page[MAX_ARG_PAGES];     // 参数和环境字符串空间的页面指针数组。
18    int i, argc, envc;
19    int e_uid, e_gid;          // 有效用户id 和有效组id。
20    int retval;                // 返回值。
21    int sh_bang = 0;           // 控制是否需要执行脚本处理代码。
22  // 参数和环境字符串空间中的偏移指针，初始化为指向该空间的最后一个长字处。
23    unsigned long p = PAGE_SIZE * MAX_ARG_PAGES - 4;
24
25  // eip[1]中是原代码段寄存器cs，其中的选择符不可以是内核段选择符，也即内核不能调用本函数。
26    if ((0xffff & eip[1]) != 0x000f)
27      panic ("execve called from supervisor mode");
28  // 初始化参数和环境串空间的页面指针数组（表）。
29    for (i = 0; i < MAX_ARG_PAGES; i++)    /* clear page-table */
30      page[i] = 0;
31  // 取可执行文件的对应i 节点号。
32    if (!(inode = namei (filename)))  /* get executables inode */
33      return -ENOENT;
```

1)找可执行程序文件i节点，与open_namei比？

do_execve过程分析—权限检测

```
34  // 计算参数个数和环境变
35    argc = count (argv)
36    envc = count (envp)
37
38  // 执行文件必须是常规文
39  restart_interp:
40    if (!S_ISREG (inode
41      {                   /* must be regular file */
42        retval = -EACCES;
43        goto exec_error2;
44      }
45  // 检查被执行文件的执行权限。根据其属性(对应i 节点的uid 和gid)，看本进程是否有权执行它。
46    i = inode->i_mode;                  // 取文件属性字段值。
```

在盘上和内存中的字段，共 32 字节

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。

2)从i节点中取文件属性信息，权限检测

do_execve过程分析—权限检测

```
48    // 如果文件的设置用户ID 标志（set-user-id）置位的话，则后面执行进程的有效用户ID（euid）就
49    // 设置为文件的用户ID，否则设置成当前进程的euid。这里将该值暂时保存在e_uid 变量中。
50    // 如果文件的设置组ID 标志（set-group-id）置位的话，则执行进程的有效组ID（egid）就设置为
51    // 文件的组ID。否则设置成当前进程的egid。
52    e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
53    e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
54    // 如果文件属于运行进程的用户，则把文件属性字右移6 位，则最低3 位是文件宿主的访问权限标志。
55    // 否则的话如果文件与运行进程的用户属于同组，则使属性字最低3 位是文件组用户的访问权限标志。
56    // 否则属性字最低3 位是其他用户访问该文件的权限。
57    if (current->euid == inode->i_uid)
58      i >>= 6;
59    else if (current->egid == inode->i_gid)
60      i >>= 3;
61    // 如果上面相应用户没有执行权并且其他用户也没有任何权限，并且不是超级用户，则表明该文件不
62    // 能被执行。于是置不可执行出错码，跳转到exec_error2 处去处理。
63    if (!(i & 1) && !((inode->i_mode & 0111) && suser ()))
64      {
65        retval = -ENOEXEC;
66        goto exec_error2;
67      }
```

2)从i节点中取文件属性信息，权限检测

do_execve过程分析—文件头

```
68  // 读取执行文件的第一块数据到高速缓冲区，若出错则置出错码，跳转到exec_error2 处去处理。
69    if (!(bh = bread (inode->i_dev, inode->i_zone[0])))
70      {
71        retval = -EACCES;
72        goto exec_error2;
73      }
74  // 下面对执行文件的头结构数据进行处理，首先让ex 指向执行头部分的数据结构。
75    ex = *((struct exec *) bh->b_data);    /* read exec-header *//* 读取执行头部分 */
76  ...
77  //shell
78  ...
79  // 释放该缓冲区。
80    brelse (bh);
```

3)根据i节点读取文件头信息，放入缓冲区

4)将缓冲区内容转化为ex程序头结构体指针？

把b_data数据复制到ex中，这说明文件的b_data存储内容的结构就是exec结构。

do_execve过程分析—合法性检测

```
81  // 下面对执行头信息进行处理。
82  // 对于下列情况，将不执行程序：如果执行文件不是需求页可执行文件(ZMAGIC)、或者代码重定位部分
83  // 长度a_trsize 不等于0、或者数据重定位信息长度不等于0、或者代码段+数据段+堆段长度超过50MB、
84  // 或者i 节点表明的该执行文件长度小于代码段+数据段+符号表长度+执行头部分长度的总和。
85    if (N_MAGIC (ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
86        ex.a_text + ex.a_data + ex.a_bss > 0x3000000 ||
87        inode->i_size < ex.a_text + ex.a_data + ex.a_syms + N_TXTOFF (ex))
88      {
89        retval = -ENOEXEC;
90        goto exec_error2;
91      }
92  // 如果执行文件执行头部分长度不等于一个内存块大小（1024 字节），也不能执行。转exec_error2。
93    if (N_TXTOFF (ex) != BLOCK_SIZE)
94      {
95        printk ("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
96        retval = -ENOEXEC;
97        goto exec_error2;
98      }
```

5)对可执行程序合法性检测

do_execve过程分析—挂接与复位

```
113  /* OK，下面开始就没有返回的地方了 */
114  // 如果原程序也是一个执行程序，则释放其i 节点，并让进程executable 字段指向新程序i 节点。
115    if (current->executable)
116      iput (current->executable);
117    current->executable = inode;
118  // 清复位所有信号处理句柄。但对于SIG_IGN 句柄不能复位，因此在322 与323 行
119  // if 语句: if (current->sa[I].sa_handler != SIG_IGN
120    for (i = 0; i < 32; i++)
121      current->sigaction[i].sa_handler = NULL;
122  // 根据执行时关闭(close_on_exec)文件句柄位图标志，关闭指定的打开文件，并复位该标志。
123    for (i = 0; i < NR_OPEN; i++)
124      if ((current->close_on_exec >> i) & 1)
125        sys_close (i);
126    current->close_on_exec = 0;
```

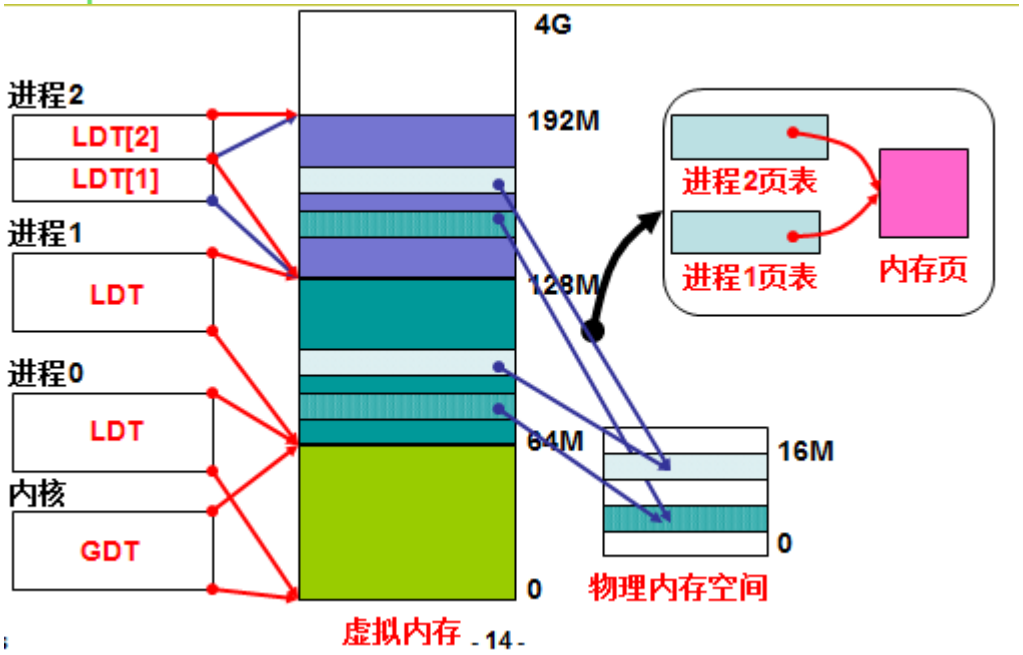
6)将当前进程可执行文件i节点更新

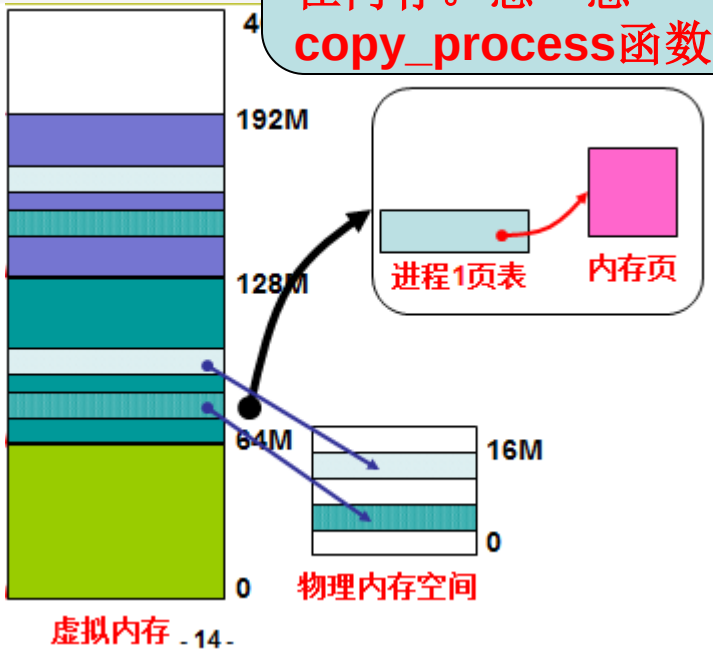
7)对文件和信号进行复位（归零）

do_execve过程分析—旧页表操作

```
127   // 根据指定的基地址和限长，释放原来程序代码段和数据段所对应的内存页表指定的内存块及页表本身。
128     // 此时被执行程序没有占用主内存区任何页面。在执行时会引起内存管理程序执行缺页处理而为其申请
129     // 内存页面，并把程序读入内存。
130     free_page_tables (get_base (current->ldt[1]), get_limit (0x0f));
131     free_page_tables (get_base (current->ldt[2]), get_limit (0x17));
132   // 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器的标志。
133     if (last_task_used_math == current)
134       last_task_used_math = NULL;
135     current->used_math = 0;
```

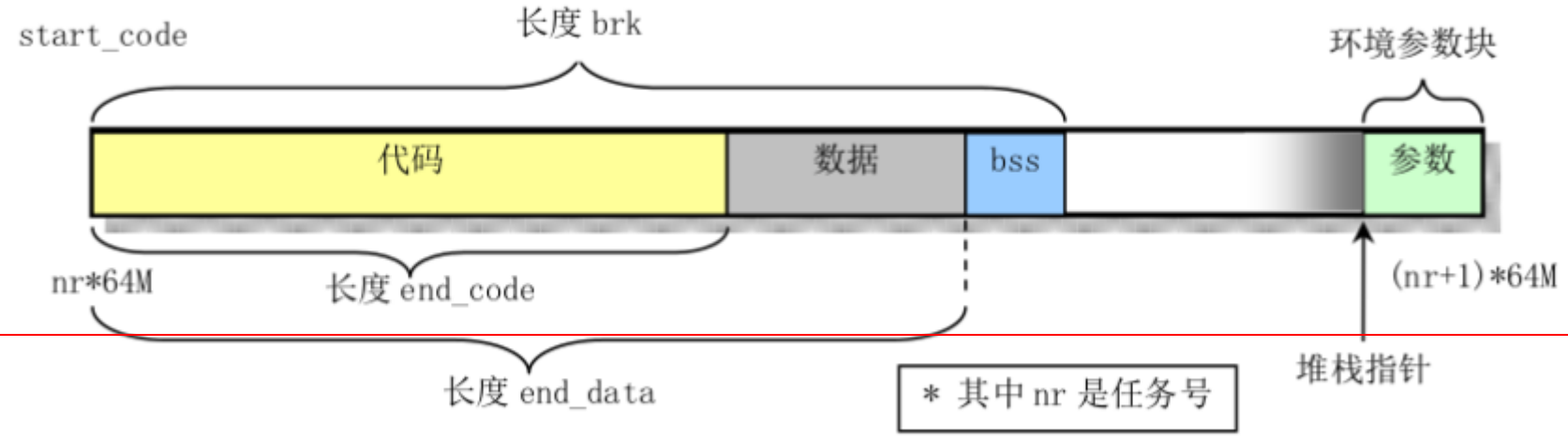
7)释放代码段和数据段页表，代码和数据段未在内存。想一想copy_process函数？





- 14 -

do_execve过程分析—段设置（限长）




```
136
137
138
139
140
141
142
143
144     current->brk = ex.a_bss +
145       (current->end_data = ex.a_data + (current->end_code = ex.a_text));
146   // 设置进程堆栈开始字段为堆栈指针所在的页面，并重新设置进程的有效用户id 和有效组id。
147     current->start_stack = p & 0xfffff000;
148     current->euid = e_uid;
149     current->egid = e_gid;
150   // 初始化一页bss 段数据，全为零。
151     i = ex.a_text + ex.a_data;
152     while (i & 0xfff)
153       put_fs_byte (0, (char *) (i++));
```

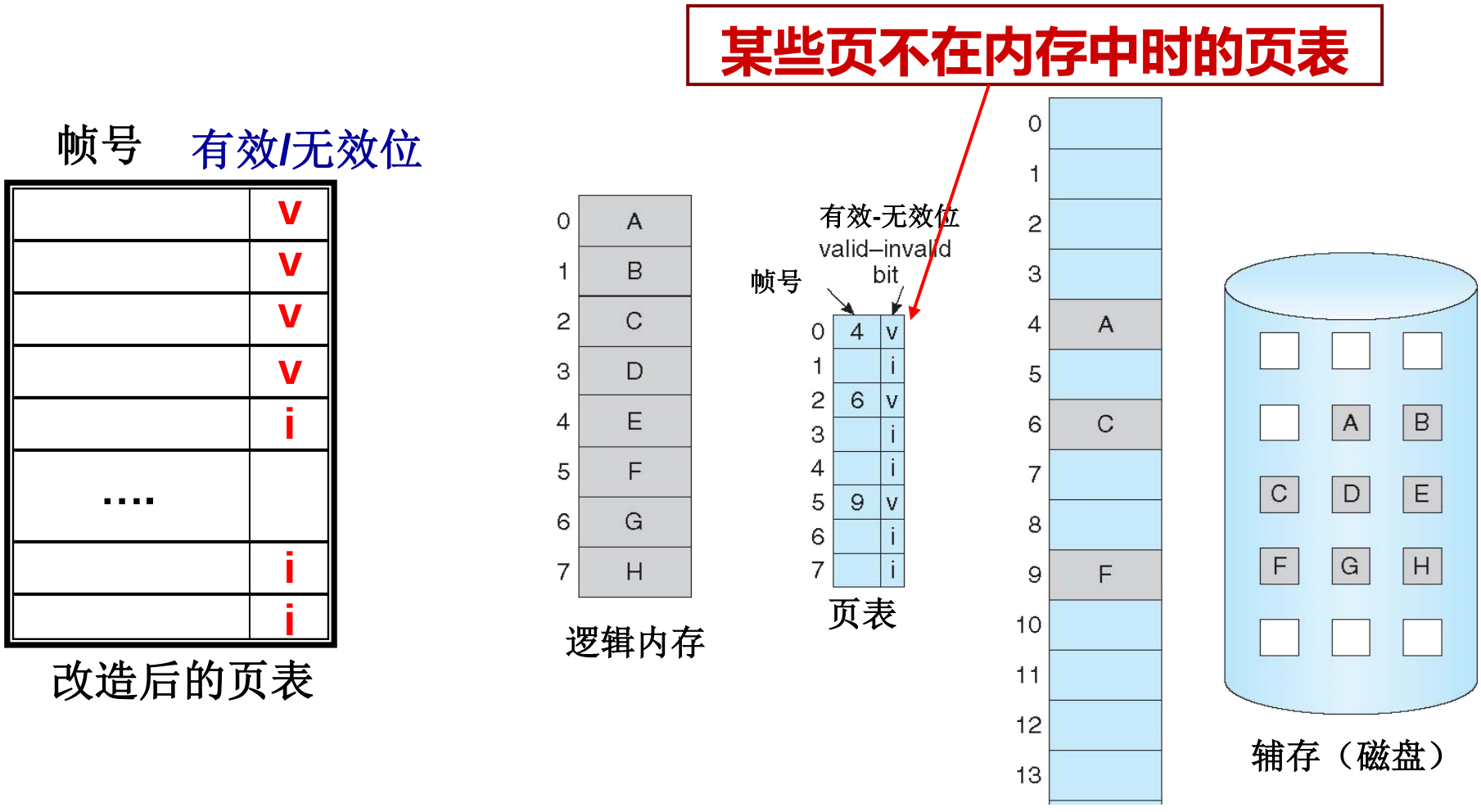
8)设置新代码段数据段和栈地址信息

do_execve过程分析—入口地址

```
154  // 将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将堆栈指针替换
155  // 为新执行程序的堆栈指针。返回指令将弹出这些堆栈数据并使得CPU 去执行新的执行程序，因此不会
156  // 返回到原调用系统中断的程序中去了。
157    eip[0] = ex.a_entry;      /* eip, magic happens :-) *//* eip，魔法起作用了 */
158    eip[3] = p;               /* stack pointer *//* esp，堆栈指针 */
```

9)修改do_execve返回时的**eip**和**esp**为新进程的入口和栈，则返回后后进入新进程。都在新进程的地址空间。但代码和数据还未拷入！！

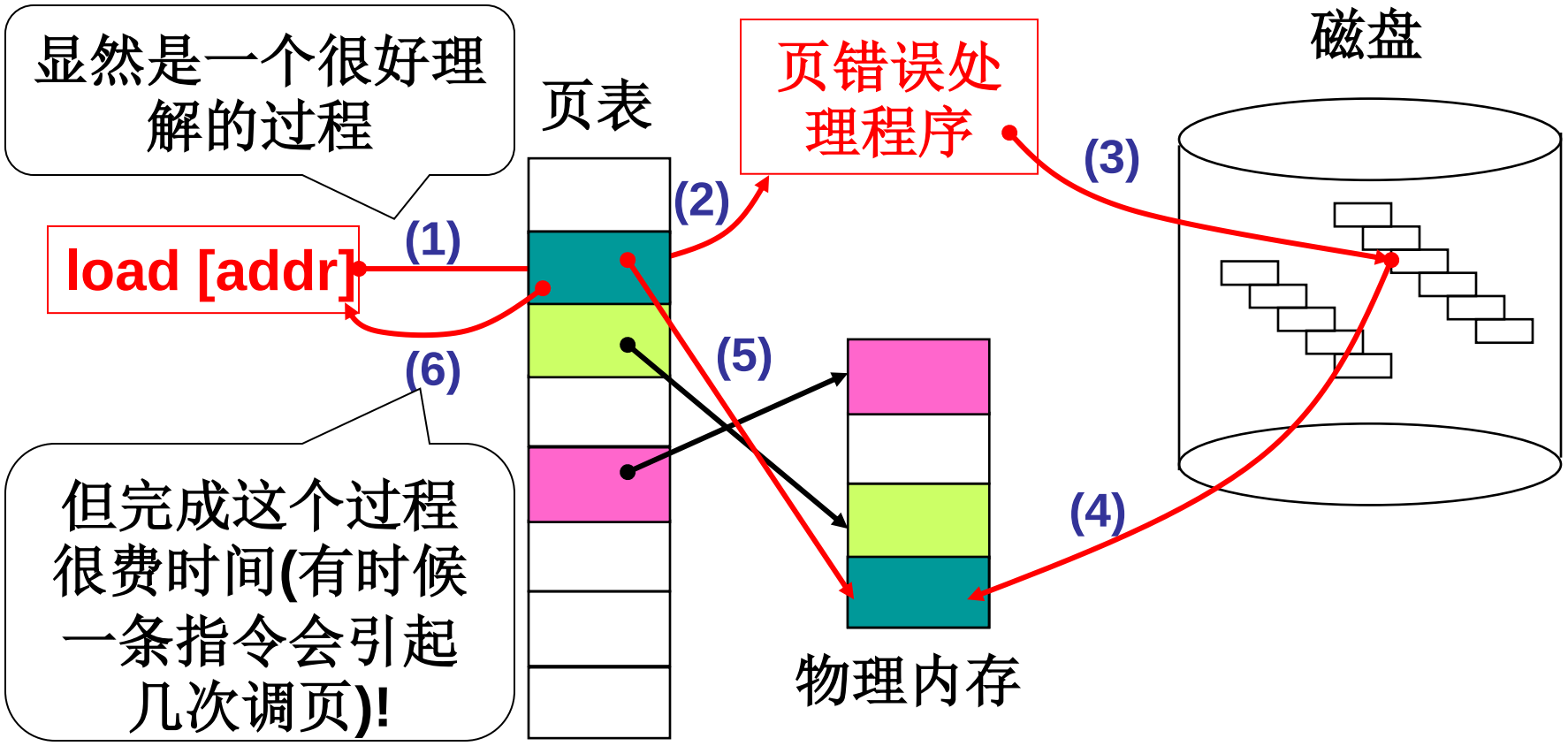
回顾：如何记录页是否在内存？



- 改造页表，页表项增加“有效/无效位”

请求调页过程

- 当访问没有映射的线性地址时…



请求调页过程-- do_no_page

```
// 处理缺页异常的函数体。address是事发地点线性地址。
void do_no_page (unsigned long error_code,unsigned long address)
{
int nr[4]; unsigned long tmp; unsigned long page; int block,i;
address &= 0xfffff000;  //4k对齐
/*  address是线性页面地址，current->start_code是进程在线性地址空间的起始地址， 两个相减tmp是发生缺页时的相对于代码段起始地址的偏移*/
tmp = address - current->start_code;
// 缺页中断有多种情况，第一种如下：
// current->executable == 0 表明当前进程没有可执行文件。
// tmp>=current_end_data，表示缺页的逻辑地址大于进程的代码段和数据段之和。这两种情况都对应着第一种缺页
// 即当前缺页是由于进程压栈（为堆或栈中数据寻找新的页面）造成的，因此 直接调用get_empty_page为进程申请一页新物理内存即可。
if (!current->executable || tmp >= current->end_data) {
        get_empty_page(address);
        return;
}
//*若走到这里，则说明缺页异常不是由于进程压栈造成，那肯定就是执行exevce导致。先尝试share_page。即先看当前进程的executable是否被其他进程同样引用*/
if (share_page(tmp))
```

请求调页过程-- do_no_page

```
if (share_page(tmp))        return;
// 要是没能share成功，退而求其次吧。只能为该进程注册一页新的物理内存，并且读取 相应内容到这页物理内存中了。

if (!(page = get_free_page()))  oom(); //申请一个页框（帧）
block = 1 + tmp/BLOCK_SIZE;     /* 第一块文件头*/
for (i=0 ; i<4 ; block++,i++)  nr[i] = bmap(current->executable,block);
```

1)首先申请一个页框（帧）
2)计算address地址处内存在可执行文件中的块序号
3)由address得到缺页逻辑地址tmp，然后由tmp/BLOCK_SIZE即得到tmp逻辑地址在可执行文件中的块序号
4)+1是考虑到文件头占用的开头一个block
5)接下去？读取address地址处4096字节内容到内存中刚申请的page物理页面中。一个block等于两个扇区（1k），四个block等于一个页大小（4k）

请求调页过程-- do_no_page

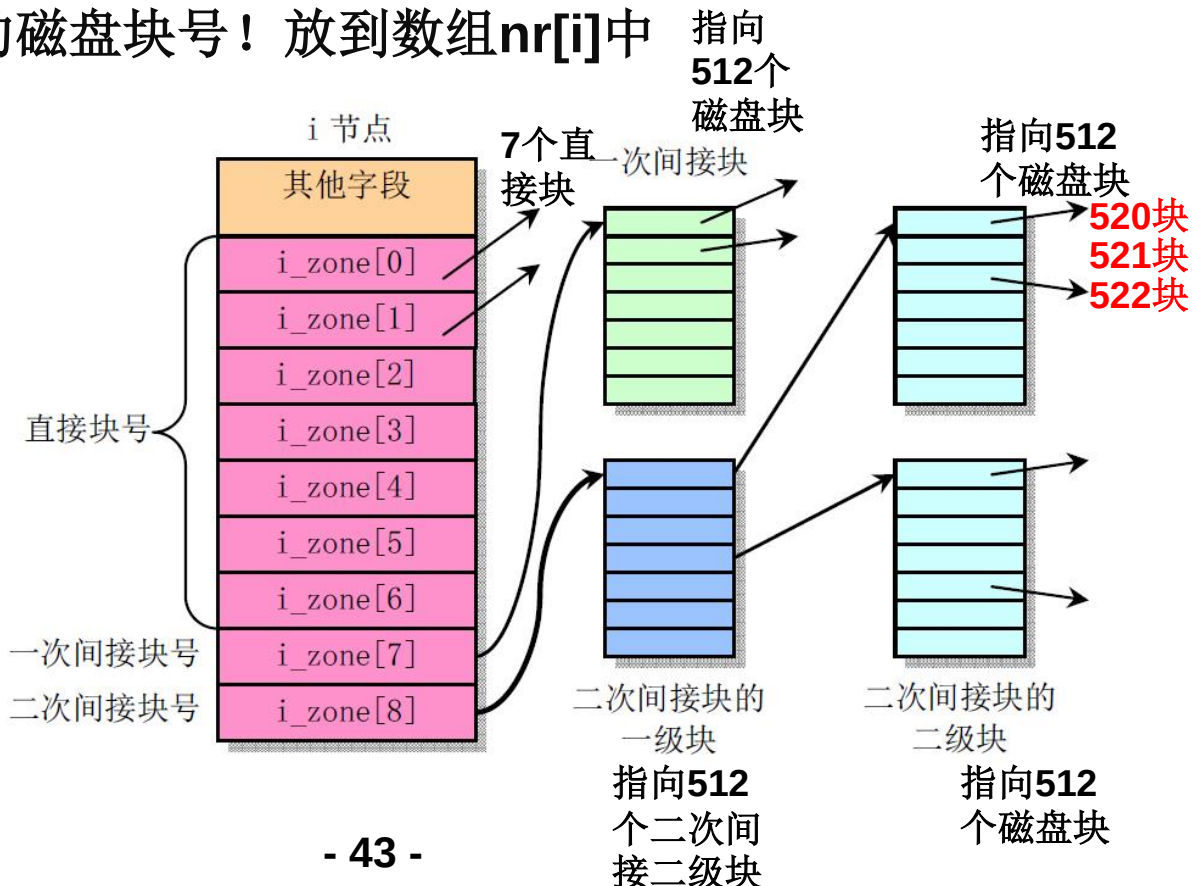
for (i=0 ; i<4 ; block++,i++) nr[i] = bmap(current->executable,block);

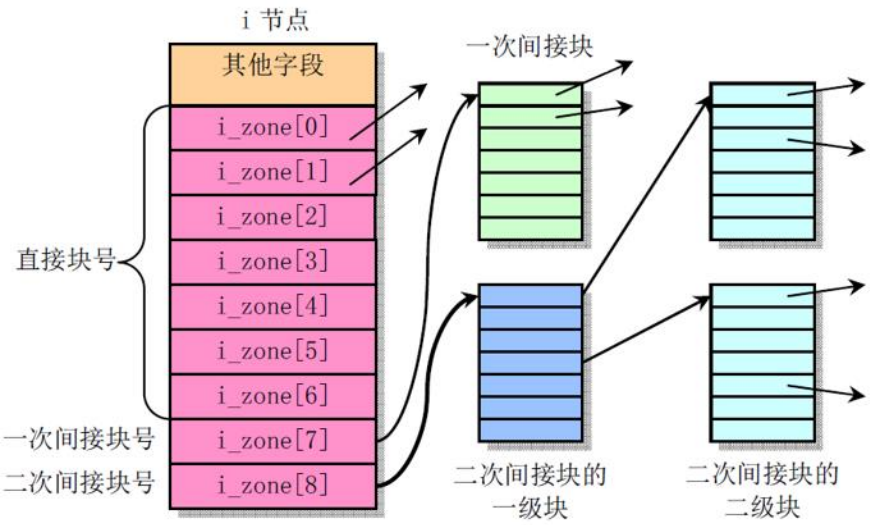
磁盘块号采用两个字节存储；一个间接块占一个磁盘块1k（两个扇区）。

i节点中可以访问的总块数：直接块7+一次间接512+二次间接512*512

bmap找到block对应的磁盘块号！放到数组nr[i]中

一个page有4k，而一个block为1k，因此要读取从block开始的4个连续数据块。

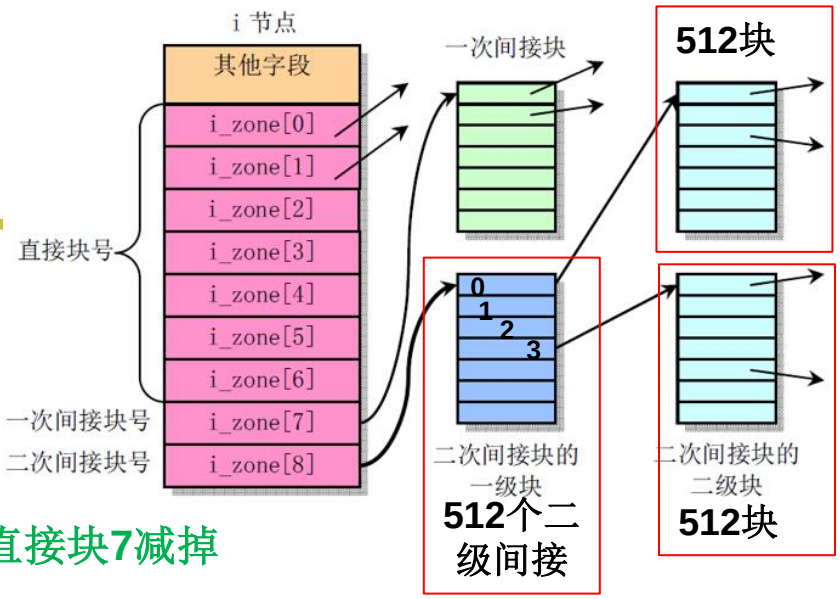





```
// create是否创建新块，1创建，0不创建；block文件
static int _bmap(struct m_inode * inode,
        int block,int create)
{
        struct buffer_head * bh;
        int i;

        if (block<0) panic("_bmap: block<0");

        if (block >= 7+512+512*512) panic("_bmap: block>big");

        if (block<7) { ... ...  //说明block序号0-6直接快
                return inode->i_zone[block]; }

        block -= 7;    //减掉直接块
        if (block<512) {... ...
                if (!(bh = bread(inode->i_dev,inode->i_zone[7]))) return 0;
                //一次间接存储的物理盘块号
                i = ((unsigned short *) (bh->b_data))[block];
    ... ...
                return i;
        }
 ... ...
}
```

char *b_data; //
指向缓冲块数据区

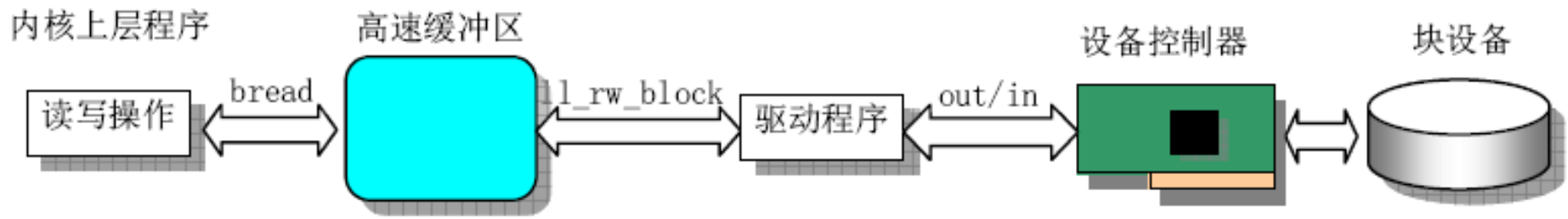
bh->b_data
(unsigned short *) (bh->b_data)
[block];




```
// create是否创建新块，1创建，0不创建；block文件
static int _bmap(struct m_inode * inode,
        int block,int create)
{
    … …
        block -= 512;  //减掉一次间接块，之前已将直接块7减掉
    … …
        if (!(bh=bread(inode->i_dev, inode->i_zone[8] ))) return 0;

        // block>>9右移9位，除512(0x0100)，找到二次间接二级块号
        i = ((unsigned short *)bh->b_data) [block>>9] ;
    … …//i是二次间接二级块的磁盘块号
        if (!(bh=bread(inode->i_dev,i)))  return 0;  //读取二次间接二级块存入bh

        // block&511 (0xff)，二次间接二级块内偏移，读出物理盘块号
        i = ((unsigned short *)bh->b_data)[block&511];
    … …
        return i;
}
}
```

请求调页过程-- do_no_page

```
/* 调用bread_page函数读取这4个连续block的数据，放入page中*/
bread_page(page,current->executable->i_dev,nr);

// 以下代码用来完成对多于读出来的无用字节清零。这主要是针对可执行文件中 tmp后面的内容不足一页的情况下，读出的一页内容就有很多字节是无用的， 这种情况下就要把这些内容给清零。
i = tmp + 4096 - current->end_data;
        // 将tmp指向page物理页面的尾端，这里是要清除位置的最后面。
tmp = page + 4096;
        // 当上面计算的i为负值时清除代码是直接跳过的，很有技巧…
        // 若i为正值，则用i来计数开始从page页面的尾端开始逐个字节清除。
while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
}
```

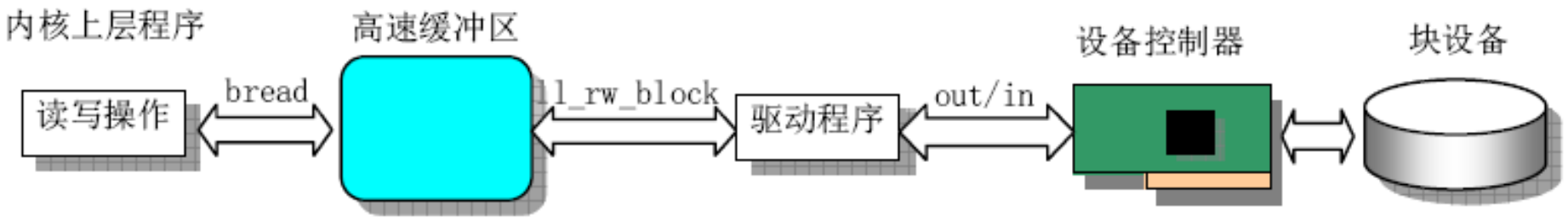



```
{
        struct buffer_head * bh[4];
        int i;

        for (i=0 ; i<4 ; i++)
                if (b[i]) {
                        if ((bh[i] = getblk(dev,b[i])))
                                if (!bh[i]->b_uptodate)
                                        ll_rw_block(READ,bh[i]);
                } else
                        bh[i] = NULL;
        for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
                if (bh[i]) {
                        wait_on_buffer(bh[i]);
                        if (bh[i]->b_uptodate)
                                COPYBLK((unsigned long) bh[i]->b_data,address);
                        brelse(bh[i]);
                }
}
```

定义四个bh缓冲块，先查缓冲，缓冲没有再读盘

bread_page与bread

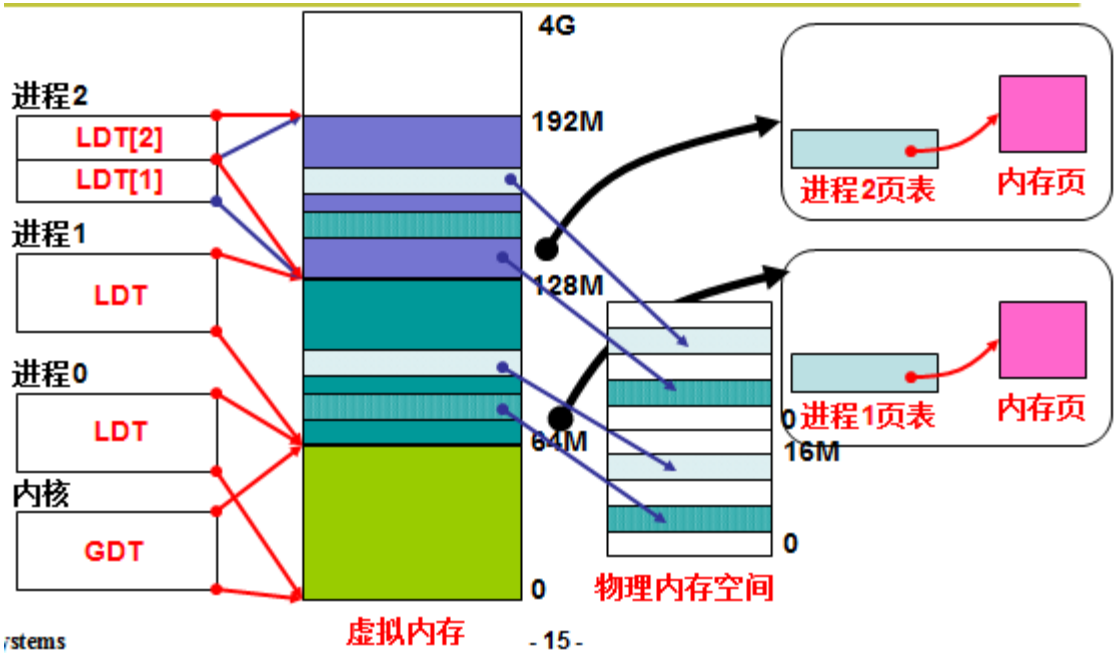
```
struct buffer_head * bread(int dev,int block)
{
        struct buffer_head * bh;

        if (!(bh=getblk(dev,block)))
                panic("bread: getblk returned NULL\n");
        if (bh->b_uptodate)  //缓冲区与硬盘的是不是一致的？
                return bh;
        ll_rw_block(READ,bh);
        wait_on_buffer(bh);
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return NULL;
}
```



请求调页过程-- do_no_page

- /* 最后调用put_page来将当前进程address线性地址空间页面与page处物理页面挂接起来，建立页表填充页框。（返回后会重新执行刚才导致缺页异常的那句代码*/
- **if (put_page(page,address)) return;**
- // 执行到这表示put_page失败，这表示此次load on demand失败了。善后时要 free刚才本函数申请的一页物理内存。并且提示out of memory。

```
free_page(page);
oom();
}
```



请求调页过程-- put_page

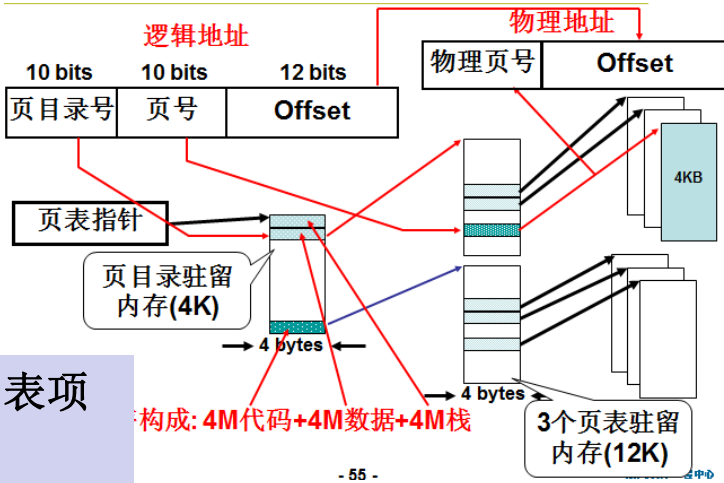
```
unsigned long put_page(unsigned long page, unsigned long address)
{
    unsigned long tmp, *page_table;
    … …
    //在页目录项中的位置，*page_table存储页表地址
    page_table = (unsigned long *) ((address>>20) & 0xffc); //转为指针，页目录在哪？0地址
    //该页目录项已经存在
    if ( (*page_table) &1)
            page_table = (unsigned long *) (0xfffff000 & *page_table);
    else {  //该页表不存在，申请新页表
            if (!(tmp=get_free_page()))
                    return 0;
            *page_table = tmp|7;//目录项指向新页表
            //重新设置指page_table向新页表
            page_table = (unsigned long *) tmp;
    }
    page_table[(address>>12) & 0x3ff] = page | 7;
    return page;
}
```

(address>>22)<<2，先得到目录项，每项四个字节，转化为地址

page_table先作为页目录项，后又作为页表项

[(address>>12) & 0x3ff]

页表项4k对齐，一个页表只能有1k个项

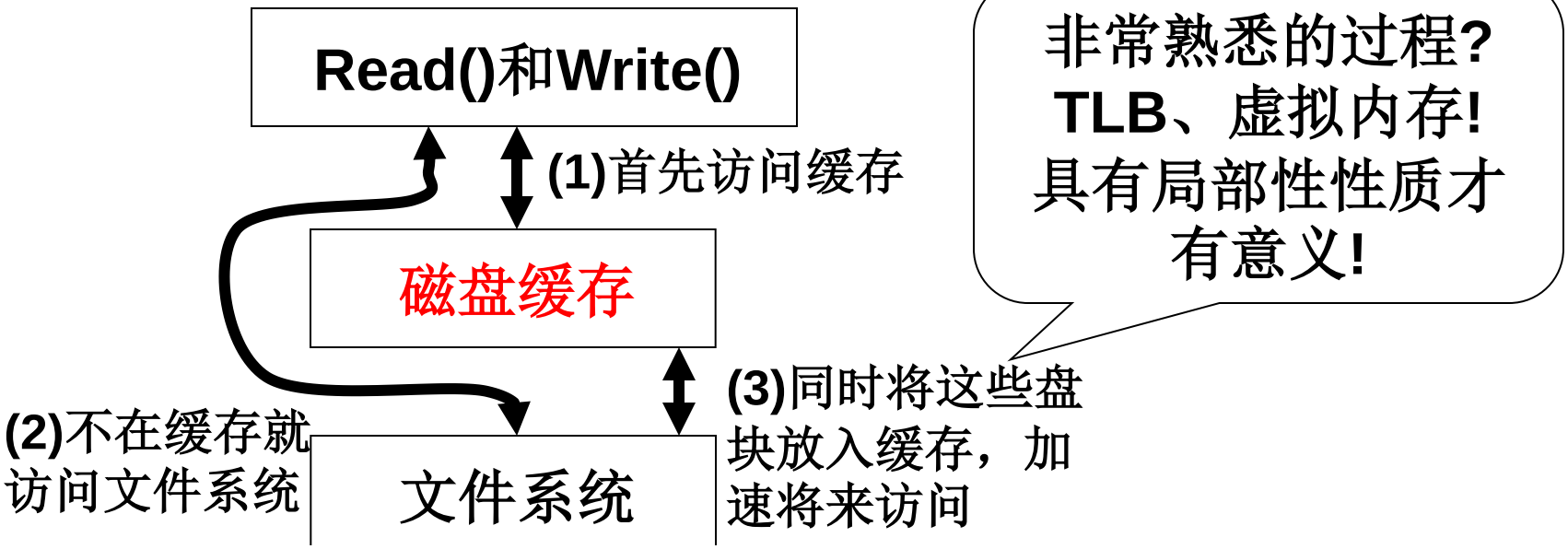


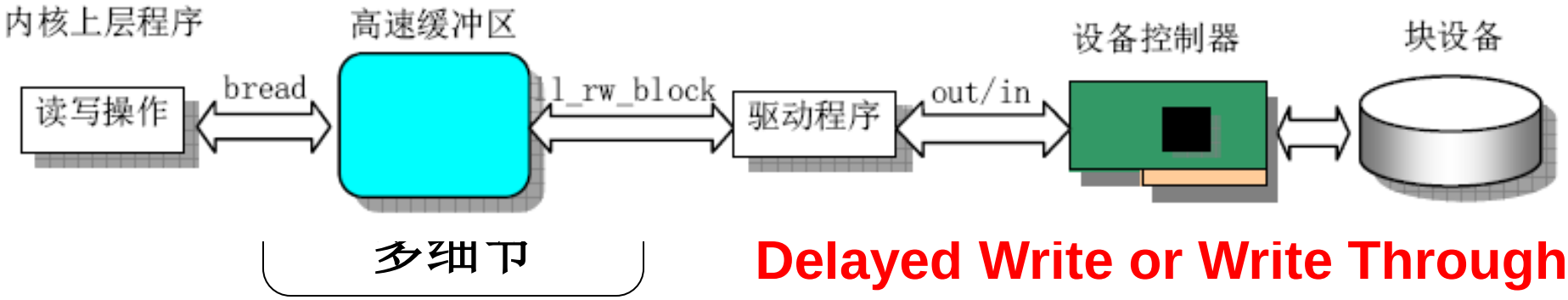
将一个可执行文件变成进程？

- fork一个子进程（PCB、内存）
- do_execve让子进程执行新的程序（找到文件inode，读文件头，更新进程控制块中的代码段、数据段、栈段地址，释放原页表）
- do_no_page根据缺页地址address按需调页，put_page建立页表

基于缓冲区的磁盘访问

- 在内存中缓存磁盘上的部分(很少部分)盘块

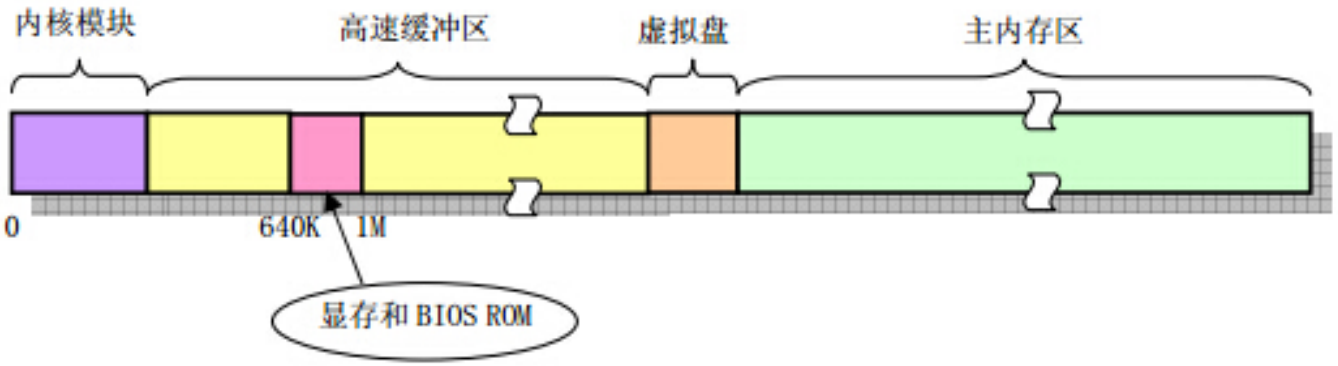




Delayed Write or Write Through

基于缓冲区的磁盘访问

- 在Linux0.11初始化时，会根据内存大小来初始化高速缓冲区大小
- 缓冲区位于内核end之后，buffer_memory_end值之前 4M内存处，一共可以划出3千多个逻辑块
- 缓冲区被分成一些缓冲块，每个缓冲块大小为1024字节，即两个扇区
- 在缓冲区中包含struct buffer_head结构体（缓冲头结构）
- struct buffer_head *start_buffer = (struct buffer_head *) &end // 可见struct buffer_head结构体起始于缓冲区的低地址（类似于inode头集中存储）
- 在buffer_init函数里会对start_buffer之后的一些buffer_head结构体进行初始化。最后的结果如下图：



基于缓冲区的磁盘访问

还记得下面的操作吗？
struct buffer_head * bh[4];
bh[i]->b_data

- 缓冲块管理buffer_head结构体：

```
struct buffer_head {
        char *b_data;                   // 指向缓冲块数据区
        unsigned long b_blocknr;        // 块号
        unsigned short b_dev;           // 设备号
        unsigned char b_uptodate;       // 更新标示：表示数据是否已更新
        unsigned char b_dirt;           // 修改标志：0--未修改， 1--已修改
        unsigned char b_count;          // 使用该块的进程数（在
        unsigned char b_lock;           // 缓冲区是否被锁定，0--unlocked
，1--locked
        struct task_struct *b_wait;     // 等待缓冲区解锁的进程
        struct buffer_head *b_prev;     // hash表的前一个
        struct buffer_head *b_next;     // hash表的后一个
        struct buffer_head *b_prev_free;  // free表的前一个
        struct buffer_head *b_next_free;  // free表的后一个
}
```

- 管理方式：hash表（设备号^逻辑块号） + 双向链表（所有的buffer_head）

```
struct buffer_head * hash_table[NR_HASH];
static struct buffer_head * free_list;
```

基于缓冲区的磁盘访问

- 刚初始化的时候所有的缓存块依次连接成一个双向的循环链表

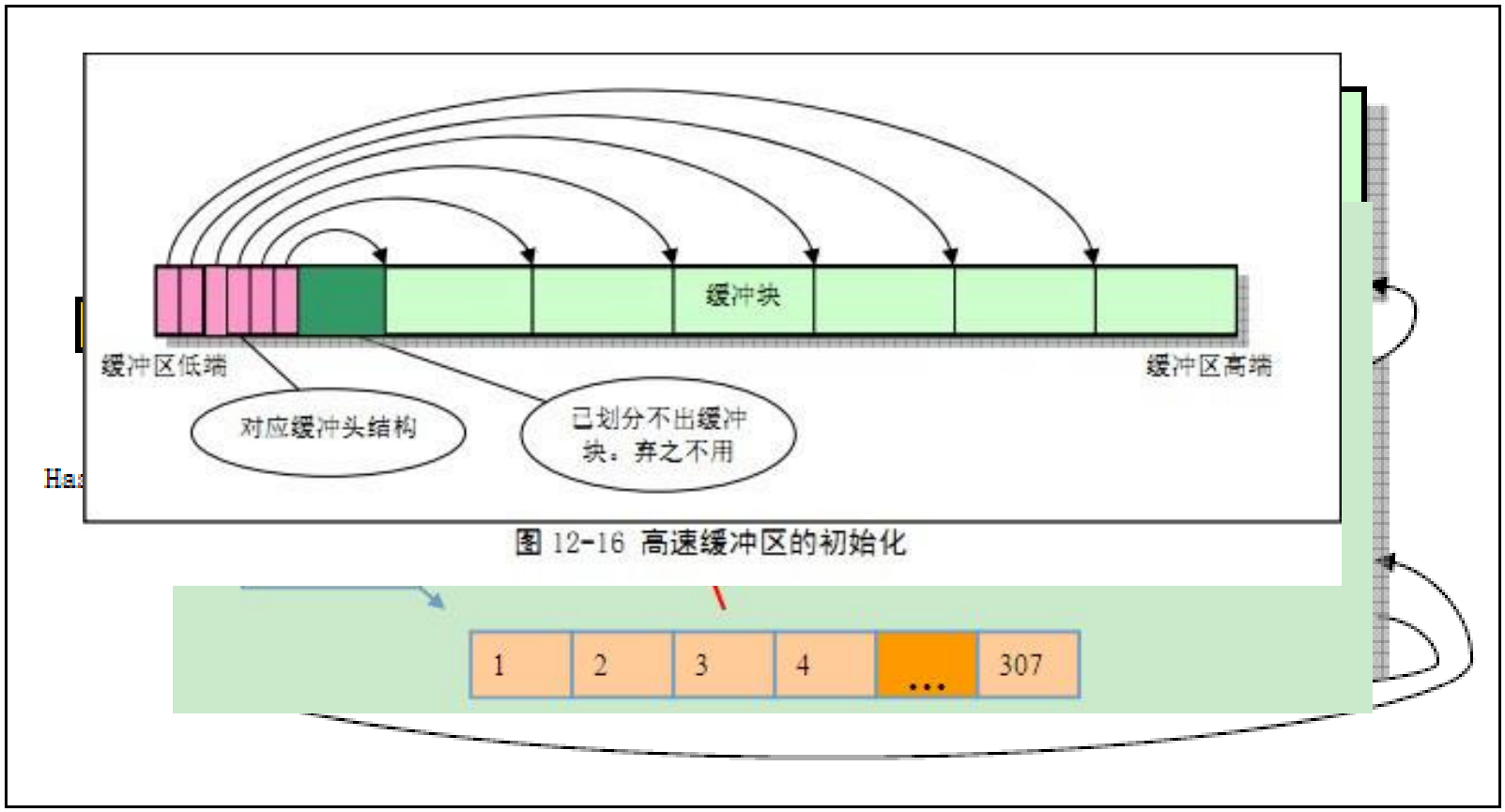
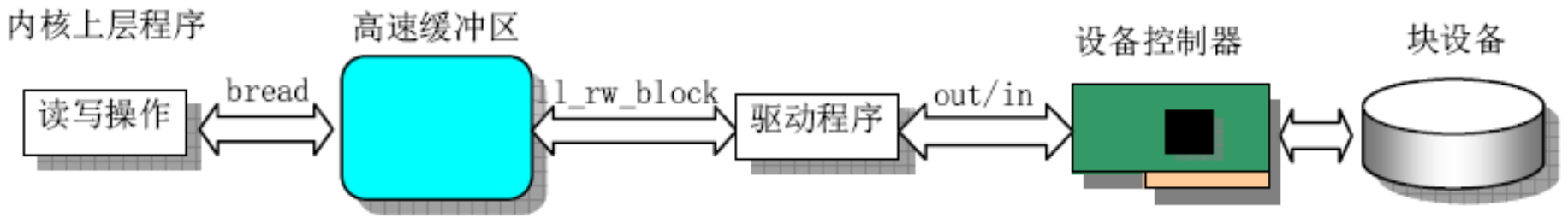


图 12-16 高速缓冲区的初始化

图 9-11 空闲缓冲块双向循环链表结构

bread_page与bread

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

        if (!(bh=getblk(dev,block)))   //先从缓冲区找一个缓冲块
                panic("bread: getblk returned NULL\n");
        if (bh->b_uptodate)  return bh;   //缓冲块数据有效，已更新
        ll_rw_block(READ,bh);   //从块设备读取数据
        wait_on_buffer(bh);
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return NULL;
}
```

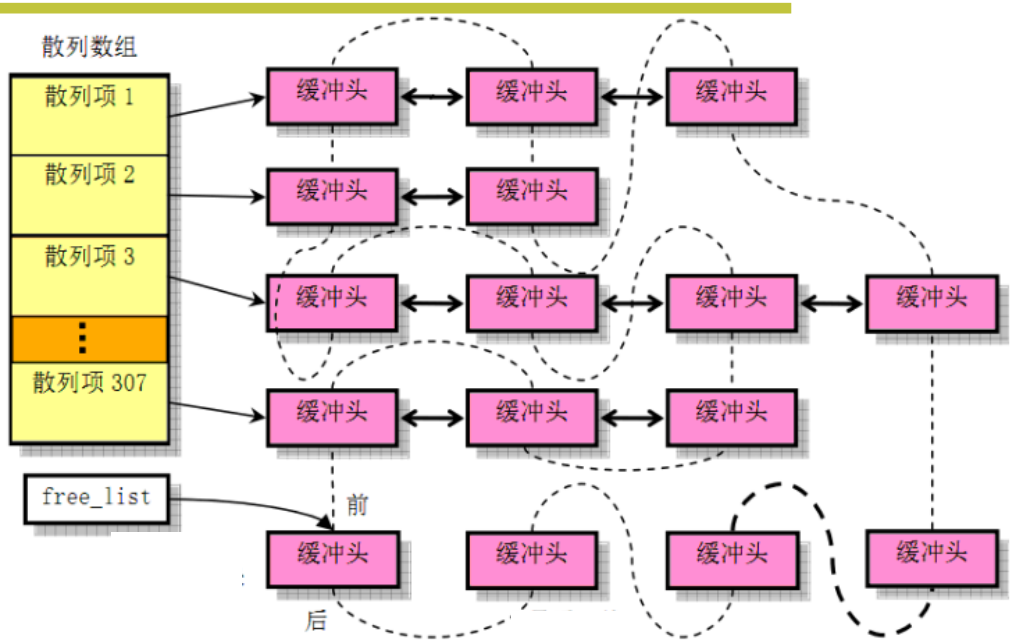


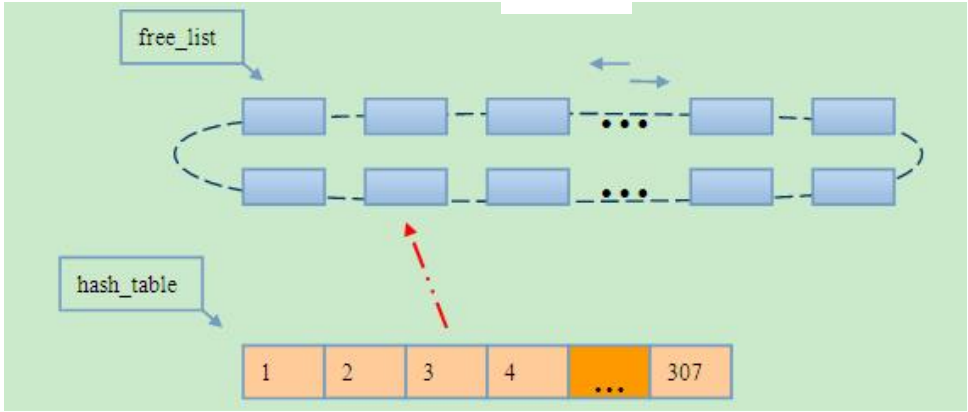
基于缓冲区的磁盘访问

- 这个结构体的最后四个指针，表示了对buffer_head的管理。
- 通过两种方式对buffer_head进行管理
- 1）free_list指针
- free_list指针指向第一个空闲的缓冲头（buffer_head），全部的空闲buffer_head通过b_prev_free和b_next_free构成一个双向循环链表。free_list指针会在每次insert_into_queues和remove_from_queues两个函数中改变。
- 2）hash数组
- struct buffer_head *hash[NR_HASH], NR_HASH的值为307，对一个正在使用的buffer_head，根据其中的设备号和逻辑块号异或计算出该buffer_head的hash值，并把该buffer_head放入hash数组中去。

基于缓冲区的磁盘访问

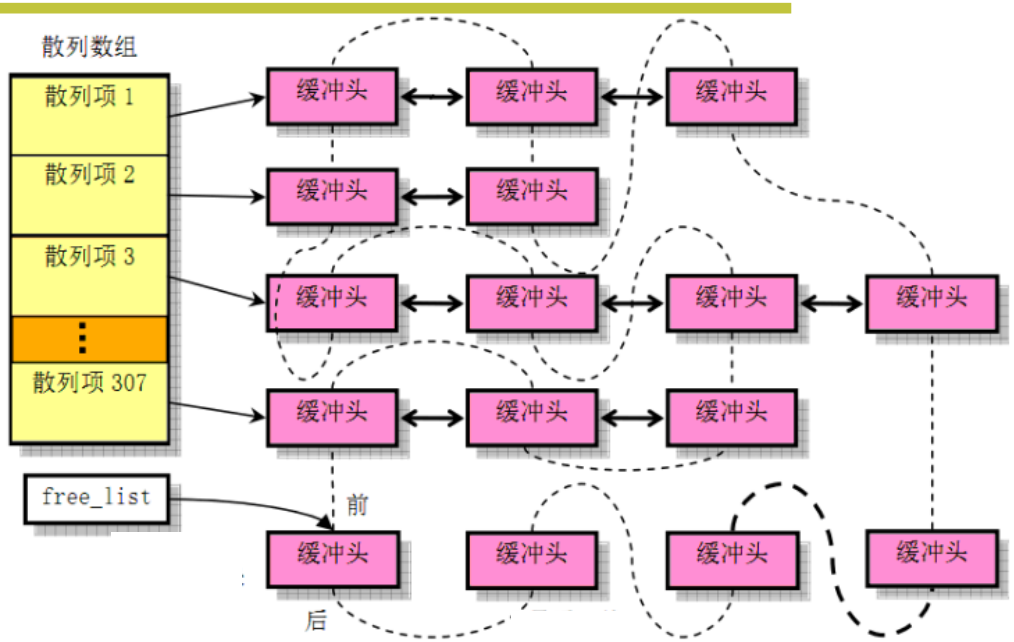
- getblk函数首先将根据设备号和逻辑块号按照hash函数得到hash值key（假定设备号为0x0201，逻辑块号为1500，key=171），其实该key值即为hash_table指针数组的索引号
- 然后检索hash_table，看hash_table [171]中是否指向一个有效的缓冲块，如果存在则直接返回，如果不存在，则首先会扫描可能因hash冲突被移到后面的缓存块（find_buffer，tmp->b_next）

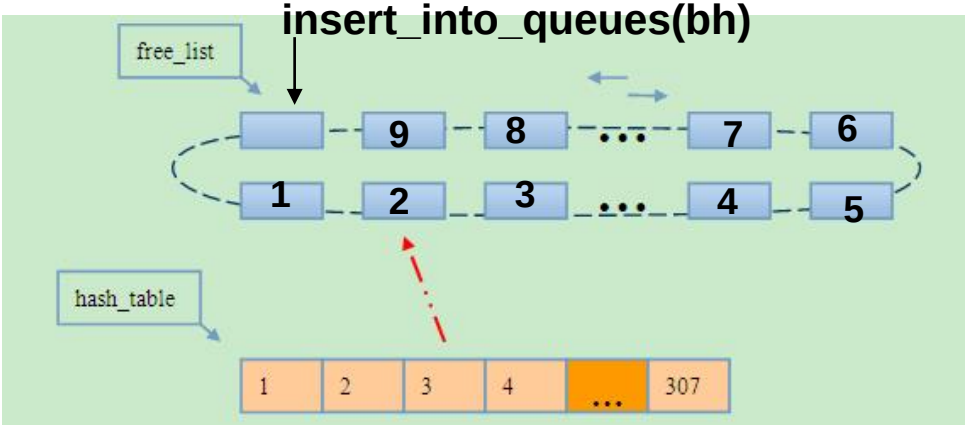




基于缓冲区的磁盘访问

- 如果没找到，从3千多个缓冲区中找到一个空闲的缓冲块，并将该缓冲块的指针赋值给hash_table [171]
- **remove_from_queues(bh)**
  bh->b_dev=dev;
  bh->b_blocknr=block;
  insert_into_queues(bh);
- 如果没有空闲块，置换。将缓冲块一道free_list链表的最后去（最近最少使用算法）
- 当下次再对该设备块进行操作时，就可以根据hash值直接找到该块。





基于缓冲区的磁盘访问

```
static inline void remove_from_queues(struct buffer_head * bh)
{
/* remove from hash-queue */
        if (bh->b_next)
                bh->b_next->b_prev = bh->b_prev;
        if (bh->b_prev)
                bh->b_prev->b_next = bh->b_next;
        if (hash(bh->b_dev,bh->b_blocknr) == bh)
                hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
/* remove from free list */
        if (!(bh->b_prev_free) || !(bh->b_next_free))
                panic("Free block list corrupted");
        bh->b_prev_free->b_next_free = bh->b_next_free;
        bh->b_next_free->b_prev_free = bh->b_prev_free;
        if (free_list == bh)
                free_list = bh->b_next_free;
}
```

基于缓冲区的磁盘访问

```
static inline void insert_into_queues(struct buffer_head * bh)
{
/* put at end of free list */
        bh->b_next_free = free_list;
        bh->b_prev_free = free_list->b_prev_free;
        free_list->b_prev_free->b_next_free = bh;
        free_list->b_prev_free = bh;
/* put the buffer in new hash-queue if it has a device */
        bh->b_prev = NULL;
        bh->b_next = NULL;
        if (!bh->b_dev)
                return;
        bh->b_next = hash(bh->b_dev,bh->b_blocknr);
        hash(bh->b_dev,bh->b_blocknr) = bh;
        bh->b_next->b_prev = bh;
}
```

基于缓冲区的磁盘访问

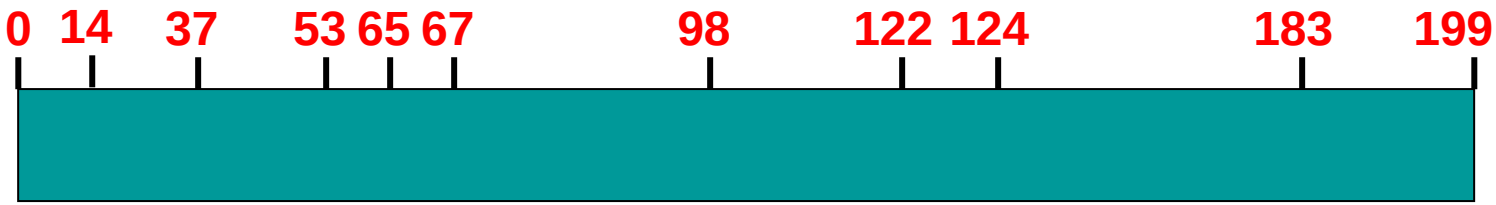
- **ll_rw_block()函数**
  在实际要读取块设备时，最终都会执行到**ll_rw_block**函数，该函数根据**buffer_head**生成一个**request**，并将请求加入设备的请求队列中去。

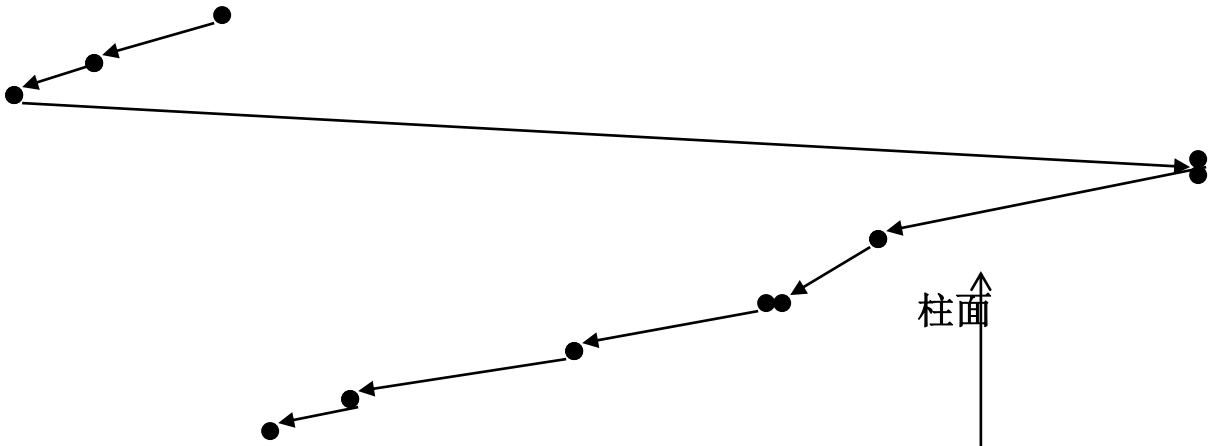
```
1.  void ll_rw_block(int rw, struct buffer_head *bh)
2.  {
3.          unsigned int major;
4.          if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
5.              printk("Trying to read nonexistent block-devicenr");
6.              return;
7.          }
8.          make_request(major, rw, bh);    // 生成请求
9.  }
```

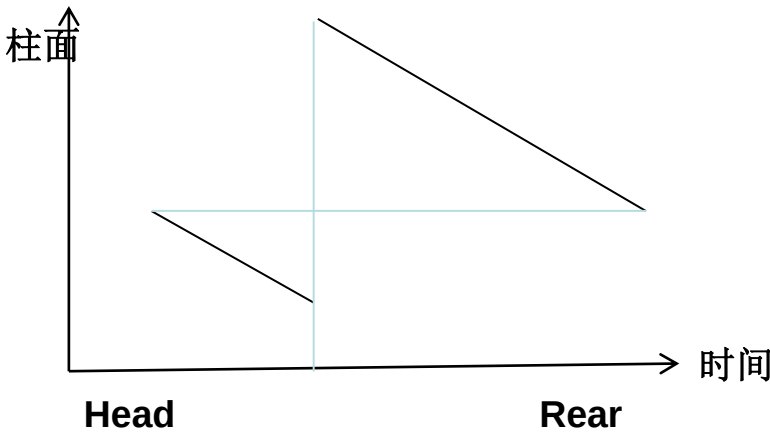
回顾：C-LOOK磁盘调度

- 继续该实例: 磁头开始位置=53;

  请求队列=98, 183, 37, 122, 14, 124, 65, 67







1）磁道请求队列的的形式

2）新磁道请求如何入队列

53-37-14- 183-124-122-98-67-65

磁盘访问调度

```
// kernel\blk_drv\ll_rw_blk.c
static void make_request(int major,int rw, struct buffer_head * bh)
{
/* 初始化请求项的值 */
    req->dev = bh->b_dev;
    req->cmd = rw;
    req->errors=0;
    req->sector = bh->b_blocknr<<1;
    req->nr_sectors = 2;
    req->buffer = bh->b_data;
    req->waiting = NULL;
    req->bh = bh;
    req->next = NULL;
    add_request(major+blk_dev,req);    // 将请求项放入设备的请求队列中
}
```

磁盘访问调度

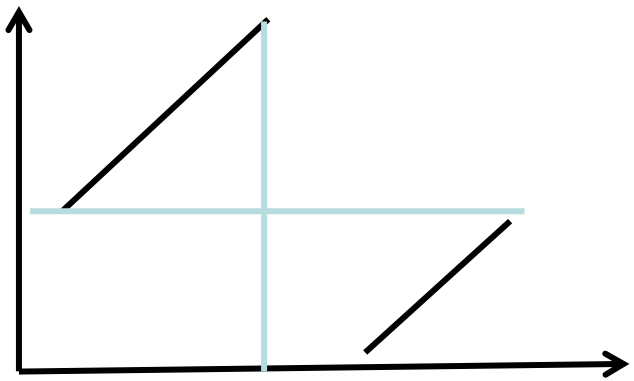
```
// kernel\blk_drv\blk.h 中的宏定义函数
#define IN_ORDER(s1,s2) \
((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd && \
((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
(s1)->sector < (s2)->sector)))
```

```
#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 )
)
```

（1）写请求小于读请求。
（2）若请求类型相同，低设备号小于高设备号。
（3）若请求同一设备，低扇区号小于高扇区号。

磁盘访问调度

```
// kernel\blk_drv\ll_rw_blk.c
static void add_request(struct blk_dev_struct * dev, struct request * req)
{  ……
   //只有一个请求， tmp->next 空，直接放到tmp后，存在至少两个请求执行for内部
   for ( ; tmp->next ; tmp=tmp->next)
      if (   ( IN_ORDER(tmp,req)   ||  ! IN_ORDER(tmp,tmp->next) ) &&
       IN_ORDER(req,tmp->next)    )
          break;

   // 将请求插入tmp之后。
   req->next=tmp->next;
   tmp->next=req;
   sti();
}
```



IN_ORDER(tmp,req) && IN_ORDER(req, tmp->next) → tmp<req< tmp->next

||

IN_ORDER(tmp->next,tmp)) && IN_ORDER(req,tmp->next) → req< tmp->next<tmp

操作系统中的进程、内存、文件系统

- 进程带动内存的使用与管理，进程的段页内存使用、部分加载、按需调页、换入换出。
- 文件系统支撑进程本身的创建、按需调页、换入换出以，以及进程对各类文件的使用。

操作系统的地位

操作系统让你对组成原理、汇编语言、C语言、数据结构与算法、编译原理有一个综合运用的机会