第31回 | 拿到硬盘信息

Original 闪客 低并发编程 2022-03-27 17:30

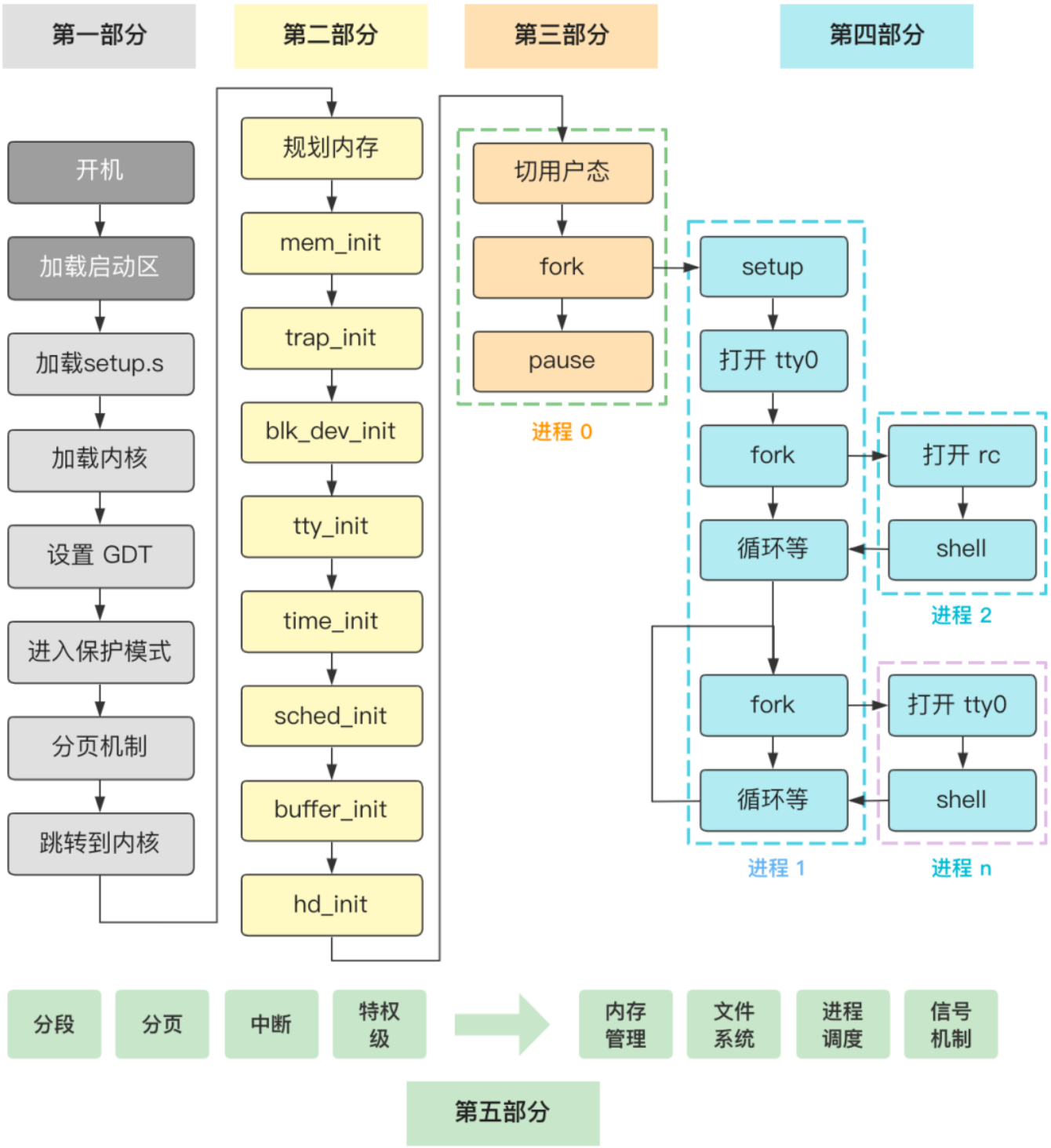
收录于合集
#操作系统源码

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（文末阅读原文可直接跳转）
https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

上一个大部分的名字叫一个新进程的诞生，讲述了进程 0 调用了 fork 函数创建了一个新的进程 —— 进程 1，并且使其达到了可以被调度的状态，fork 就算正式完成了自己的使命。

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

由于 fork 函数一调用，就又多出了一个进程，子进程（进程 1）会返回 0，父进程（进程 0）返回子进程的 ID，所以 **init** 函数只有进程 1 才会执行。

第三部分结束后，就到了现在的第四部分，**shell 程序的到来**。而整个第四部分的故事，就是这个 init 函数做的事情。

虽然就一行代码，但这里的事情可多了去了，我们先看一下整体结构。我已经把单纯的日志打印和错误校验逻辑去掉了。

```
void init(void) {
    int pid,i;
    setup((void *) &drive_info);
    (void) open("/dev/tty0",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
    if (!(pid=fork())) {
        open("/etc/rc",O_RDONLY,0);
        execve("/bin/sh",argv_rc,envp_rc);
    }
    if (pid>0)
        while (pid != wait(&i))
            /* nothing */;
    while (1) {
        if (!pid=fork()) {
            close(0);close(1);close(2);
            setsid();
            (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);
            (void) dup(0);
            _exit(execve("/bin/sh",argv,envp));
        }
        while (1)
            if (pid == wait(&i))
                break;
        sync();
    }
    _exit(0);   /* NOTE! _exit, not exit() */
}
```

是不是看着还挺复杂?

不过别急，今天我们就只讲第一行代码 **setup** 的一部分，硬盘信息的获取。

```
struct drive_info { char dummy[32]; } drive_info;

// drive_info = (*(struct drive_info *)0x90080);

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

先看入参。

drive_info 是来自内存 0x90080 的数据，这部分是由之前 第5回 | 进入保护模式前的最后一次折腾内存 讲的 setup.s 程序将硬盘 1 的参数信息放在这里了，包括柱面数、磁头数、扇区数等信息。

setup 是个系统调用，会通过中断最终调用到 sys_setup 函数。关于系统调用的原理，在 第25回 | 通过 fork 看一次系统调用 中已经讲得很清楚了，此处不再赘述。

所以直接看 sys_setup 函数，我仍然是对代码做了少许的简化，去掉了日志打印和错误判断分支，并且仅当作只有一块硬盘，去掉了一层 for 循环。

```
int sys_setup(void * BIOS) {

    hd_info[0].cyl = *(unsigned short *) BIOS;
    hd_info[0].head = *(unsigned char *) (2+BIOS);
    hd_info[0].wpcom = *(unsigned short *) (5+BIOS);
    hd_info[0].ctl = *(unsigned char *) (8+BIOS);
    hd_info[0].lzone = *(unsigned short *) (12+BIOS);
    hd_info[0].sect = *(unsigned char *) (14+BIOS);
    BIOS += 16;

    hd[0].start_sect = 0;
    hd[0].nr_sects =
        hd_info[0].head * hd_info[0].sect * hd_info[0].cyl;

    struct buffer_head *bh = bread(0x300, 0);
    struct partition *p = 0x1BE + (void *)bh->b_data;
    for (int i=1;i<5;i++,p++) {
        hd[i].start_sect = p->start_sect;
        hd[i].nr_sects = p->nr_sects;
    }
    brelse(bh);

    rd_load();
    mount_root();
    return (0);
}
```

好，我们一点点看。

先看第一部分，硬盘基本信息的赋值的操作。

```
int sys_setup(void * BIOS) {
    hd_info[0].cyl = *(unsigned short *) BIOS;
    hd_info[0].head = *(unsigned char *) (2+BIOS);
    hd_info[0].wpcom = *(unsigned short *) (5+BIOS);
    hd_info[0].ctl = *(unsigned char *) (8+BIOS);
    hd_info[0].lzone = *(unsigned short *) (12+BIOS);
    hd_info[0].sect = *(unsigned char *) (14+BIOS);
    BIOS += 16;
    ...
}
```

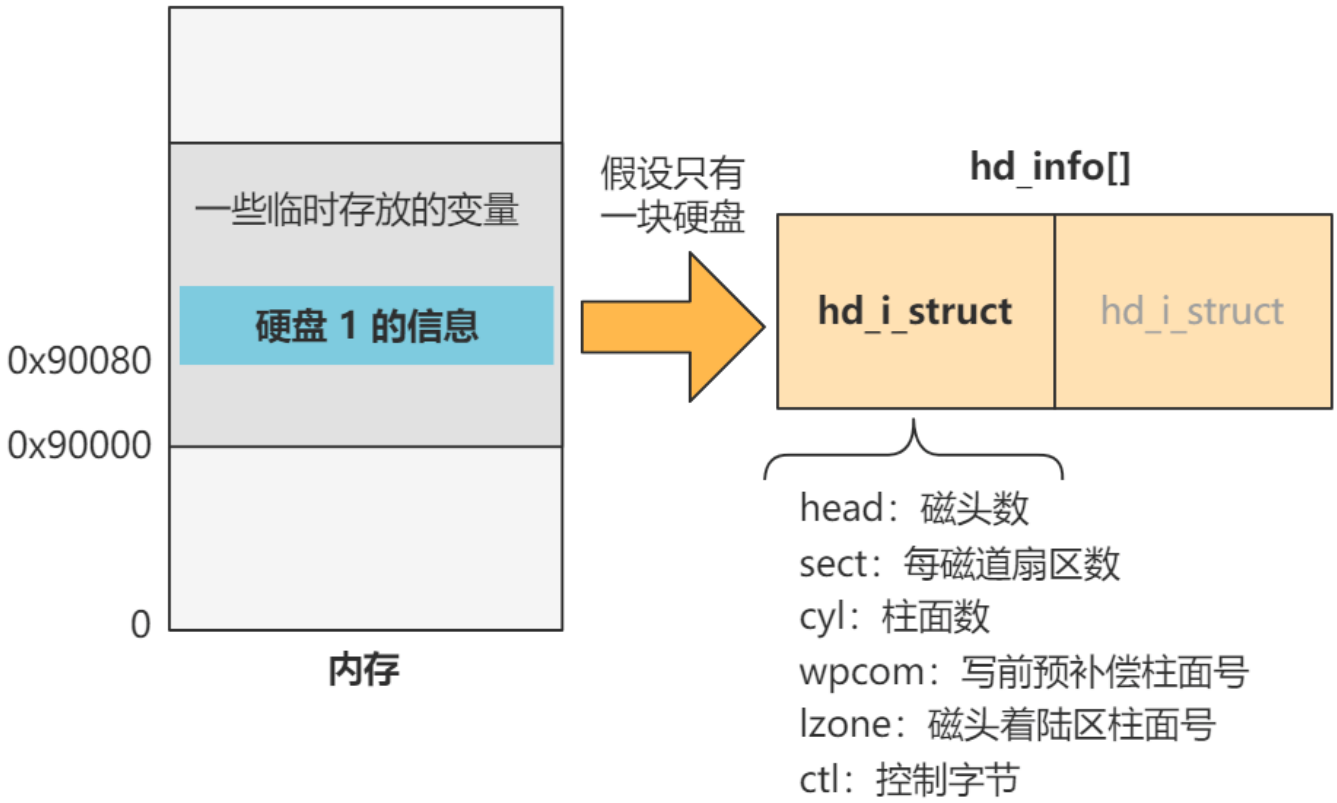
刚刚说了，入参 BIOS 是来自内存 0x90080 的数据，这部分是由之前 第5回 | 进入保护模式前的最后一次折腾内存 讲的 setup.s 程序将硬盘 1 的参数信息放在这里了，包括柱面数、磁头数、扇区数等信息。

所以，一开始先往 **hd_info** 数组的 0 索引处存上这些信息。我们假设就只有一块硬盘，所以这个数组也只有一个元素。

这个数组里的结构就是 **hd_i_struct**，就表示硬盘的参数。

```
struct hd_i_struct {
    // 磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆区柱面号、控制字节
    int head,sect,cyl,wpcom,lzone,ctl;
};
struct hd_i_struct hd_info[] = {};
```

最终效果就是这样。



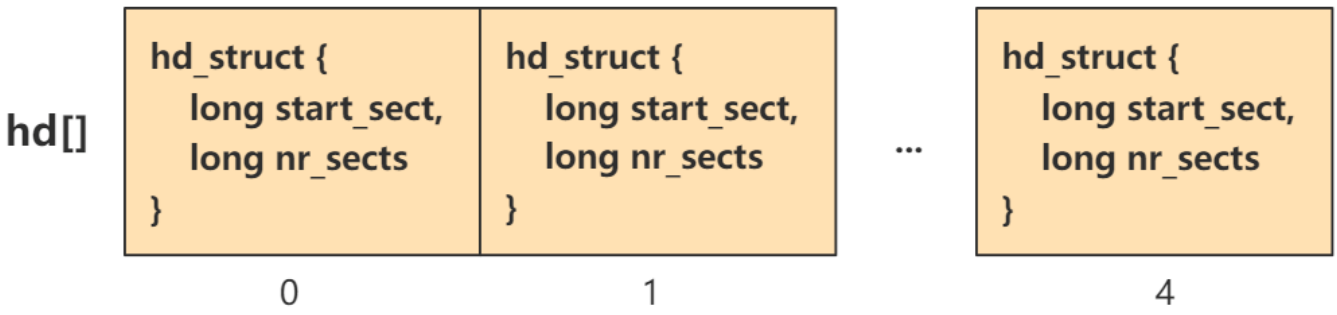
OK，我们继续。

看第二部分，硬盘分区表的设置。

```
static struct hd_struct {
    long start_sect;
    long nr_sects;
} hd[5] = {}

int sys_setup(void * BIOS) {
    ...
    hd[0].start_sect = 0;
    hd[0].nr_sects =
        hd_info[0].head * hd_info[0].sect * hd_info[0].cyl;
    struct buffer_head *bh = bread(0x300, 0);
    struct partition *p = 0x1BE + (void *)bh->b_data;
    for (int i=1;i<5;i++,p++) {
        hd[i].start_sect = p->start_sect;
        hd[i].nr_sects = p->nr_sects;
    }
    brelse(bh);
    ...
}
```

只看最终效果，就是给 hd 数组的五项附上了值。



这表示硬盘的分区信息，每个分区用 **start_sect** 和 **nr_sects**，也就是开始扇区和总扇区数来记录。

这些信息是从哪里获取的呢？就是在硬盘的第一个扇区的 0x1BE 偏移处，这里存储着该硬盘的分区信息，只要把这个地方的数据拿到就 OK 了。

所以 bread 就是干这事的，从硬盘读取数据。

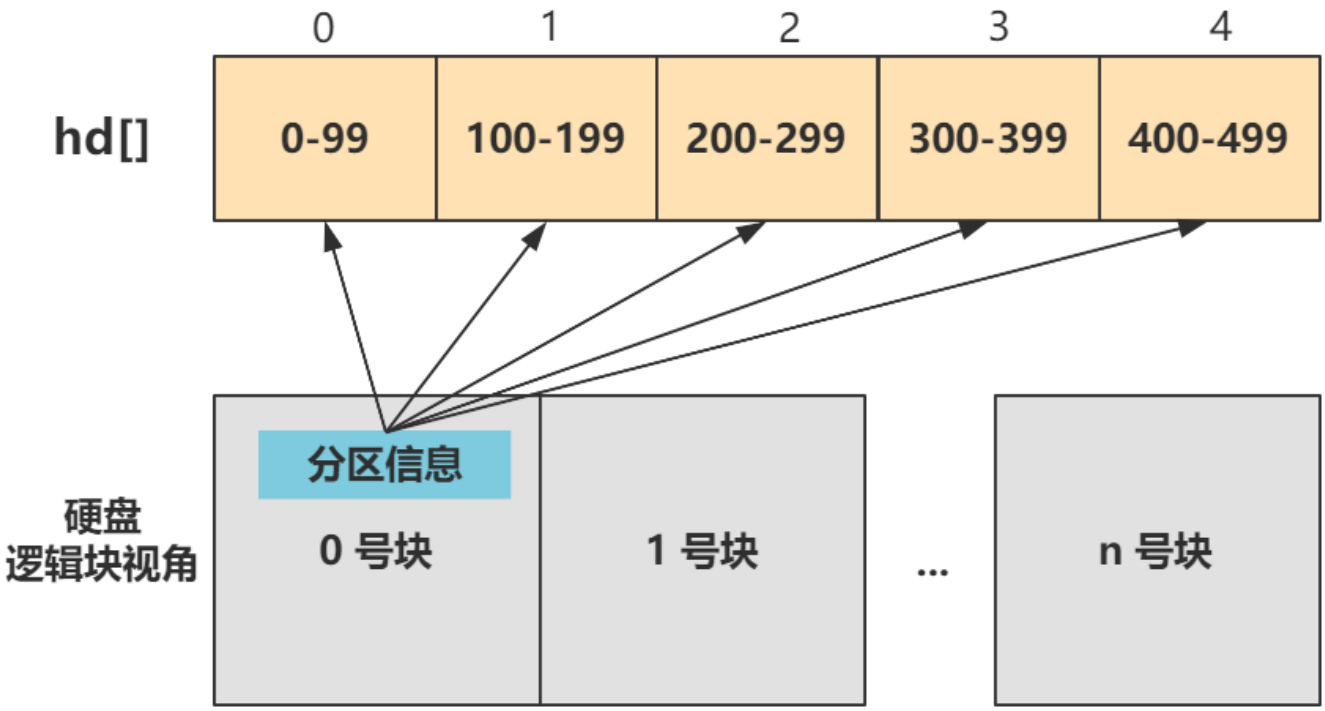
```
struct buffer_head *bh = bread(0x300, 0);
```

第一个参数 0x300 是第一块硬盘的主设备号，就表示要读取的块设备是硬盘一。第二个参数 0 表示读取第一个块，一个块为 1024 字节大小，也就是连续读取硬盘开始处 0 ~ 1024 字节的数据。

拿到这部分数据后，再取 0x1BE 偏移处，就得到了分区信息。

```
struct partition *p = 0x1BE + (void *)bh->b_data;
```

就这么点事。



至于如何从硬盘中读取指定位置（块）的数据，也就是 **bread** 函数的内部实现，那是相当复杂的，涉及到与缓冲区配合的部分，还有读写请求队列的设置，以及中断。

当然，这个函数就是经典的问题，**从硬盘中读取数据的原理**，但这些都不影响主流程，因为仅仅是把硬盘某位置的数据读到内存而已，先不去深入细节，细节部分将在第五部分展开说明。

OK，目前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[]，我们继续往下看。

```
int sys_setup(void * BIOS) {
    ...
    rd_load();
    mount_root();
    return (0);
}
```

就剩两个函数了。

其中 **rd_load** 是当有 ramdisk 时，也就是虚拟内存盘，才会执行。虚拟内存盘是通过软件将一部分内存（RAM）模拟为硬盘来使用的一种技术，一种小玩法而已，我们就先当做没有，否则很影响看主流程的心情。

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。

为了加载根文件系统，或者说所谓的加载根文件系统，就是把硬盘中的数据加载到内存里，以文件系统的数据格式来解读这些信息。

所以第一，需要硬盘本身就有文件系统的信息，硬盘不能是裸盘，这个不归操作系统管，你为了启动我的 Linux 0.11，必须拿来一块做好了文件系统的硬盘来。

第二，需要读取硬盘的数据到内存，那就必须需要知道硬盘的参数信息，这就是我们本讲所做的事情的意义。

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。

收录于合集 #操作系统源码 43

上一篇
写时复制就这么几行代码，麻烦你先看看再BB 行吗?

下一篇
第32回 | 加载根文件系统

一个逻辑完备的线程池

程序喵大人

模拟网站攻击到提权的全部过程

编码安全研究

理解Linux CPU上下文切换

码农的荒岛求生