第32回 | 加载根文件系统

Original 闪客 低并发编程 2022-04-06 17:30

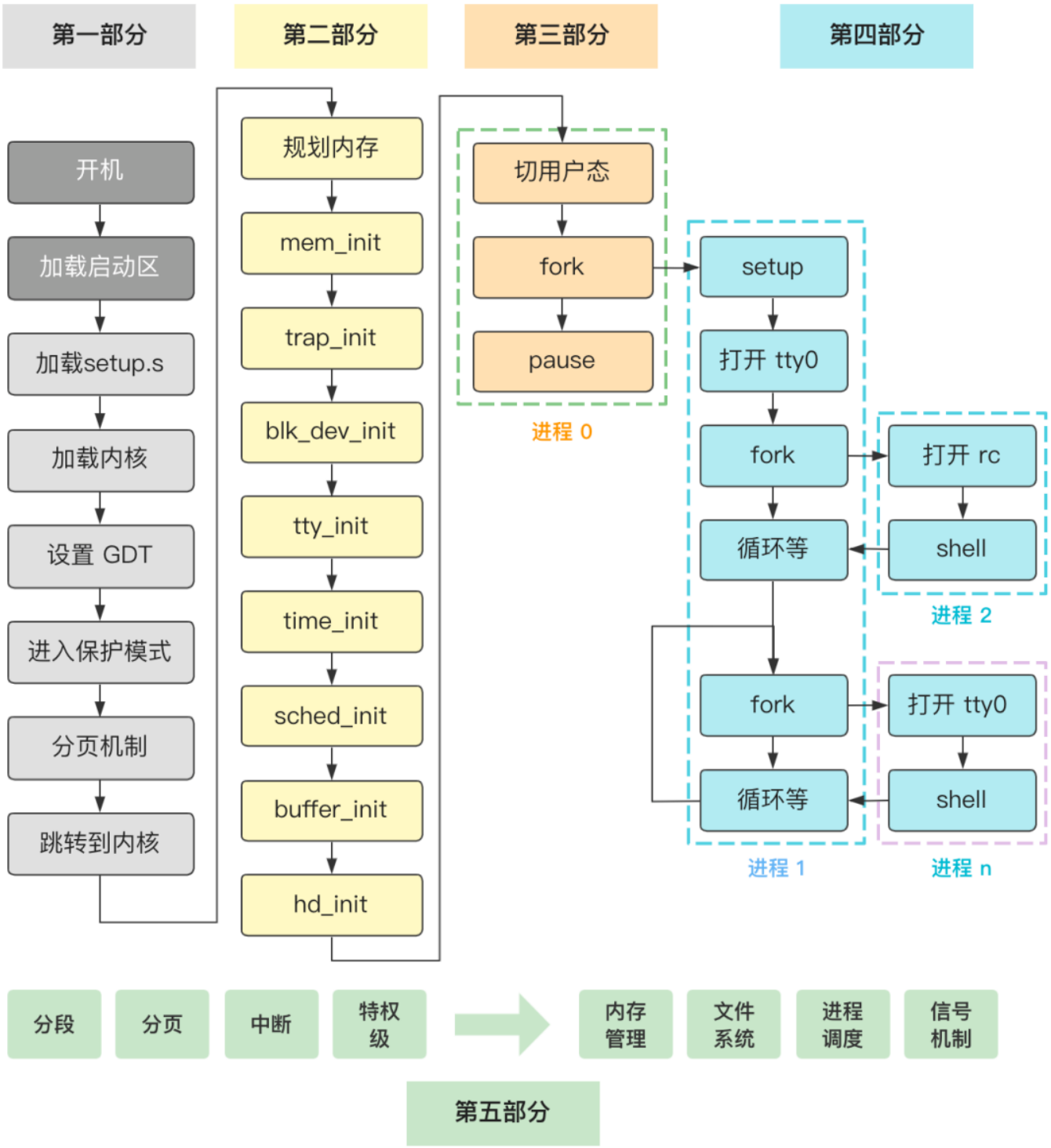
收录于合集
#操作系统源码

43个

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本回的内容属于第四部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一部分 进入内核前的苦力活

第二部分 大战前期的初始化工作

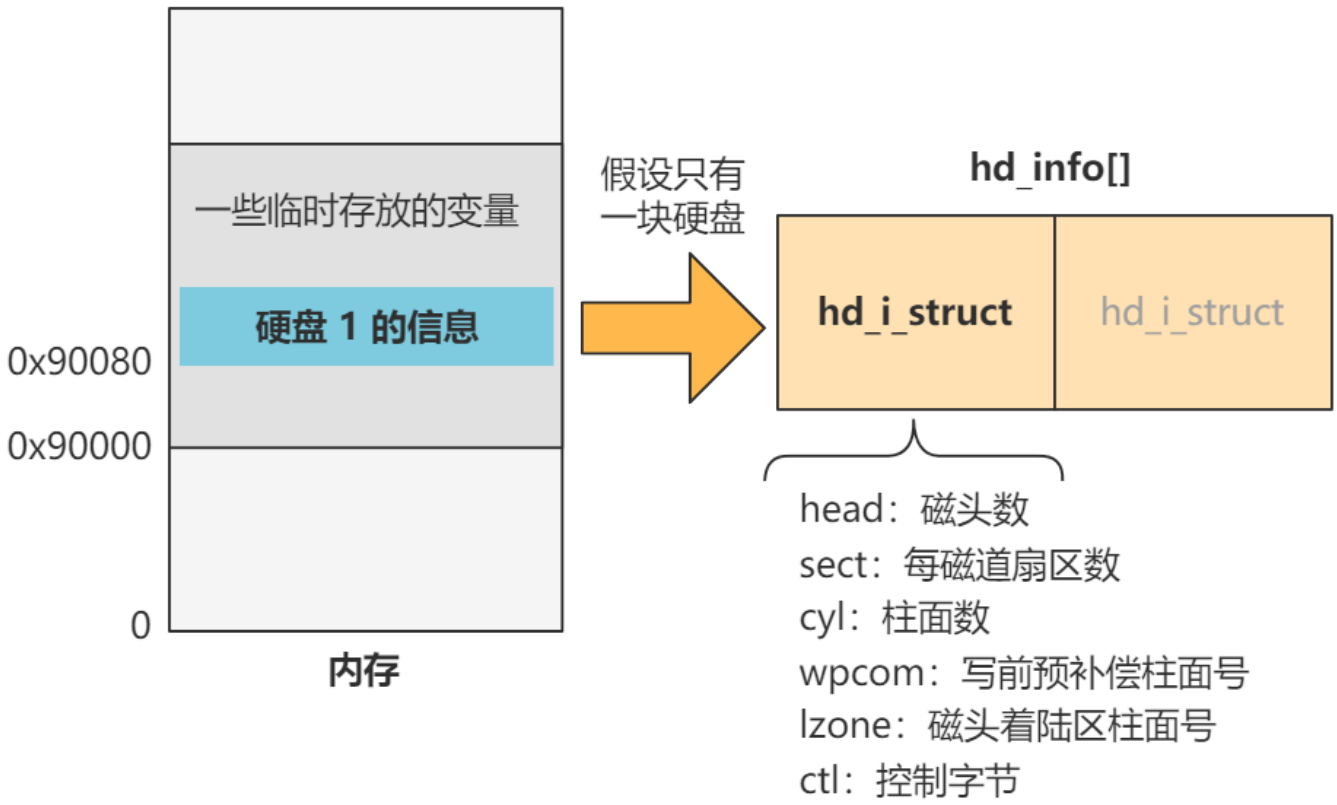
第三部分：一个新进程的诞生

第四部分：shell 程序的到来

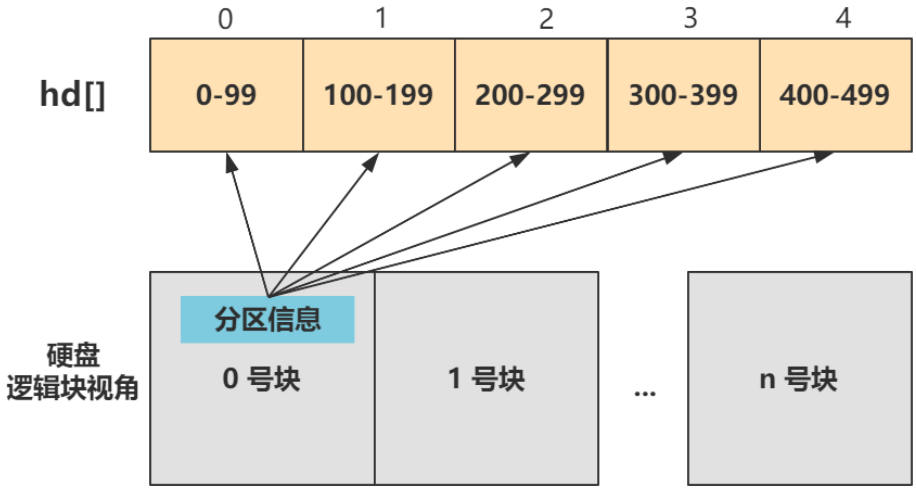
本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末阅读原文可直接跳转）
https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

书接上回，上回书咱们说到，我们已经把硬盘的基本信息存入了 hd_info[]。



把硬盘的分区信息存入了 hd[]。



并且留了个读取硬盘数据的 bread 函数没有讲，等主流程讲完再展开这些函数的细节，我知道这是你们关心的内容。

这些都是 setup 方法里做的事情，也就是进程 0 fork 出的进程 1 所执行的第一个方法。

今天我们说 setup 方法中的最后一个函数 **mount_root**。

```
int sys_setup(void * BIOS) {
    ...
    mount_root();
}
```

mount_root 直译过来就是**加载根**。

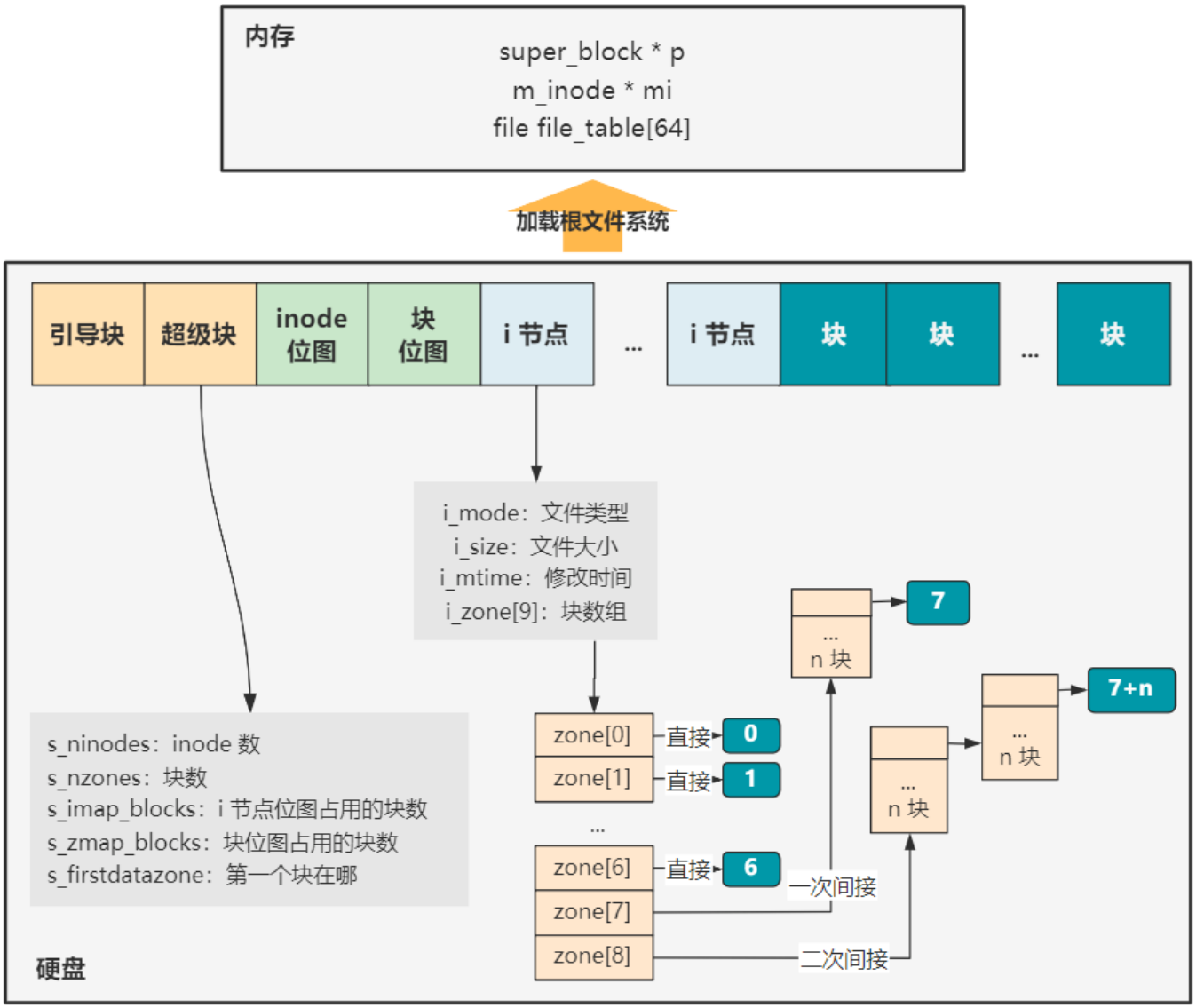
再多说几个字是**加载根文件系统**，有了它之后，操作系统才能从一个**根儿**开始找到所有存储在硬盘中的文件，所以它是文件系统的基石，很重要。

我们翻开看看。

```
void mount_root(void) {
    int i,free;
    struct super_block * p;
    struct m_inode * mi;

    for(i=0;i<64;i++)
        file_table[i].f_count=0;

    for(p = &super_block[0] ; p < &super_block[8] ; p++) {
        p->s_dev = 0;
        p->s_lock = 0;
        p->s_wait = NULL;
    }
    p=read_super(0);
    mi=iget(0,1);

    mi->i_count += 3 ;
    p->s_isup = p->s_imount = mi;
    current->pwd = mi;
    current->root = mi;
    free=0;
    i=p->s_nzones;
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
            free++;

    free=0;
    i=p->s_ninodes+1;
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
            free++;
}
```

很简单。

从整体上说，它就是要把硬盘中的数据，以文件系统的格式进行解读，加载到内存中设计好的数据结构，这样操作系统就可以通过内存中的数据，以文件系统的方式访问硬盘中的一个个文件了。



那其实搞清楚两个事情即可：

第一，硬盘中的文件系统格式是怎样的？

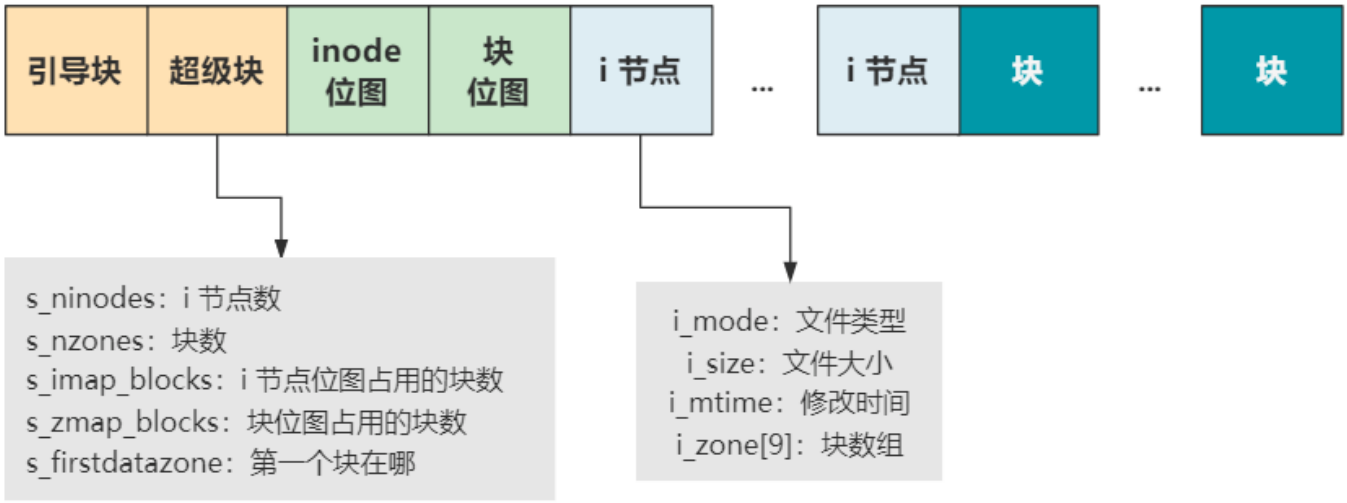
第二，内存中用于文件系统的数据结构有哪些？

我们一个个来。

硬盘中的文件系统格式是怎样的

首先硬盘中的文件系统，无非就是硬盘中的一堆数据，我们按照一定格式去解析罢了。Linux-

0.11 中的文件系统是 **MINIX** 文件系统，它就长成这个样子。



每一个块结构的大小是 1024 字节，也就是 1KB，硬盘里的数据就按照这个结构，妥善地安排在硬盘里。

可是硬盘中凭什么就有了这些信息呢？这就是个鸡生蛋蛋生鸡的问题了。你可以先写一个操作系统，然后给一个硬盘做某种文件系统类型的格式化，这样你就得到一个有文件系统的硬盘了，有了这个硬盘，你的操作系统就可以成功启动了。

总之，想个办法给这个硬盘写上数据呗。

好了，现在我们简单看看 MINIX 文件系统的格式。

引导块就是我们系列最开头说的启动区，当然不一定所有的硬盘都有启动区，但我们还是得预留出这个位置，以保持格式的统一。

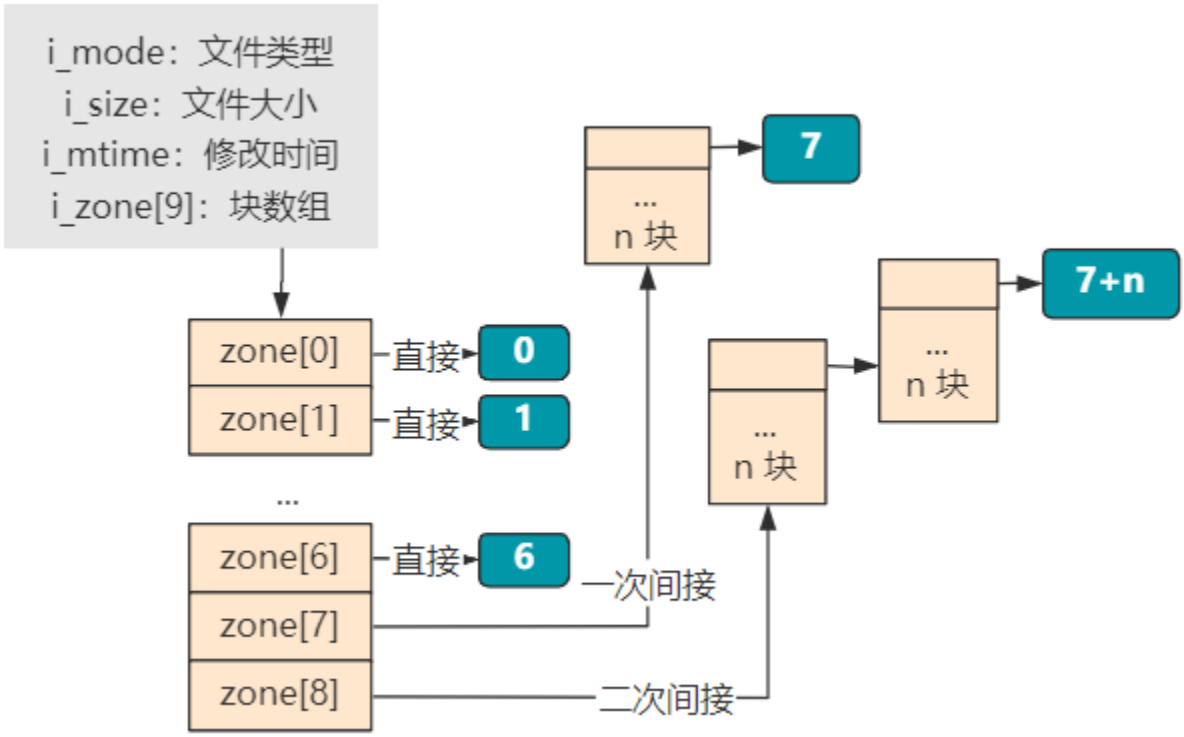
超级块用于描述整个文件系统的整体信息，我们看它的字段就知道了，有后面的 inode 数量，块数量，第一个块在哪里等信息。有了它，整个硬盘的布局就清晰了。

inode 位图和块位图，就是位图的基本操作和作用了，表示后面 inode 和块的使用情况，和我们之前讲的内存占用位图 mem_map[] 是类似的。

再往后，**inode** 存放着每个文件或目录的元信息和索引信息，元信息就是文件类型、文件大小、修改时间等，索引信息就是大小为 9 的 i_zone[9] 块数组，表示这个文件或目录的具体

数据占用了哪些块。

其中块数组里，0~6 表示直接索引，7 表示一次间接索引，8 表示二次间接索引。当文件比较小时，比如只占用 2 个块就够了，那就只需要 zone[0] 和 zone[1] 两个直接索引即可。



再往后，就都是存放具体文件或目录实际信息的**块**了。如果是一个普通文件类型的 inode 指向的块，那里面就直接是文件的二进制信息。如果是一个目录类型的 inode 指向的块，那里面存放的就是这个目录下的文件和目录的 inode 索引以及文件或目录名称等信息。

好了，文件系统格式的说明，我们就简单说明完毕了，MINIX 文件系统已经过时，你可以阅读我之前写的 图解 | 你管这破玩意叫文件系统？ 来全面了解一个 ext2 文件系统的来龙去脉，基本思想都是一样的。

内存中用于文件系统的数据结构有哪些

赶紧回过头来看我们的代码，是如何加载以这样一种格式存放在硬盘里的数据，以被我们操作系统所管控的。

从头看。

```
struct file {
    unsigned short f_mode;
    unsigned short f_flags;
    unsigned short f_count;
    struct m_inode * f_inode;
    off_t f_pos;
};

void mount_root(void) {
    for(i=0;i<64;i++)
        file_table[i].f_count=0;
    ...
}
```

把 64 个 **file_table** 里的 f_count 清零。

这个 file_table 表示进程所使用的文件，进程每使用一个文件，都需要记录在这里，包括文件类型、文件 inode 索引信息等，而这个 f_count 表示被引用的次数，此时还没有引用，所以设置为零。

而这个 file_table 的索引（当然准确说是进程的filp索引才是），就是我们通常说的文件描述符。比如有如下命令。

```
echo "hello" > 0
```

就表示把 hello 输出到 0 号文件描述符。

0 号文件描述符是哪个文件呢？就是 file_table[0] 所表示的文件。

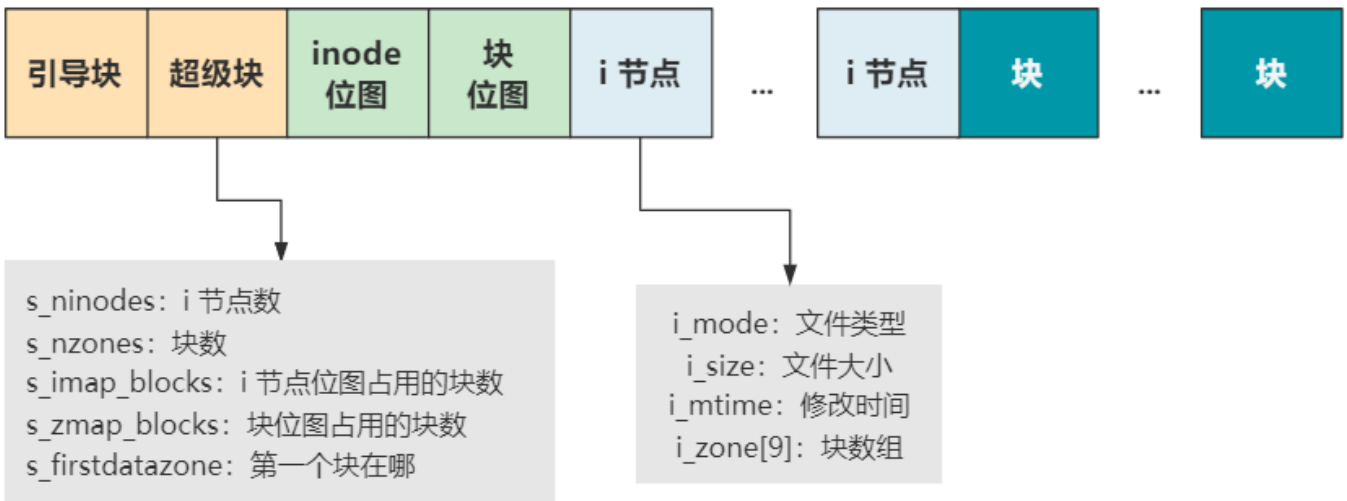
这个文件在哪里呢？注意到 file 结构里有个 f_inode 字段，通过 f_inode 即可找到它的 inode 信息，inode 信息包含了一个文件所需要的全部信息，包括文件的大小、文件的类型、文件所在的硬盘块号，这个所在硬盘块号，就是文件的位置咯。

接着看。

```
struct super_block super_block[8];
void mount_root(void) {
    ...
    struct super_block * p;
    for(p = &super_block[0] ; p < &super_block[8] ; p++) {
        p->s_dev = 0;
        p->s_lock = 0;
        p->s_wait = NULL;
    }
    ...
}
```

又是把一个数组 **super_block** 做清零工作。

这个 super_block 存在的意义是，操作系统与一个设备以文件形式进行读写访问时，就需要把这个设备的超级块信息放在这里。



这样通过这个超级块，就可以掌控这个设备的文件系统全局了。

果然，接下来的操作，就是读取硬盘的超级块信息到内存中来。

```
void mount_root(void) {
    ...
    p=read_super(0);
    ...
}
```

read_super 就是读取硬盘中的超级块。

接下来，读取根 inode 信息。

```
struct m_inode * mi;
void mount_root(void) {
    ...
    mi=iget(0,1);
    ...
}
```

然后把该 inode 设置为当前进程（也就是进程 1）的当前工作目录和根目录。

```
void mount_root(void) {
    ...
    current->pwd = mi;
    current->root = mi;
    ...
}
```

然后记录块位图信息。

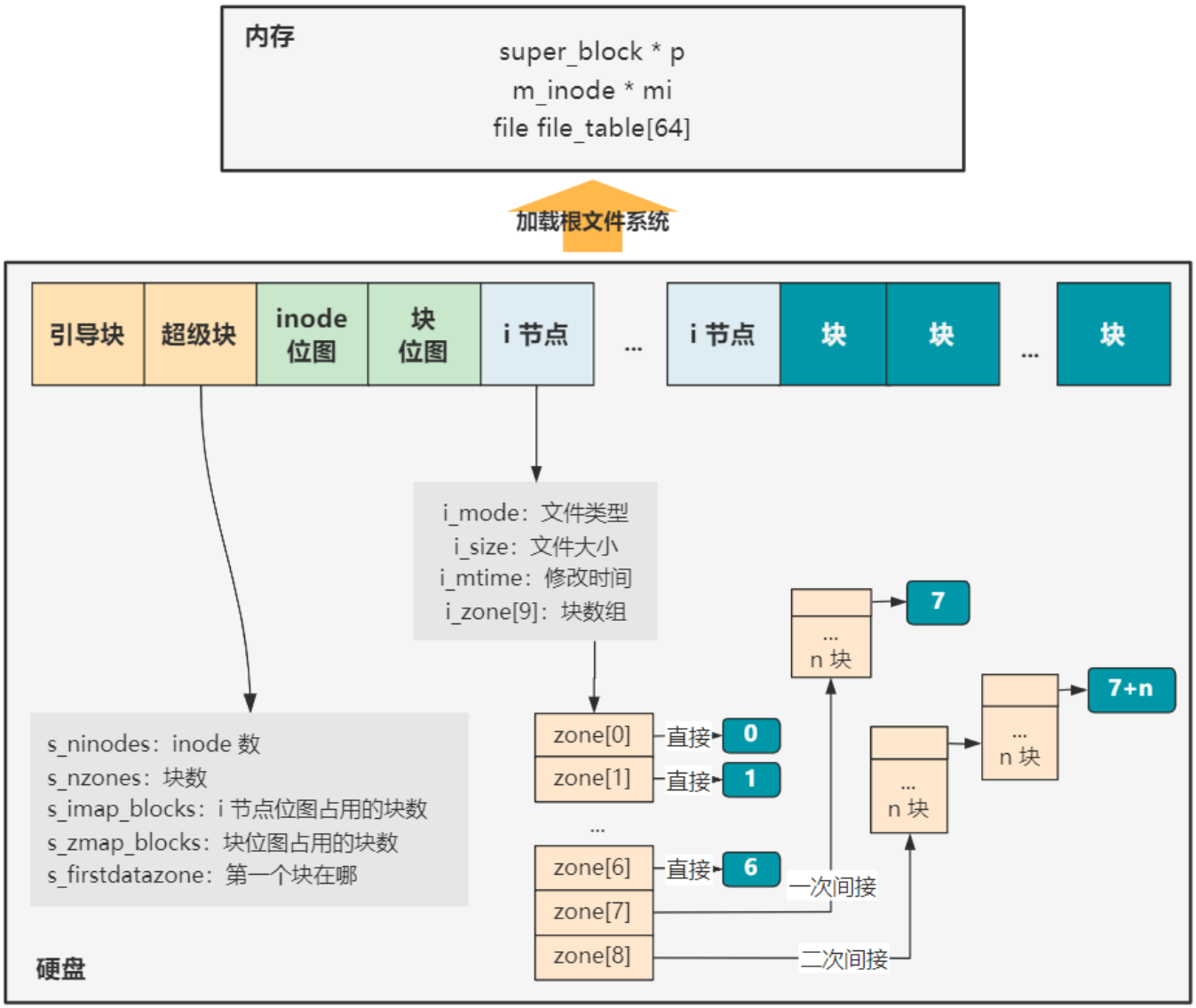
```
void mount_root(void) {
    ...
    i=p->s_nzones;
    while (-- i >= 0)
        set_bit(i&8191, p->s_zmap[i>>13]->b_data);
    ...
}
```

最后记录 inode 位图信息。

```
void mount_root(void) {
    ...
    i=p->s_ninodes+1;
    while (-- i >= 0)
        set_bit(i&8191, p->s_imap[i>>13]->b_data);
}
```

就完事了。

其实整体上就是把硬盘中文件系统的各个信息，搬到内存中。之前的图可以说非常直观了。



有了内存中的这些结构，我们就可以顺着根 inode，找到所有的文件了。

至此，加载根文件系统的 **mount_root** 函数就全部结束了。同时，让我们回到全局视野，发现 **setup** 函数也一并结束了。

```
void main(void) {
    ...
    move_to_user_mode();
    if (!fork()) {
        init();
    }
    for(;;) pause();
}

void init(void) {
    setup((void *) &drive_info);
    ...
}

int sys_setup(void * BIOS) {
    ...
    mount_root();
}
```

setup 的主要工作就是我们今天所讲的，**加载根文件系统**。

我们继续往下看 init 函数。

```
void init(void) {
    setup((void *) &drive_info);
    (void) open("/dev/tty0",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
}
```

看到这相信你也明白了。

之前 setup 函数的一番折腾，加载了根文件系统，顺着根 inode 可以找到所有文件，就是为了下一行 open 函数可以通过文件路径，从硬盘中把一个文件的信息方便地拿到。

在这里，我们 open 了一个 **/dev/tty0** 的文件，那我们接下来的焦点就在这个 /dev/tty0 是

个啥？

欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这，开篇词

本系列的番外故事看这，让我们一起来写本书？也可以直接无脑加入星球，共同参与这场旅行。

最后，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 GitHub 项目点个 star，就在**阅读原文**处，这些就足够让我坚持写下去了！我们下回见。

低并发编程
战略上藐视技术，战术上重视技术
175篇原创内容

Official Account

收录于合集 #操作系统源码 43

上一篇
第31回 | 拿到硬盘信息

下一篇
第33回 | 打开终端设备文件