第16回 | 按下键盘后为什么屏幕上就会有输出

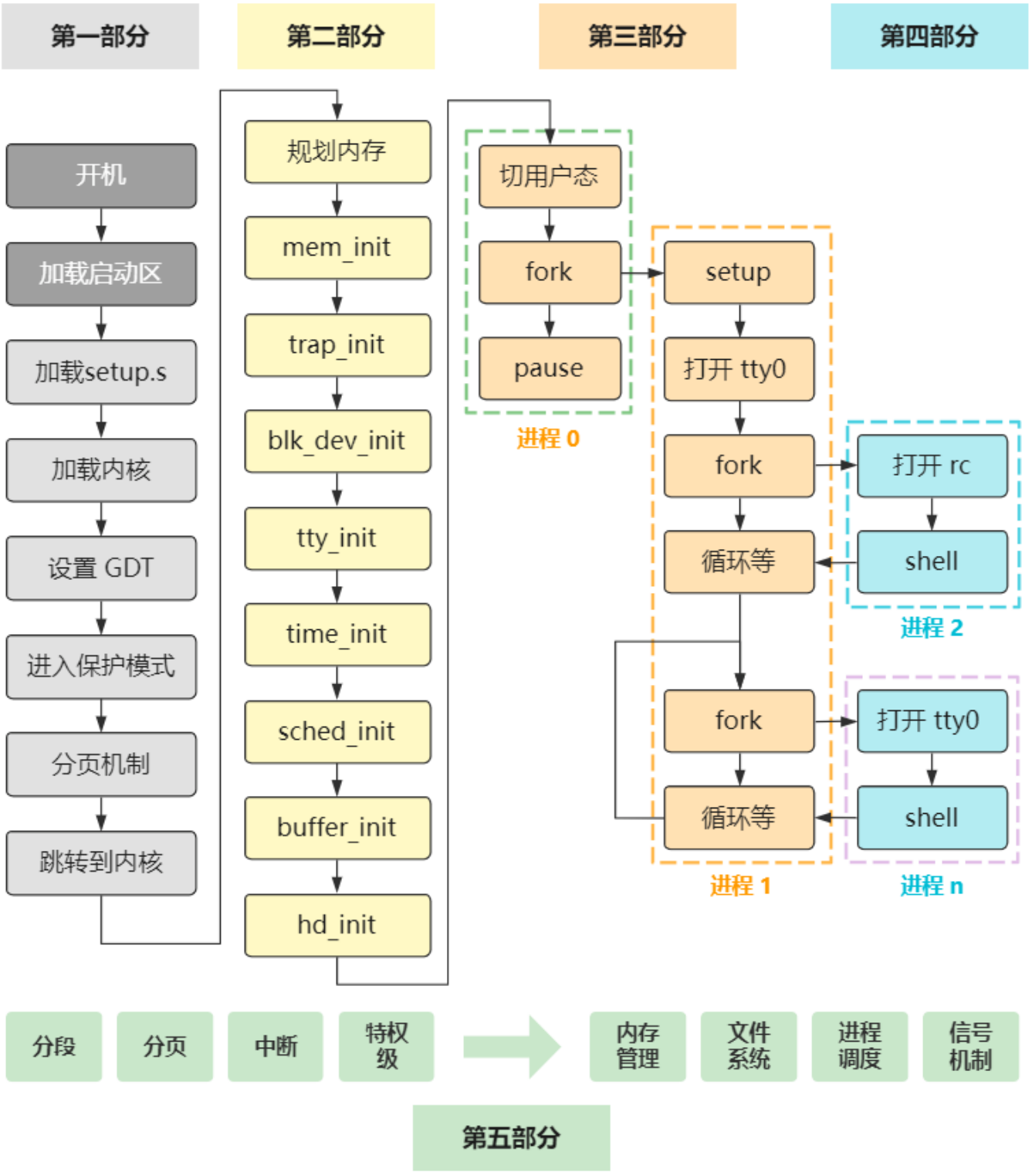
Original 闪客 低并发编程 2022-01-09 16:30

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

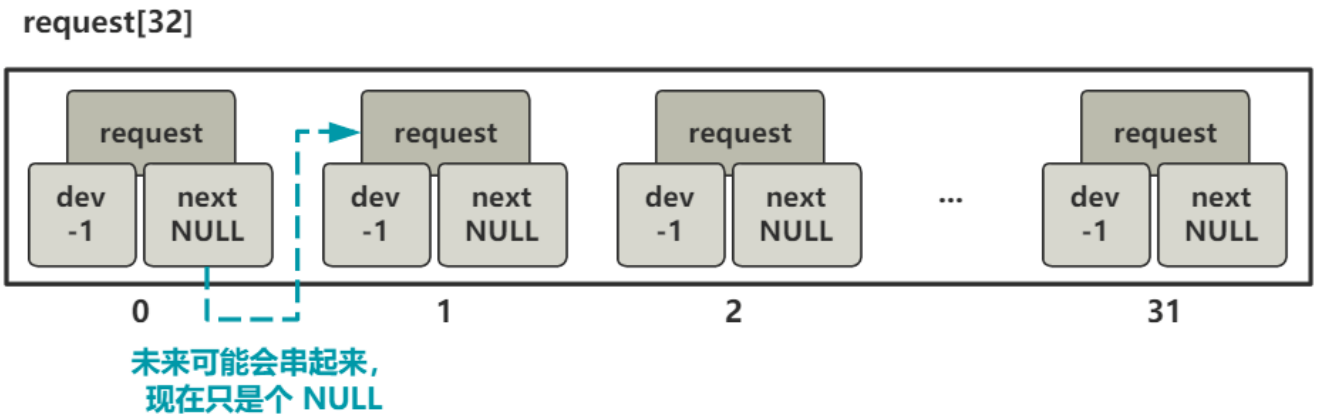
第二部分 大战前期的初始化工作

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，上回书咱们说到，继内存管理结构 mem_map 和中断描述符表 **idt** 建立好之后，我们又在内存中倒腾出一个新的数据结构 request。

并且把它们都放在了一个数组中。



这是**块设备驱动程序**与**内存缓冲区**的桥梁，通过它可以完整地表示一个块设备读写操作要做的事。

我们继续往下看，tty_init。

```
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();
}
```

这个方法执行完成之后，我们将会具备键盘输入到显示器输出字符这个最常用的功能。

打开这个函数后我有点慌。

```
void tty_init(void)
{
    rs_init();
    con_init();
}
```

看来这个方法已经多到需要拆成两个子方法了。

打开第一个方法，还好。

```
void rs_init(void)
{
    set_intr_gate(0x24,rs1_interrupt);
    set_intr_gate(0x23,rs2_interrupt);
    init(tty_table[1].read_q.data);
    init(tty_table[2].read_q.data);
    outb(inb_p(0x21)&0xE7,0x21);
}
```

这个方法是串口中断的开启，以及设置对应的中断处理程序，串口在我们现在的 PC 机上已经很少用到了，所以这个直接忽略，要讲我也不懂。

看第二个方法，这是重点。代码非常长，有点吓人，我先把大体框架写出。

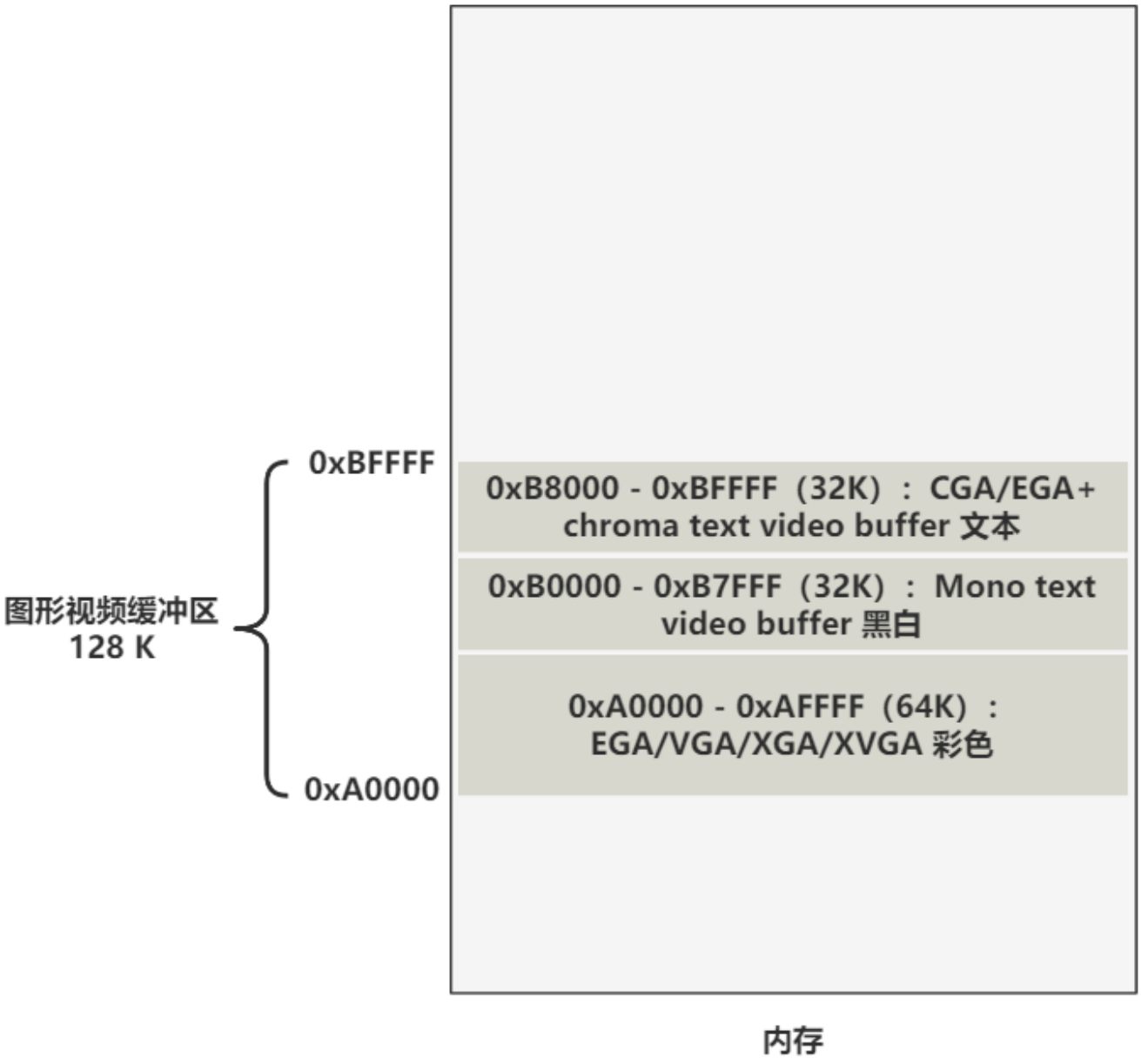
```
void con_init(void) {
    ...
    if (ORIG_VIDEO_MODE == 7) {
        ...
        if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10) {...}
        else {...}
    } else {
        ...
        if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10) {...}
        else {...}
    }
    ...
}
```

可以看出，非常多的 if else。

这是为了应对不同的显示模式，来分配不同的变量值，那如果我们仅仅找出一个显示模式，这些分支就可以只看一个了。

啥是显示模式呢？那我们得简单说说显示，**一个字符是如何显示在屏幕上的呢**？换句话说，如果你可以随意操作内存和 CPU 等设备，你如何操作才能使得你的显示器上，显示一个字符‘a’呢？

我们先看一张图。



内存中有这样一部分区域，是和显存映射的。啥意思，就是你往上图的这些内存区域中写数据，相当于写在了显存中。而往显存中写数据，就相当于在屏幕上输出文本了。

没错，就是这么简单。

如果我们写这一行汇编语句。

```
mov [0xB8000],'h'
```

后面那个 h 相当于汇编编辑器帮我们转换成 ASCII 码的二进制数值，当然我们也可以直接写。

```
mov [0xB8000],0x68
```

其实就是往内存中 **0xB8000** 这个位置写了一个值，只要一写，屏幕上就会是这样。

简单吧，具体说来，这片内存是每两个字节表示一个显示在屏幕上的字符，**第一个是字符的编码，第二个是字符的颜色**，那我们先不管颜色，如果多写几个字符就像这样。

```
mov [0xB8000],'h'
mov [0xB8002],'e'
mov [0xB8004],'l'
mov [0xB8006],'l'
mov [0xB8008],'o'
```

此时屏幕上就会是这样。

是不是贼简单？那我们回过头看刚刚的代码，我们就假设显示模式是我们现在的这种文本模式，那条件分支就可以去掉好多。

代码可以简化成这个样子。

```
#define ORIG_X          (*(unsigned char *)0x90000)
#define ORIG_Y          (*(unsigned char *)0x90001)
void con_init(void) {
    register unsigned char a;
    // 第一部分 获取显示模式相关信息
    video_num_columns = (((*(unsigned short *)0x90006) & 0xff00) >> 8);
    video_size_row = video_num_columns * 2;
    video_num_lines = 25;
    video_page = (*(unsigned short *)0x90004);
    video_erase_char = 0x0720;
    // 第二部分 显存映射的内存区域
    video_mem_start = 0xb8000;
    video_port_reg  = 0x3d4;
    video_port_val  = 0x3d5;
    video_mem_end = 0xba000;
    // 第三部分 滚动屏幕操作时的信息
    origin  = video_mem_start;
    scr_end = video_mem_start + video_num_lines * video_size_row;
    top = 0;
    bottom  = video_num_lines;
    // 第四部分 定位光标并开启键盘中断
    gotoxy(ORIG_X, ORIG_Y);
    set_trap_gate(0x21,&keyboard_interrupt);
    outb_p(inb_p(0x21)&0xfd,0x21);
    a=inb_p(0x61);
    outb_p(a|0x80,0x61);
    outb(a,0x61);
}
```

别看这么多，一点都不难。

首先还记不记得之前汇编语言的时候做的工作，存了好多以后要用的数据在内存中。就在 第五回 | 进入保护模式前的最后一次折腾内存

内存地址	长度(字节)	名称

0x90000	2	光标位置
0x90002	2	扩展内存数
0x90004	2	显示页面
0x90006	1	显示模式
0x90007	1	字符列数
0x90008	2	未知
0x9000A	1	显示内存
0x9000B	1	显示状态
0x9000C	2	显卡特性参数
0x9000E	1	屏幕行数
0x9000F	1	屏幕列数
0x90080	16	硬盘1参数表
0x90090	16	硬盘2参数表
0x901FC	2	根设备号

所以，**第一部分**获取 0x90006 地址处的数据，就是获取显示模式等相关信息。

第二部分就是显存映射的内存地址范围，我们现在假设是 CGA 类型的文本模式，所以映射的内存是从 0xB8000 到 0xBA000。

第三部分是设置一些滚动屏幕时需要的参数，定义顶行和底行是哪里，这里顶行就是第一行，底行就是最后一行，很合理。

第四部分是把光标定位到之前保存的光标位置处（取内存地址 0x90000 处的数据），然后设置并开启键盘中断。

开启键盘中断后，键盘上敲击一个按键后就会触发中断，中断程序就会读键盘码转换成 ASCII 码，然后写到光标处的内存地址，也就相当于往显存写，于是这个键盘敲击的字符就显示在了屏幕上。

这一切具体是怎么做到的呢？我们先看看我们干了什么。

1. 我们现在根据已有信息已经可以实现往屏幕上的任意位置写字符了，而且还能指定颜色。

2. 并且，我们也能接受键盘中断，根据键盘码中断处理程序就可以得知哪个键按下了。

有了这俩功能，那我们想干嘛还不是为所欲为？

好，接下来我们看看代码是怎么处理的，很简单。一切的起点，就是第四步的 **gotoxy** 函数，定位当前光标。

```
#define ORIG_X          (*(unsigned char *)0x90000)
#define ORIG_Y          (*(unsigned char *)0x90001)
void con_init(void) {
    ...
    // 第四部分 定位光标并开启键盘中断
    gotoxy(ORIG_X, ORIG_Y);
    ...
}
```

这里面干嘛了呢？

```
static inline void gotoxy(unsigned int new_x,unsigned int new_y) {
   ...
   x = new_x;
   y = new_y;
   pos = origin + y*video_size_row + (x<<1);
}
```

就是给 **x y pos** 这三个参数附上了值。

其中 **x** 表示光标在哪一列，**y** 表示光标在哪一行，**pos** 表示根据列号和行号计算出来的内存指针，也就是往这个 pos 指向的地址处写数据，就相当于往控制台的 x 列 y 行处写入字符了，简单吧?

然后，当你按下键盘后，触发键盘中断，之后的程序调用链是这样的。

```
_keyboard_interrupt:
    ...
    call _do_tty_interrupt
    ...

void do_tty_interrupt(int tty) {
   copy_to_cooked(tty_table+tty);
}

void copy_to_cooked(struct tty_struct * tty) {
    ...
    tty->write(tty);
    ...
}

// 控制台时 tty 的 write 为 con_write 函数
void con_write(struct tty_struct * tty) {
    ...
    __asm__("movb _attr,%%ah\n\t"
      "movw %%ax,%1\n\t"
      ::"a" (c),"m" (*(short *)pos)
      :"ax");
     pos += 2;
     x++;
    ...
}
```

前面的过程不用管，我们看最后一个函数 con_write 中的关键代码。

__asm__ 内联汇编，就是把键盘输入的字符 **c** 写入 **pos** 指针指向的内存，相当于往屏幕输出了。

之后两行 pos+=2 和 x++，就是调整所谓的**光标**。

你看，写入一个字符，最底层，**其实就是往内存的某处写个数据，然后顺便调整一下光标。**

由此我们也可以看出，光标的本质，其实就是这里的 x y pos 这仨变量而已。

我们还可以做**换行效果**，当发现光标位置处于某一行的结尾时（这个应该很好算吧，我们都知道屏幕上一共有几行几列了），就把光标计算出一个新值，让其处于下一行的开头。

就一个小计算公式即可搞定，仍然在 con_write 源码处有体现，就是判断列号 x 是否大于了总列数。

```
void con_write(struct tty_struct * tty) {
    ...
    if (x>=video_num_columns) {
        x -= video_num_columns;
        pos -= video_size_row;
        lf();
  }
  ...
}

static void lf(void) {
   if (y+1<bottom) {
      y++;
      pos += video_size_row;
      return;
   }
 ...
}
```

相似的，我们还可以实现**滚屏**的效果，无非就是当检测到光标已经出现在最后一行最后一列

了，那就把每一行的字符，都复制到它上一行，其实就是算好哪些内存地址上的值，拷贝到哪些内存地址，就好了。

这里大家自己看源码寻找。

所以，有了这个初始化工作，我们就可以利用这些信息，弄几个小算法，实现各种我们常见控制台的操作。

或者换句话说，我们见惯不怪的控制台，**回车**、**换行**、**删除**、**滚屏**、**清屏**等操作，其实底层都要实现相应的代码的。

所以 console.c 中的其他方法就是做这个事的，我们就不展开每一个功能的方法体了，简单看看有哪些方法。

```
// 定位光标的
static inline void gotoxy(unsigned int new_x, unsigned int new_y){}
// 滚屏，即内容向上滚动一行
static void scrup(void){}
// 光标同列位置下移一行
static void lf(int currcons){}
// 光标回到第一列
static void cr(void){}
...
// 删除一行
static void delete_line(void){}
```

内容繁多，但没什么难度，只要理解了基本原理即可了。

OK，整个 **console.c** 就讲完了，要知道这个文件可是整个内核中代码量最大的文件，可是功能特别单一，也都很简单，主要是处理键盘各种不同的按键，需要写好多 switch case 等语句，十分麻烦，我们这里就完全没必要去展开了，就是个苦力活。

到这里，我们就正式讲完了 **tty_init** 的作用。

在此之后，内核代码就可以用它来方便地在控制台输出字符啦！这在之后内核想要在启动过程中告诉用户一些信息，以及后面内核完全建立起来之后，由用户用 shell 进行操作时手动输入命令，都是可以用到这里的代码的！

让我们继续向前进发，看下一个被初始化的倒霉鬼是什么东东。

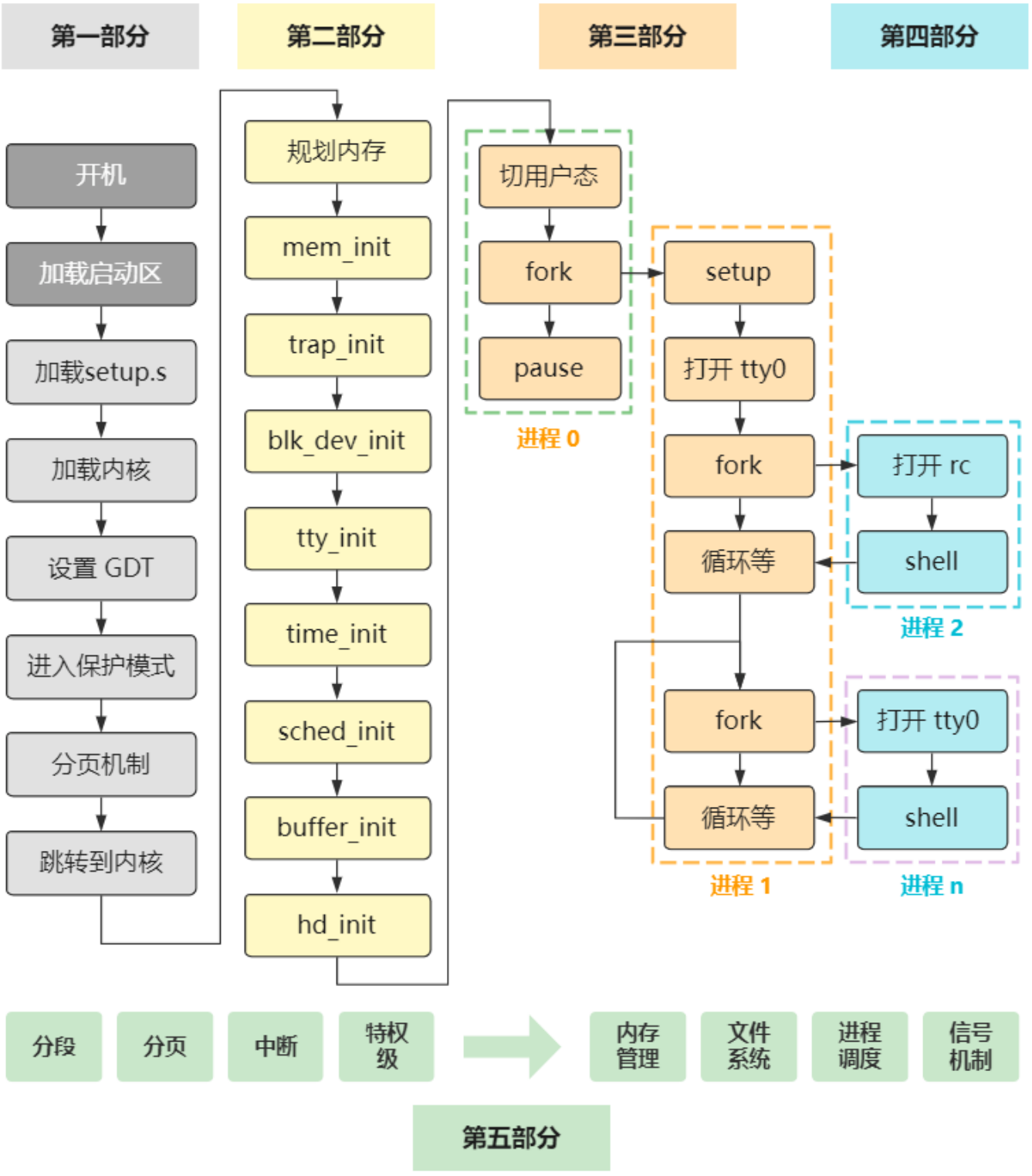
欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这
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

读取硬盘前的准备工作有哪些?

下一篇

第17回 | 原来操作系统获取时间的方式也这么 low

Read more