第19回 | 操作系统就是用这两个面试常考的结构管理的缓冲区

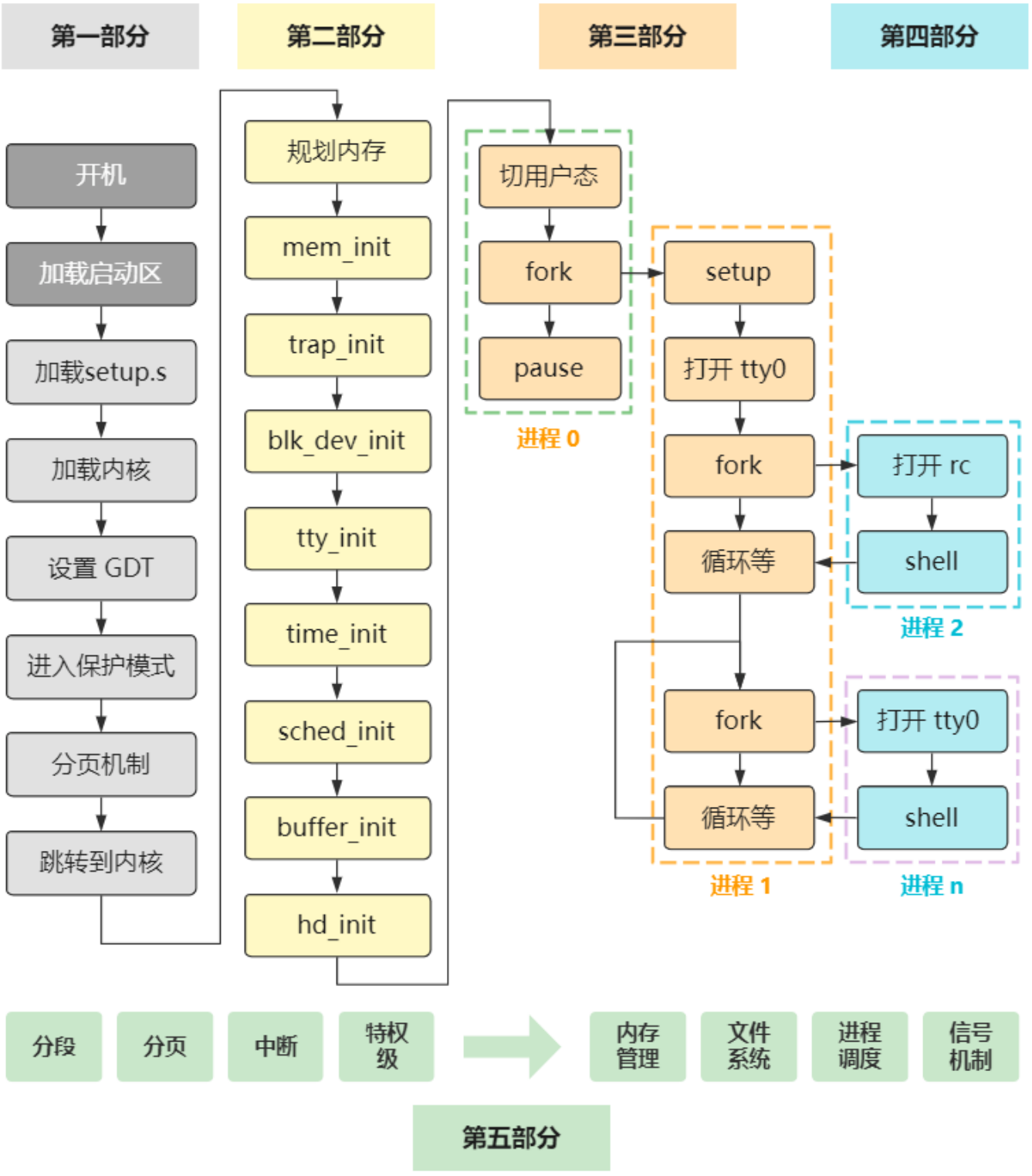
Original 闪客 低并发编程 2022-01-19 16:30

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。



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

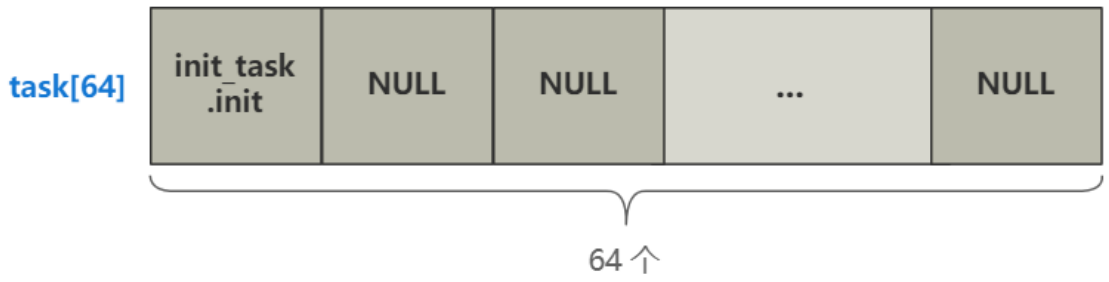
第一部分 进入内核前的苦力活

第二部分 大战前期的初始化工作

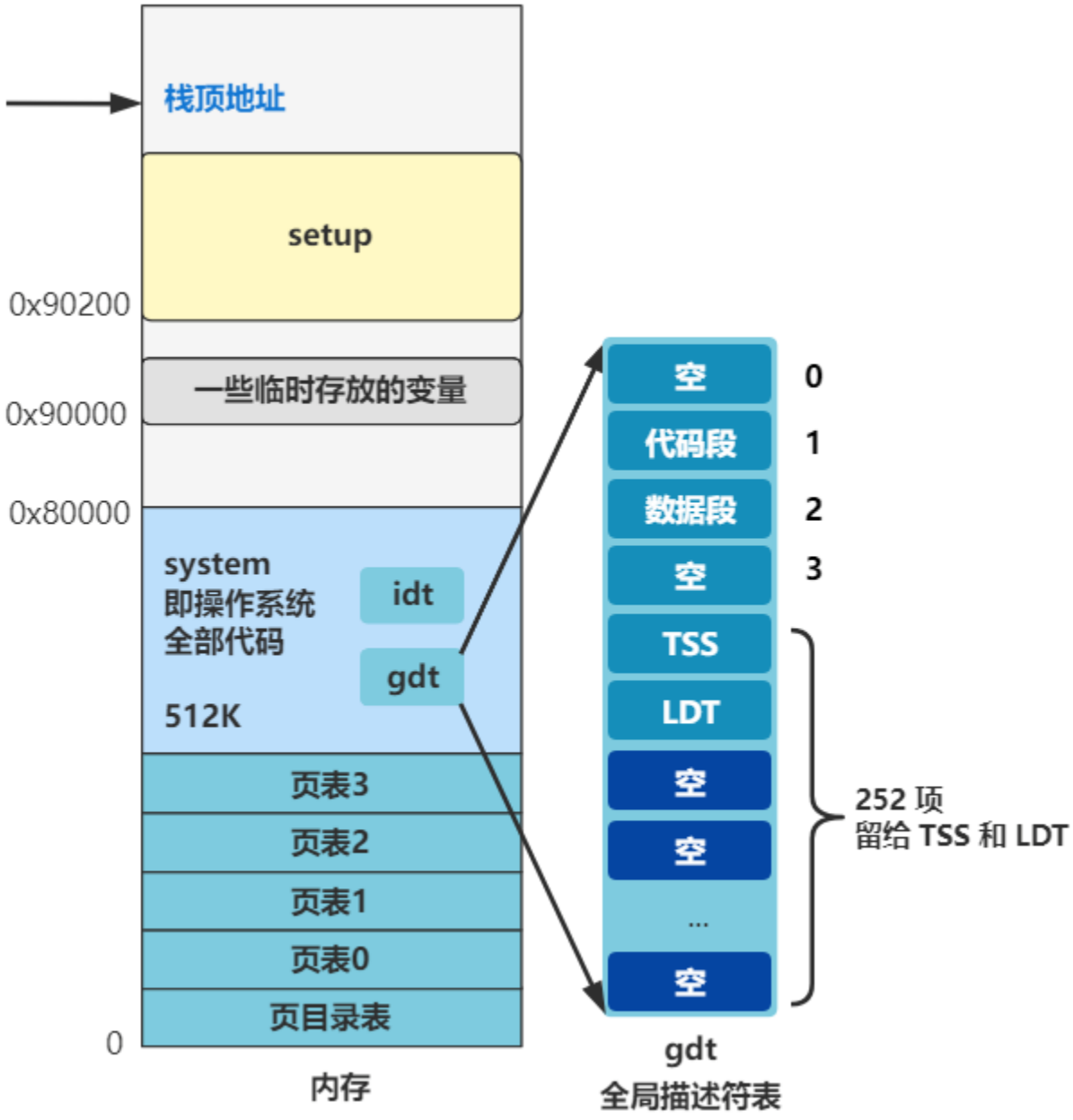
本系列的 GitHub 地址如下（文末阅读原文可直接跳转）
https://github.com/sunym1993/flash-linux0.11-talk

------- 正文开始 -------

书接上回，上回书我们说到了进程调度的初始化，定义了一个长度为 64 的 task 数组用于管理全部进程的结构。



之后在 GDT 中预定义了进程调度需要用到的 TSS 和 LDT 结构。



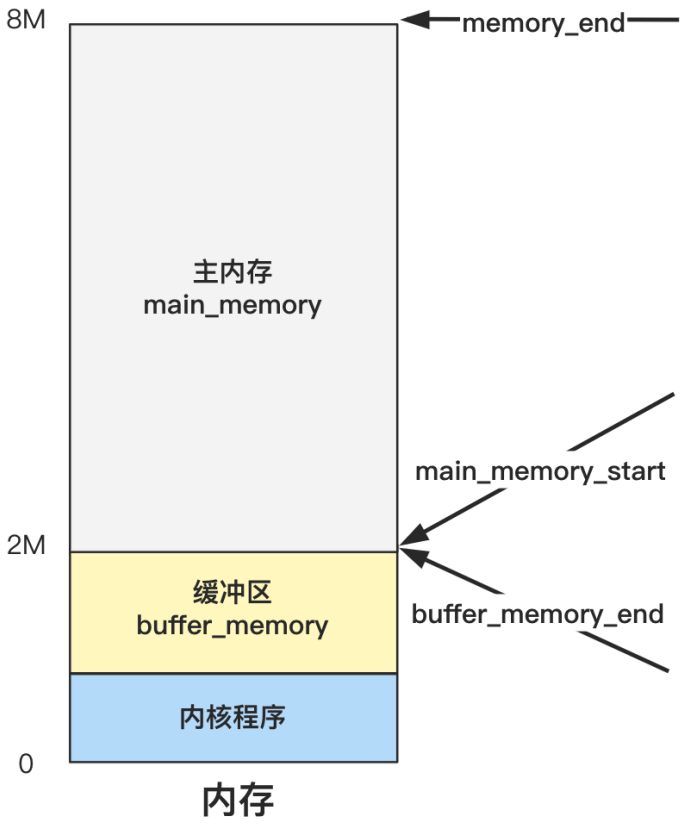
之后开启了定时器，准备迎接时钟中断的到来，进而触发进程调度。

那接下来我们就冷静下，回到 main 函数，继续看下一个初始化的过程。

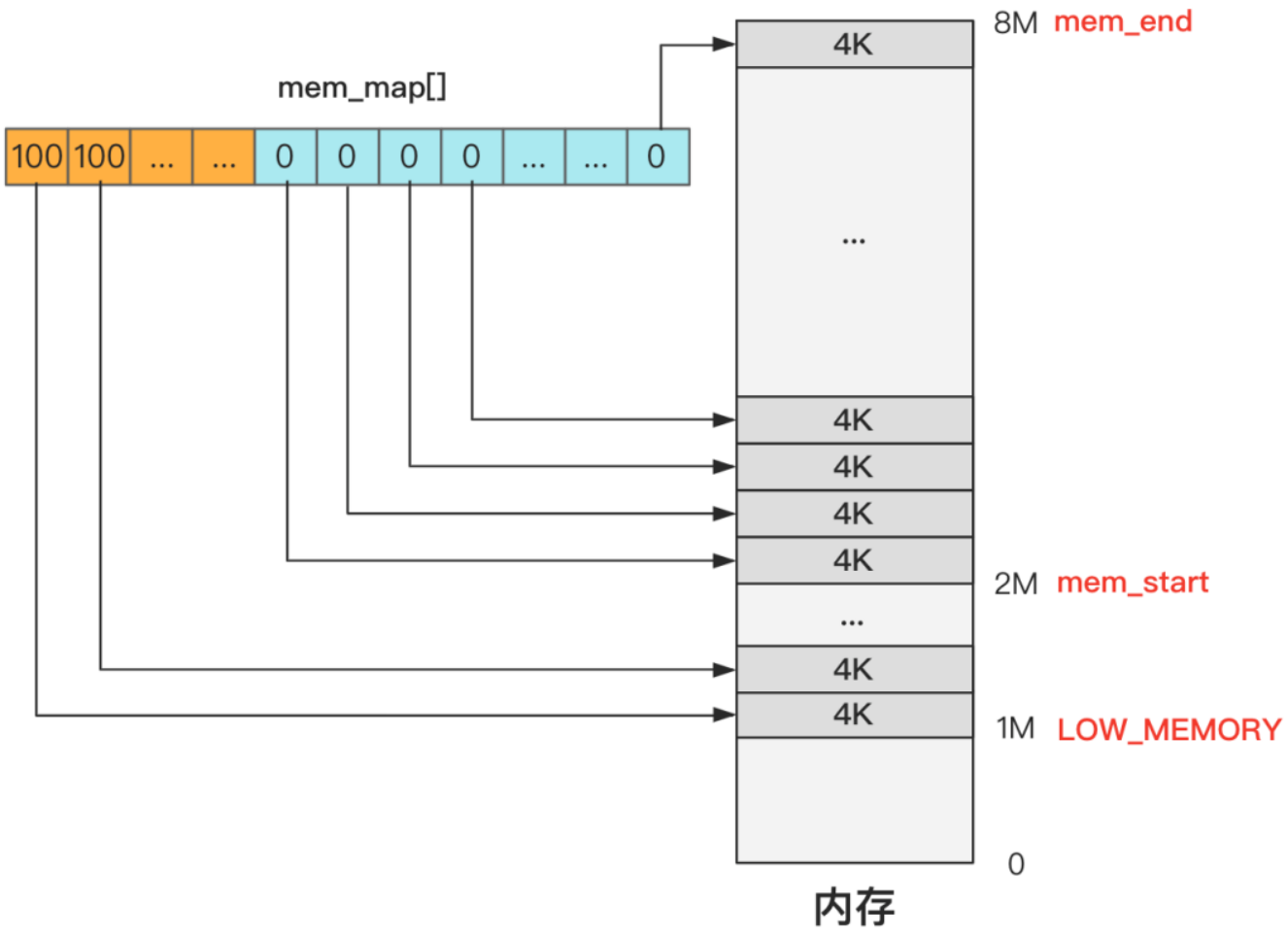
那就是缓冲区初始化 buffer_init，加油，没剩多少了！

```
void main(void) {
    ...
    mem_init(main_memory_start,memory_end);
    trap_init();
    blk_dev_init();
    chr_dev_init();
    tty_init();
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    hd_init();
    floppy_init();
    sti();
    move_to_user_mode();
    if (!fork()) {
        init();
    }
    for(;;) pause();
}
```

首先要注意到，这个函数传了个参数 buffer_memory_end，这个是在老早之前就设置好的，就在第12回 | 管理内存前先划分出三个边界值，回顾下。



想起来了吧？而且我们在 第13回 | 主内存初始化 mem_init 中，用 mem_init 设置好了主内存的管理结构 mam_map。



再想不起来那就需要把前面的章节再读一读咯，不然后面越来越难。

前面是把主内存区管理起来了，所以今天就是把剩下的缓冲区部分，也初始化管理起来。目的就是这么单纯，我们看代码。

我们还是采用之前的方式，就假设内存只有 8M，把一些不相干的分支去掉，方便理解。

```
extern int end;
struct buffer_head * start_buffer = (struct buffer_head *) &end;

void buffer_init(long buffer_end) {
    struct buffer_head * h = start_buffer;
    void * b = (void *) buffer_end;
    while ( (b -= 1024) >= ((void *) (h+1)) ) {
        h->b_dev = 0;
        h->b_dirt = 0;
        h->b_count = 0;
        h->b_lock = 0;
        h->b_uptodate = 0;
        h->b_wait = NULL;
        h->b_next = NULL;
        h->b_prev = NULL;
        h->b_data = (char *) b;
        h->b_prev_free = h-1;
        h->b_next_free = h+1;
        h++;
    }
    h--;
    free_list = start_buffer;
    free_list->b_prev_free = h;
    h->b_next_free = free_list;
    for (int i=0;i<307;i++)
        hash_table[i]=NULL;
}
```

虽然很长，但其实就造了**两个数据结构**而已。

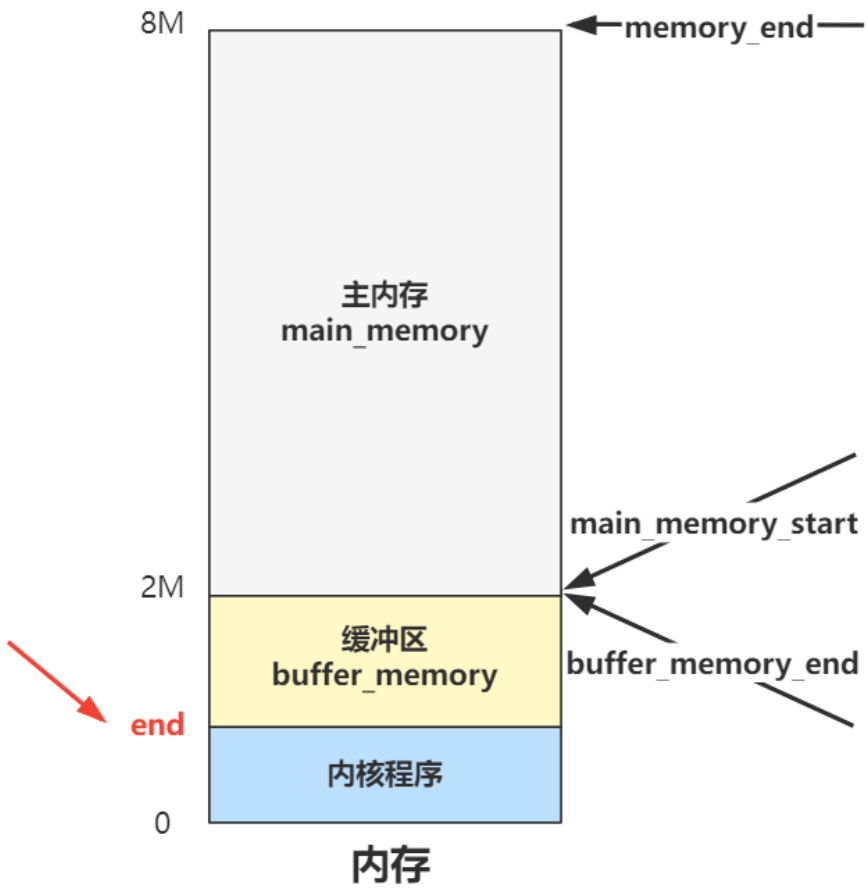
不过别急，我们先看这一行代码。

```
extern int end;
void buffer_init(long buffer_end) {
    struct buffer_head * start_buffer = (struct buffer_head *) &end;
    ...
}
```

这里有个外部变量 **end**，而我们的缓冲区开始位置 **start_buffer** 就等于这个变量的内存地址。

这个外部变量 end 并不是操作系统代码写就的，而是由**链接器 ld** 在链接整个程序时设置的一个外部变量，帮我们计算好了整个内核代码的末尾地址。

那在这之前的是内核代码区域肯定不能用，在这之后的，就给 buffer 用了。所以我们的内存分布图可以更精确一点了。



你看，之前的疑惑解决了吧？很好理解嘛，内核程序和缓冲区的划分，肯定有个分界线，这个分界线就是 end 变量的值。

这个值定多少合适呢?

像主内存和缓冲区的分界线，就直接代码里写死了，就是上图中的 2M。

可是内核程序占多大内存在写的时候完全不知道，就算知道了如果改动一点代码也会变化，所以就由程序编译链接时由链接器程序帮我们把这个内核代码末端的地址计算出来，作为一个外部变量 end 我们拿来即用，就方便多了。

好，回过头我们再看看，**整段代码创造了哪两个管理结构**?

我们先看这段结构。

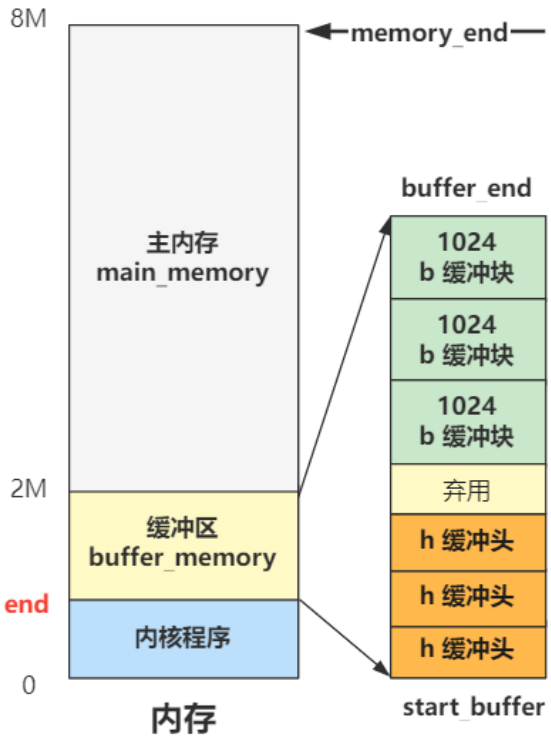
```
void buffer_init(long buffer_end) {
    struct buffer_head * h = start_buffer;
    void * b = (void *) buffer_end;
    while ( (b -= 1024) >= ((void *) (h+1)) ) {
        ...
        h->b_data = (char *) b;
        h->b_prev_free = h-1;
        h->b_next_free = h+1;
        h++;
    }
    ...
}
```

就俩变量。

一个是 **buffer_head** 结构的 **h**，代表缓冲头，其指针值是 start_buffer，刚刚我们计算过了，就是图中的内核代码末端地址 end，也就是缓冲区开头。

一个是 **b**，代表缓冲块，指针值是 buffer_end，也就是图中的 2M，就是缓冲区结尾。

缓冲区结尾的 b 每次循环 -1024，也就是一页的值，缓冲区结尾的 h 每次循环 +1（一个 buffer_head 大小的内存），直到碰一块为止。

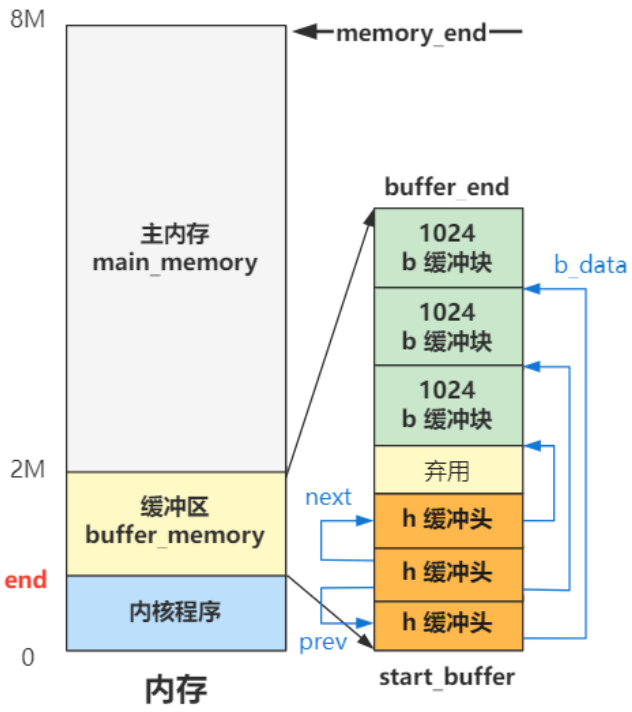


可以看到，其实这个 b 就代表缓冲块，h 代表缓冲头，一个从上往下，一个从下往上。

而且这个过程中，h　被附上了属性值，其中比较关键的是这个　buffer　所表示的数据部分 **b_data**，也就是指向了上面的缓冲块 b。

还有这个 buffer 的前后空闲 buffer 的指针 **b_prev_free** 和 **b_next_free**。

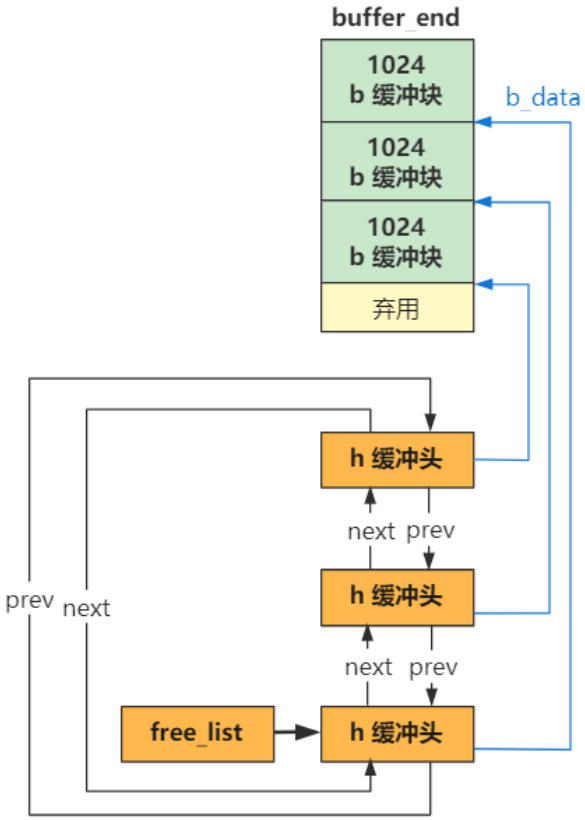
那画成图就是如下这样。



当缓冲头 h 的所有 next 和 prev 指针都指向彼此时，就构成了一个双向链表。继续看。

```
void buffer_init(long buffer_end) {
    ...
    free_list = start_buffer;
    free_list->b_prev_free = h;
    h->b_next_free = free_list;
    ...
}
```

这三行代码，结合刚刚的双向链表 h，我画出图，你就懂了。



看，free_list 指向了缓冲头双向链表的第一个结构，然后就可以顺着这个结构，从双向链表中遍历到任何一个**缓冲头**结构了，而通过缓冲头又可以找到这个缓冲头对应的**缓冲块**。

简单说，**缓冲头就是具体缓冲块的管理结构，而 free_list 开头的双向链表又是缓冲头的管理结构**，整个管理体系就这样建立起来了。

现在，从 free_list 开始遍历，就可以找到这里的所有内容了。

不过，还有最后一个事，能帮助更好管理，往下看。

```
void buffer_init(long buffer_end) {
    ...
    for (i=0;i<307;i++)
        hash_table[i]=NULL;
}
```

一个 307 大小的 **hash_table** 数组，这是干嘛的呢？

其实今天的这个代码在 **buffer.c** 中，而 buffer.c 是在 **fs** 包下的，也就是**文件系统**包下的。所以它今后是为文件系统而服务，具体是内核程序如果需要访问块设备中的数据，就都需要经

过缓冲区来间接地操作。

也就是说，读取块设备的数据（硬盘中的数据），需要先读到缓冲区中，如果缓冲区已有了，就不用从块设备读取了，直接取走。

那怎么知道缓冲区已经有了要读取的块设备中的数据呢？从双向链表从头遍历当然可以，但是这效率可太低了。所以需要一个 hashmap 的结构方便快速查找，这就是 hash_table 这个数组的作用。

现在只是**初始化**这个 hash_table，还并没有哪个地方用到了它，所以我就先简单剧透下。

之后当要读取某个块设备上的数据时，首先要搜索相应的缓冲块，是下面这个函数。

```
#define _hashfn(dev,block) (((unsigned)(dev^block))%307)
#define hash(dev,block) hash_table[_hashfn(dev,block)]

// 搜索合适的缓冲块
struct buffer_head * getblk(int dev,int block) {
    ...
    struct buffer_head bh = get_hash_table(dev,block);
    ...
}


struct buffer_head * get_hash_table(int dev, int block) {
    ...
    find_buffer(dev,block);
    ...
}

static struct buffer_head * find_buffer(int dev, int block) {
    ...
    hash(dev,block);
    ...
}
```

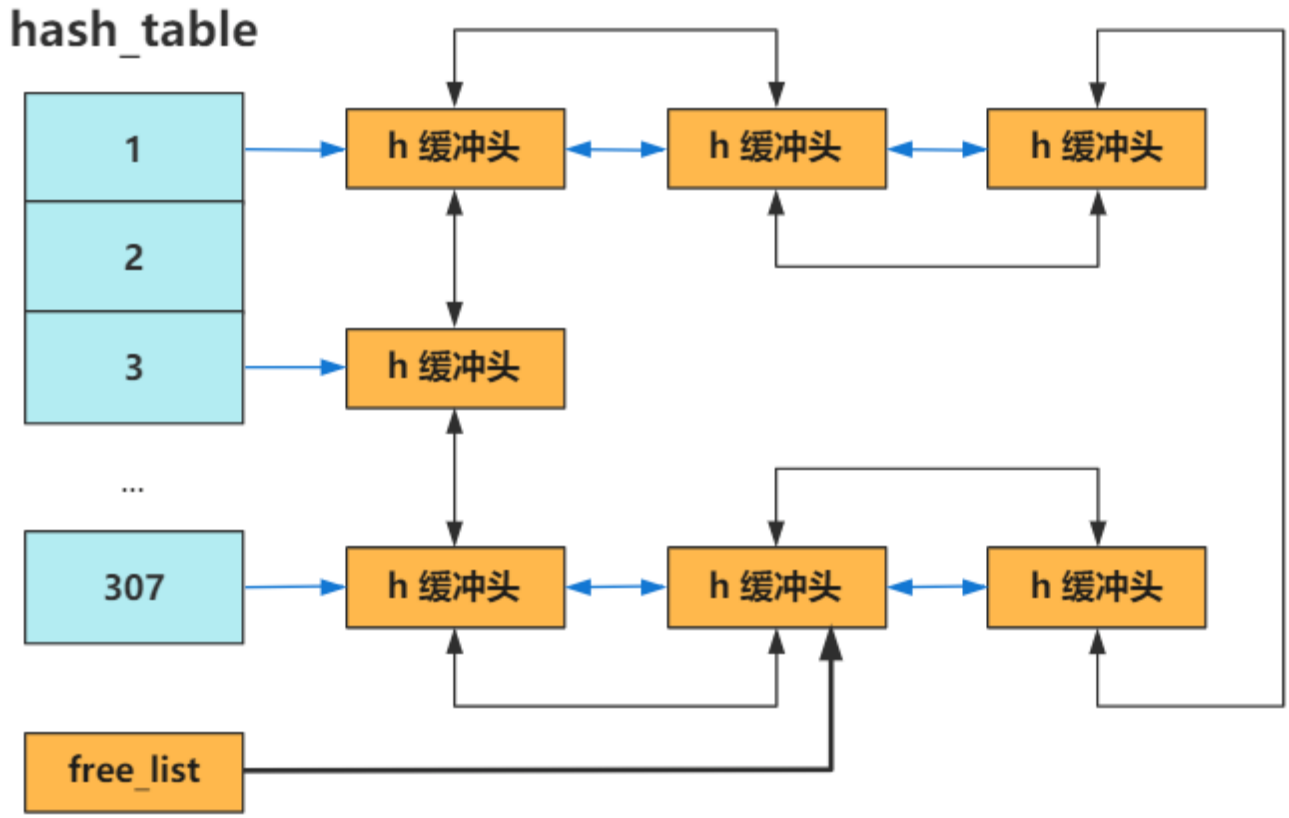
一路跟下来发现，就是通过

dev^block % 307

即

(设备号^逻辑块号) Mod 307

找到在 hash_table 里的索引下标，接下来就和 Java 里的 HashMap 类似，如果哈希冲突就形成链表，画成图就是这样。



哈希表 + 双向链表，如果刷算法题多了，很容易想到这可以实现 **LRU 算法**，没错，之后的缓冲区使用和弃用，正是这个算法发挥了作用。

也就是之后在讲通过文件系统来读取硬盘文件时，都需要使用和弃用这个缓冲区里的内容，缓冲区即是用户进程的内存和硬盘之间的桥梁。

好了好了，再多说几句就把文件系统里读操作讲出来了，压力太大，本章还是主要就了解这个缓冲区的管理工作是如何初始化的，为后面做铺垫。

回过头来看看我们目前的进度吧！

```
void main(void) {
    ...
    mem_init(main_memory_start,memory_end);
    trap_init();
    blk_dev_init();
    chr_dev_init();
    tty_init();
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    hd_init();
    floppy_init();

    sti();
    move_to_user_mode();
    if (!fork()) {init();}
    for(;;) pause();
}
```

整个初始化的部分，就差 **hd_init** 和 **floppy_init** 这两个块设备的初始化还没讲了。

而且幸运的是，floppy_init　是软盘初始化，现在软盘几乎都被淘汰了，计算机中也没有软盘驱动器了，所以这个我们完全可以不看，那就剩下一个 **hd_init 硬盘初始化**了，非常简单！

还记得小时候我特别喜欢收集软盘，里面分门别类存上我做的 Flash 动画，然后在软盘上的那个纸标签上写上文字，表示软盘存了什么，想想看还是回忆呢。

扯远了。

之前的各种初始化工作所建立的数据结构，会在后面各个模块发挥最最核心的作用，**任何操作系统的管理都离不开这些初始化工作所建立的数据结构**，所以一定要把这些根基搭建好，别急别慌。

等初始化工作全部完成，我会专门用一回给大家梳理一下，大家就尽可能把初始化这一大部分的数据结构记在心里吧！

欲知后事如何，且听下回分解。

如果觉得还行，给个 star 谢谢。

（点击阅读原文即可进入 Github 页）

------- 关于本系列 -------

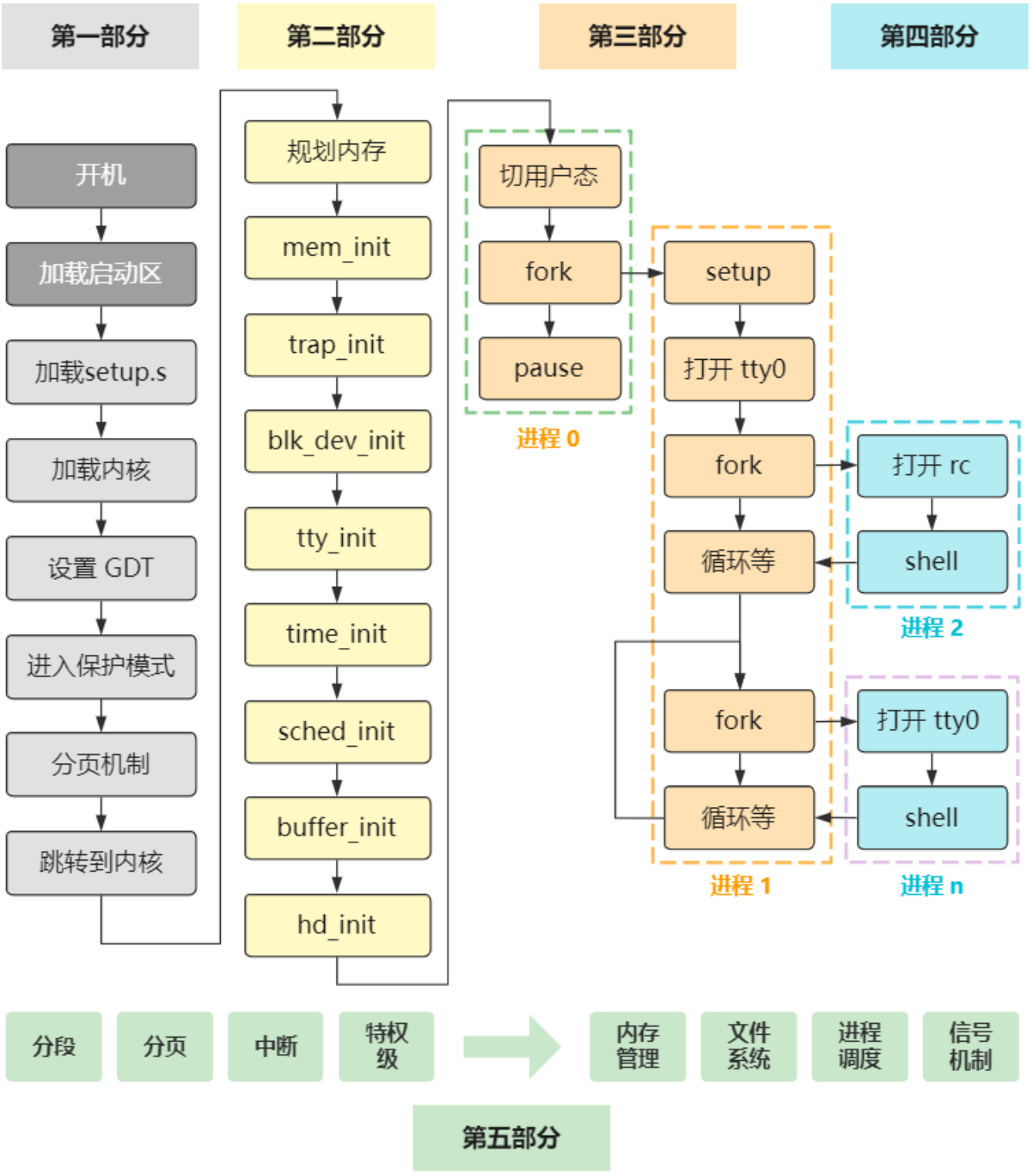
本系列的开篇词看这

闪客新系列！你管这破玩意叫操作系统源码

本系列的扩展资料看这（也可点击**阅读原文**），这里有很多有趣的资料、答疑、互动参与项目，持续更新中，希望有你的参与。

https://github.com/sunym1993/flash-linux0.11-talk

本系列全局视角



最后，祝大家都能追更到系列结束，只要你敢持续追更，并且把每一回的内容搞懂，我就敢让你在系列结束后说一句，我对 Linux 0.11 很熟悉。

公众号更新系列文章不易，阅读量越来越低，希望大家多多传播，不方便的话点个小小的**赞**我也会很开心，谢谢大家咯。

另外，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18回 | 大名鼎鼎的进程调度就是从这里开始的

下一篇

第20回 | 硬盘初始化 hd_init

Read more