第17回 | 原来操作系统获取时间的方式也这么 low

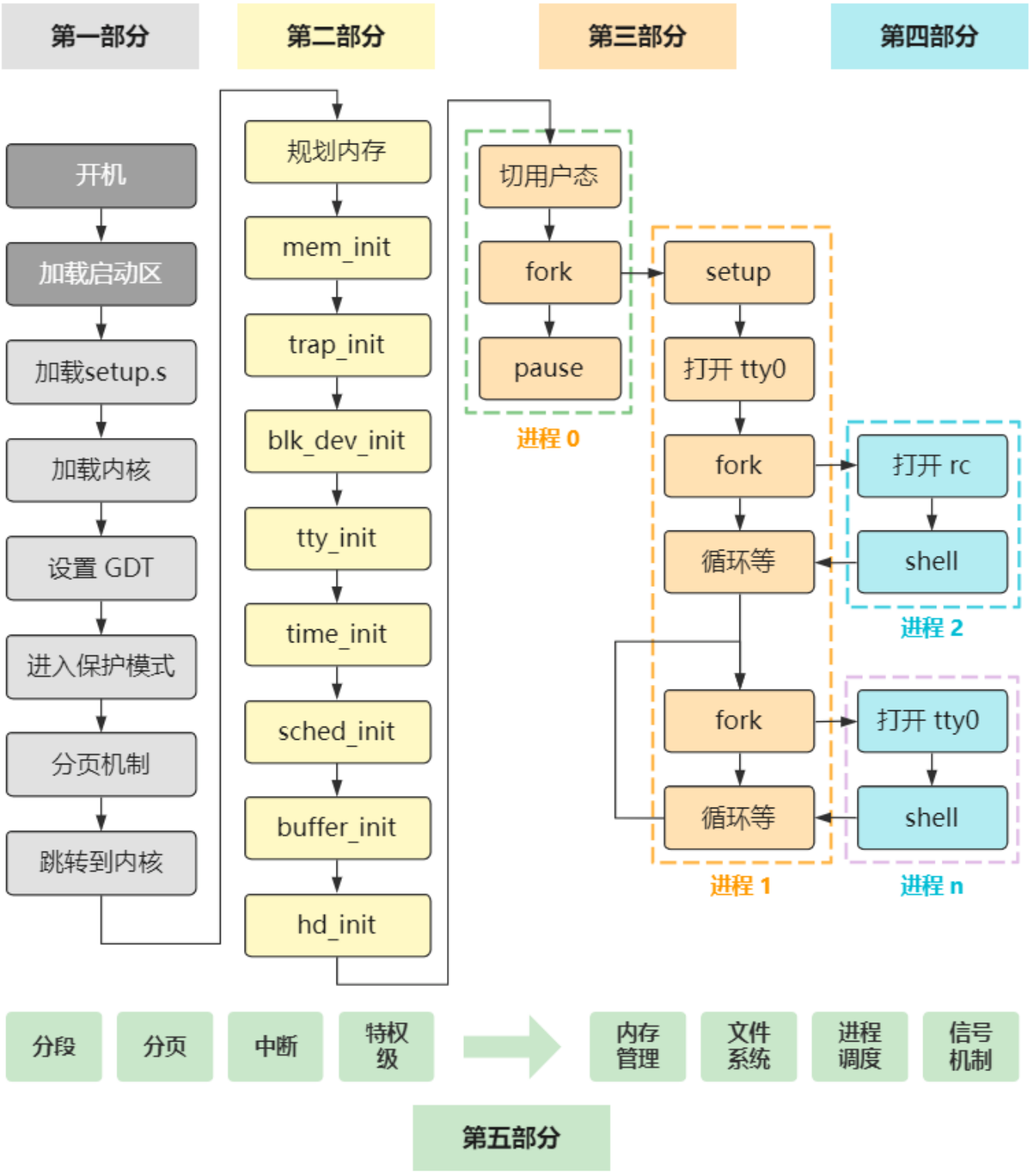
Original 闪客 低并发编程 2022-01-12 16:30

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

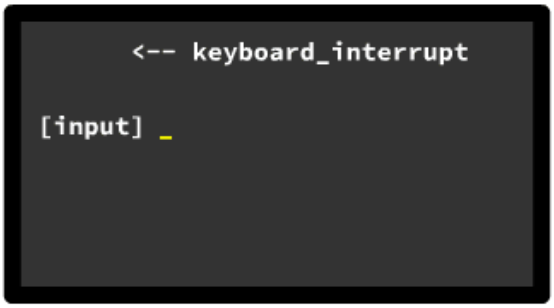
第二部分 大战前期的初始化工作

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，通过初始化控制台的 tty_init 操作，内核代码可以很方便地在控制台输出字符啦！

作为用户也可以通过敲击键盘，或调用诸如 printf 这样的库函数，在屏幕上输出信息，同时支持换行和滚屏等友好设计，这些都是 tty_init 初始化，以及其对外封装的小功能函数，来实现的。



我们继续往下看下一个初始化的倒霉鬼，**time_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();
}
```

曾经我很好奇，**操作系统是怎么获取到当前时间的呢**？

当然，现在都联网了，可以从网络上实时同步。那当没有网络时，为什么操作系统在启动之后，可以显示出当前时间呢？难道操作系统在电脑关机后，依然不停地在某处运行着，勤勤恳恳数着秒表么？

当然不是，那我们今天就打开这个 **time_init** 函数一探究竟。

打开这个函数后我又是很开心，因为很短，且没有更深入的方法调用。

```
#define CMOS_READ(addr) ({ \
    outb_p(0x80|addr,0x70); \
    inb_p(0x71); \
})

#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)

static void time_init(void) {
    struct tm time;
    do {
        time.tm_sec = CMOS_READ(0);
        time.tm_min = CMOS_READ(2);
        time.tm_hour = CMOS_READ(4);
        time.tm_mday = CMOS_READ(7);
        time.tm_mon = CMOS_READ(8);
        time.tm_year = CMOS_READ(9);
    } while (time.tm_sec != CMOS_READ(0));
    BCD_TO_BIN(time.tm_sec);
    BCD_TO_BIN(time.tm_min);
    BCD_TO_BIN(time.tm_hour);
    BCD_TO_BIN(time.tm_mday);
    BCD_TO_BIN(time.tm_mon);
    BCD_TO_BIN(time.tm_year);
    time.tm_mon--;
    startup_time = kernel_mktime(&time);
}
```

梦想的代码呀！

那主要就是对 **CMOS_READ** 和 **BCD_TO_BIN** 都是啥意思展开讲一下就明白了了。

首先是 **CMOS_READ**

```
#define CMOS_READ(addr) ({ \
    outb_p(0x80|addr,0x70); \
    inb_p(0x71); \
})
```

就是对一个端口先 **out** 写一下，再 **in** 读一下。

这是 CPU 与外设交互的一个基本玩法，CPU 与外设打交道基本是通过端口，往某些端口写值来表示要这个外设干嘛，然后从另一些端口读值来接受外设的反馈。

至于这个外设内部是怎么实现的，对使用它的操作系统而言，是个黑盒，无需关心。那对于我们程序员来说，就更不用关心了。

对 CMOS 这个外设的交互讲起来可能没感觉，我们看看与硬盘的交互。

最常见的就是读硬盘了，我们看硬盘的端口表。

端口	读	写
0x1F0	数据寄存器	数据寄存器
0x1F1	错误寄存器	特征寄存器
0x1F2	扇区计数寄存器	扇区计数寄存器
0x1F3	扇区号寄存器或 LBA 块地址 0~7	扇区号或 LBA 块地址 0~7
0x1F4	磁道数低 8 位或 LBA 块地址 8~15	磁道数低 8 位或 LBA 块地址 8~15
0x1F5	磁道数高 8 位或 LBA 块地址 16~23	磁道数高 8 位或 LBA 块地址 16~23
0x1F6	驱动器/磁头或 LBA 块地址 24~27	驱动器/磁头或 LBA 块地址 24~27
0x1F7	命令寄存器或状态寄存器	命令寄存器

那读硬盘就是，往除了第一个以外的后面几个端口写数据，告诉要读硬盘的哪个扇区，读多少。然后再从 0x1F0 端口一个字节一个字节的读数据。这就完成了一次硬盘读操作。

如果觉得不够具体，那来个具体的版本。

1. 在 0x1F2 写入要读取的扇区数
2. 在 0x1F3 ~ 0x1F6 这四个端口写入计算好的起始 LBA 地址
3. 在 0x1F7 处写入读命令的指令号
4. 不断检测 0x1F7 （此时已成为状态寄存器的含义）的忙位
5. 如果第四步骤为不忙，则开始不断从 0x1F0 处读取数据到内存指定位置，直到读完

看，是不是对 CPU 最底层是如何与外设打交道有点感觉了？是不是也不难？就是按照人家的操作手册，然后无脑按照要求读写端口就行了。

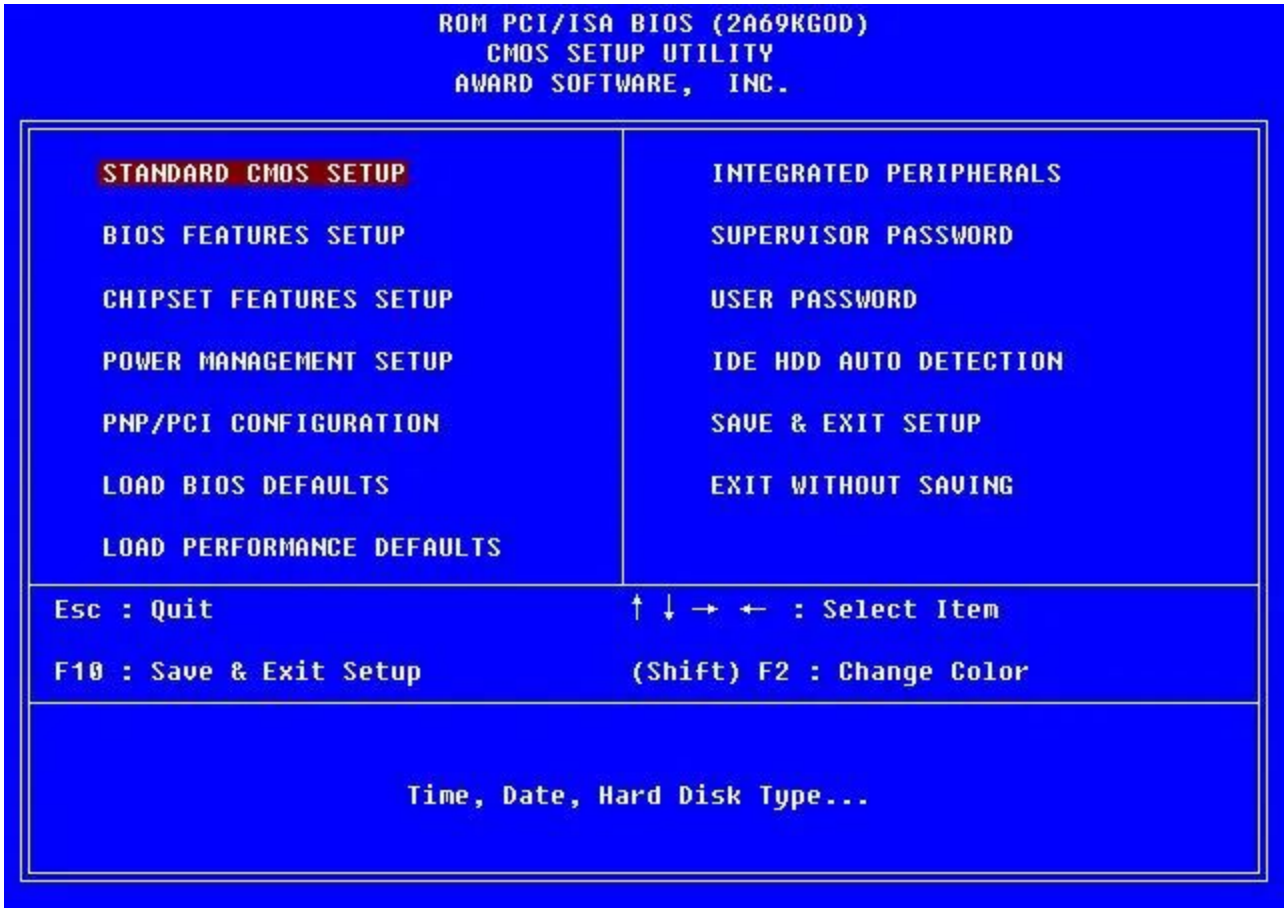
当然，读取硬盘的这个无脑循环，可以 **CPU** 直接读取并做写入内存的操作，这样就会占用 CPU 的计算资源。

也可以交给 **DMA** 设备去读，解放 CPU，但和硬盘的交互，通通都是按照硬件手册上的端口说明，来操作的，实际上也是做了一层封装。

好了，我们已经学会了和一个外设打交道的基本玩法了。

那我们代码中要打交道的是哪个外设呢？就是 **CMOS**。

它是主板上的一个可读写的 RAM 芯片，你在开机时长按某个键就可以进入设置它的页面。



那我们的代码，其实就是与它打交道，获取它的一些数据而已。

我们回过头看代码。

```
static void time_init(void) {
    struct tm time;
    do {
        time.tm_sec = CMOS_READ(0);
        time.tm_min = CMOS_READ(2);
        time.tm_hour = CMOS_READ(4);
        time.tm_mday = CMOS_READ(7);
        time.tm_mon = CMOS_READ(8);
        time.tm_year = CMOS_READ(9);
    } while (time.tm_sec != CMOS_READ(0));
    BCD_TO_BIN(time.tm_sec);
    BCD_TO_BIN(time.tm_min);
    BCD_TO_BIN(time.tm_hour);
    BCD_TO_BIN(time.tm_mday);
    BCD_TO_BIN(time.tm_mon);
    BCD_TO_BIN(time.tm_year);
    time.tm_mon--;
    startup_time = kernel_mktime(&time);
}
```

前面几个赋值语句 **CMOS_READ** 就是通过读写 CMOS 上的指定端口，依次获取**年月日时分秒**等信息。具体咋操作代码上也写了，也是按照 CMOS 手册要求的读写指定端口就行了，我们就不展开了。

所以你看，其实操作系统程序，也是要依靠与一个外部设备打交道，来获取这些信息的，并不是它自己有什么魔力。操作系统最大的魅力，就在于它借力完成了一项伟大的事，借 CPU 的力，借硬盘的力，借内存的力，以及现在借 CMOS 的力。

至于 CMOS 又是如何知道时间的，这个就不在我们讨论范围了。

接下来 **BCD_TO_BIN** 就是 BCD 转换成 BIN，因为从 CMOS 上获取的这些年月日都是 BCD 码值，需要转换成存储在我们变量上的二进制数值，所以需要一个小算法来转换一下，没什么意思。

最后一步 **kernel_mktime** 也很简单，就是根据刚刚的那些时分秒数据，计算从 **1970 年 1 月 1 日 0 时**起到开机当时经过的秒数，作为开机时间，存储在 **startup_time** 这个变量里。

想研究可以仔细看看这段代码，不过我觉得这种细节不必看。

```
startup_time = kernel_mktime(&time);


// kernel/mktime.c
long kernel_mktime(struct tm * tm)
{
    long res;
    int year;
    year = tm->tm_year - 70;
    res = YEAR*year + DAY*((year+1)/4);
    res += month[tm->tm_mon];
    if (tm->tm_mon>1 && ((year+2)%4))
        res -= DAY;
    res += DAY*(tm->tm_mday-1);
    res += HOUR*tm->tm_hour;
    res += MINUTE*tm->tm_min;
    res += tm->tm_sec;
    return res;
}
```

就这。

所以今天其实就是，计算出了一个 **startup_time** 变量而已，至于这个变量今后会被谁用，怎么用，那就是后话了。

相信你逐渐也体会到了，此时操作系统好多地方都是用外设要求的方式去询问，比如硬盘信息、显示模式，以及今天的开机时间的获取等。

所以至少到目前来说，你还不应该感觉操作系统有多么的“高端”，很多时候都是繁琐地，读人家的硬件手册，获取到想要的的信息，拿来给自己用，或者对其进行各种设置。

但你一定要耐得住寂寞，真正体现操作系统的强大设计之处，还得接着往下读。

欲知后事如何，且听下回分解。

------- 关于本系列 -------

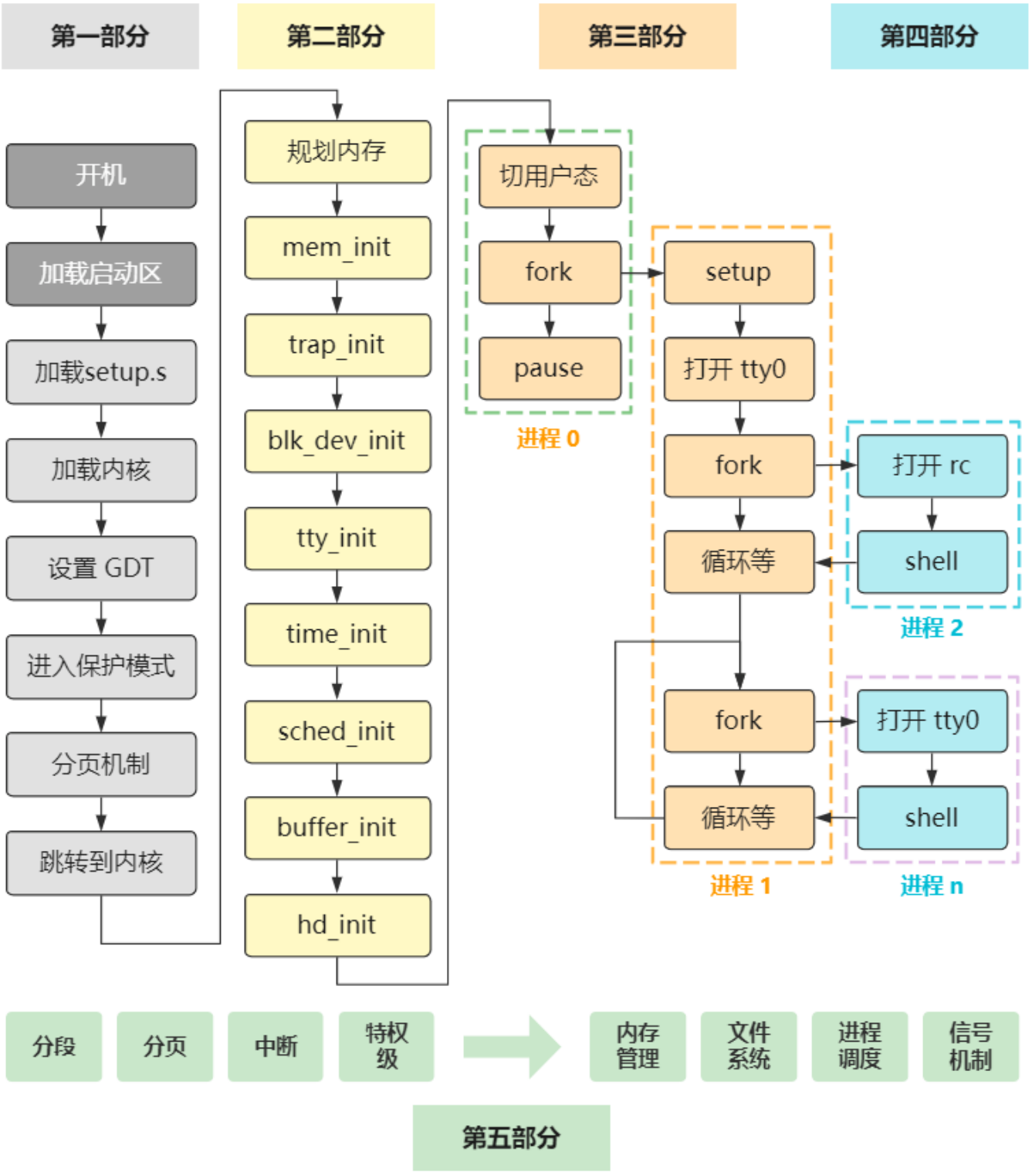
本系列的开篇词看这

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

第16回 | 按下键盘后为什么屏幕上就会有输出

下一篇

第18回 | 大名鼎鼎的进程调度就是从这里开始的

Read more