一个新进程的诞生（三）如果让你来设计进程调度

Original 闪客 低并发编程 2022-02-16 16:30

收录于合集
#操作系统源码 43 #一个新进程的诞生 8

本系列作为 你管这破玩意叫操作系统源码 的第三大部分，讲述了操作系统第一个进程从无到有的诞生过程，这一部分你将看到内核态与用户态的转换、进程调度的上帝视角、系统调用的全链路、fork 函数的深度剖析。

不要听到这些陌生的名词就害怕，跟着我一点一点了解他们的全貌，你会发现，这些概念竟然如此活灵活现，如此顺其自然且合理地出现在操作系统的启动过程中。

本篇章作为一个全新的篇章，需要前置篇章的知识体系支撑。

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

当然，没读过的也问题不大，我都会在文章里做说明，如果你觉得有困惑，就去我告诉你的相应章节回顾就好了，放宽心。

------- 第三部分目录 -------

------- 正文开始 -------

书接上回，上回书咱们说到，操作系统通过 **move_to_user_mode** 方法，通过伪造一个中断和中断返回，巧妙地从内核态切换到了用户态。

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

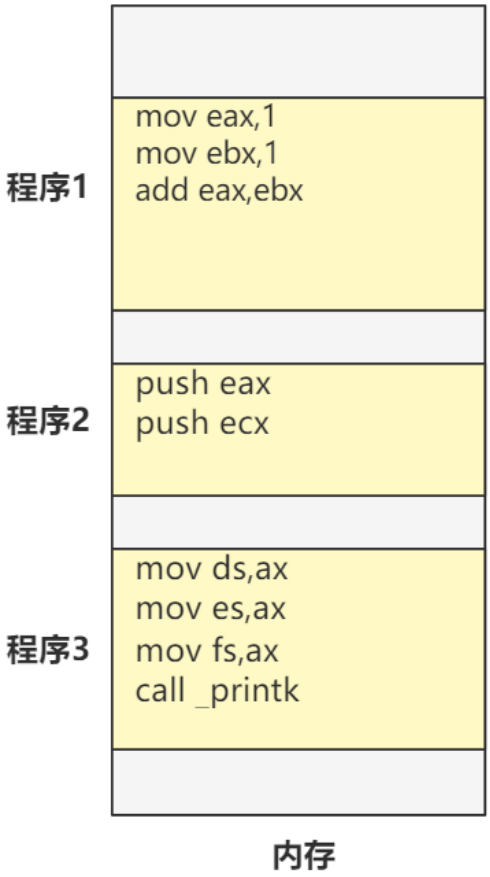
今天，本来应该再往下讲 fork。

但这个是创建新进程的过程，是一个很能体现操作系统设计的地方。

所以我们先别急着看代码，我们今天就头脑风暴一下，就是**如果让你来设计整个进程调度**，你会怎么搞？

别告诉我你先设计锁、设计 volatile 啥的，这都不是进程调度本身需要关心的最根本问题。

进程调度本质是什么？很简单，假如有三段代码被加载到内存中。



进程调度就是让 CPU 一会去程序 1 的位置处运行一段时间，一会去程序 2 的位置处运行一段时间。

嗯，就这么简单，别反驳我，接着往下看。

整体流程设计

如何做到刚刚说的，一会去这运行，一会去那运行？

第一种办法就是，程序 1 的代码里，每隔几行就写一段代码，主动放弃自己的执行权，跳转到程序 2 的地方运行。然后程序 2 也是如此。

但这种依靠程序自己的办法肯定不靠谱。

所以**第二种办法**就是，由一个不受任何程序控制的，第三方的不可抗力，每隔一段时间就中断一下 CPU 的运行，然后跳转到一个特殊的程序那里，这个程序通过某种方式获取到 CPU 下一个要运行的程序的地址，然后跳转过去。

这个每隔一段时间就中断 CPU 的不可抗力，就是由定时器触发的**时钟中断**。

不知道你是否还记得，这个定时器和时钟中断，早在 第18回 | 大名鼎鼎的进程调度就是从这里开始的 里讲的 **sched_init** 函数里就搞定了。

而那个特殊的程序，就是具体的**进程调度函数**了。

好了，整个流程就这样处理完了，那么应该设计什么样的**数据结构**，来支持这个流程呢？不妨假设这个结构叫 **tast_struct**。

```
struct task_struct {
    ?
}
```

换句话说，你总得有一个结构来记录各个进程的信息，比如它上一次执行到哪里了，要不 CPU 就算决定好了要跳转到你这个进程上运行，具体跳到哪一行运行，总得有个地方存吧？

我们一个个问题抛开来看。

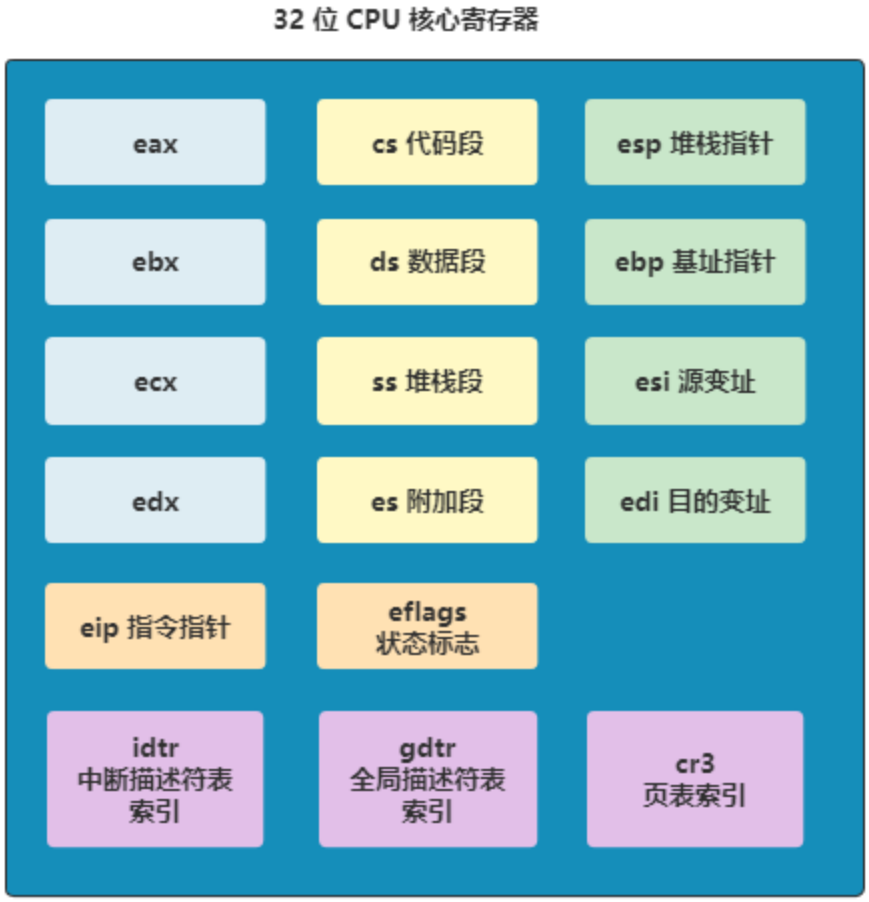
上下文环境

每个程序最终的本质就是执行指令。这个过程会涉及**寄存器**，**内存**和**外设端口**。

内存还有可能设计成相互错开的，互不干扰，比如进程 1 你就用 0~1K 的内存空间，进程 2 就用 1K~2K 的内存空间，咱谁也别影响谁。

虽然有点浪费空间，而且对程序员十分不友好，但起码还是能实现的。

不过寄存器一共就那么点，肯定做不到互不干扰，可能一个进程就把寄存器全用上了，那其他进程咋整。



比如程序 1 刚刚往 eax 写入一个值，准备用，这时切换到进程 2 了，又往 eax 里写入了一个值。那么之后再切回进程 1 的时候，就出错了。

所以最稳妥的做法就是，每次切换进程时，都把当前这些寄存器的值存到一个地方，以便之后切换回来的时候恢复。

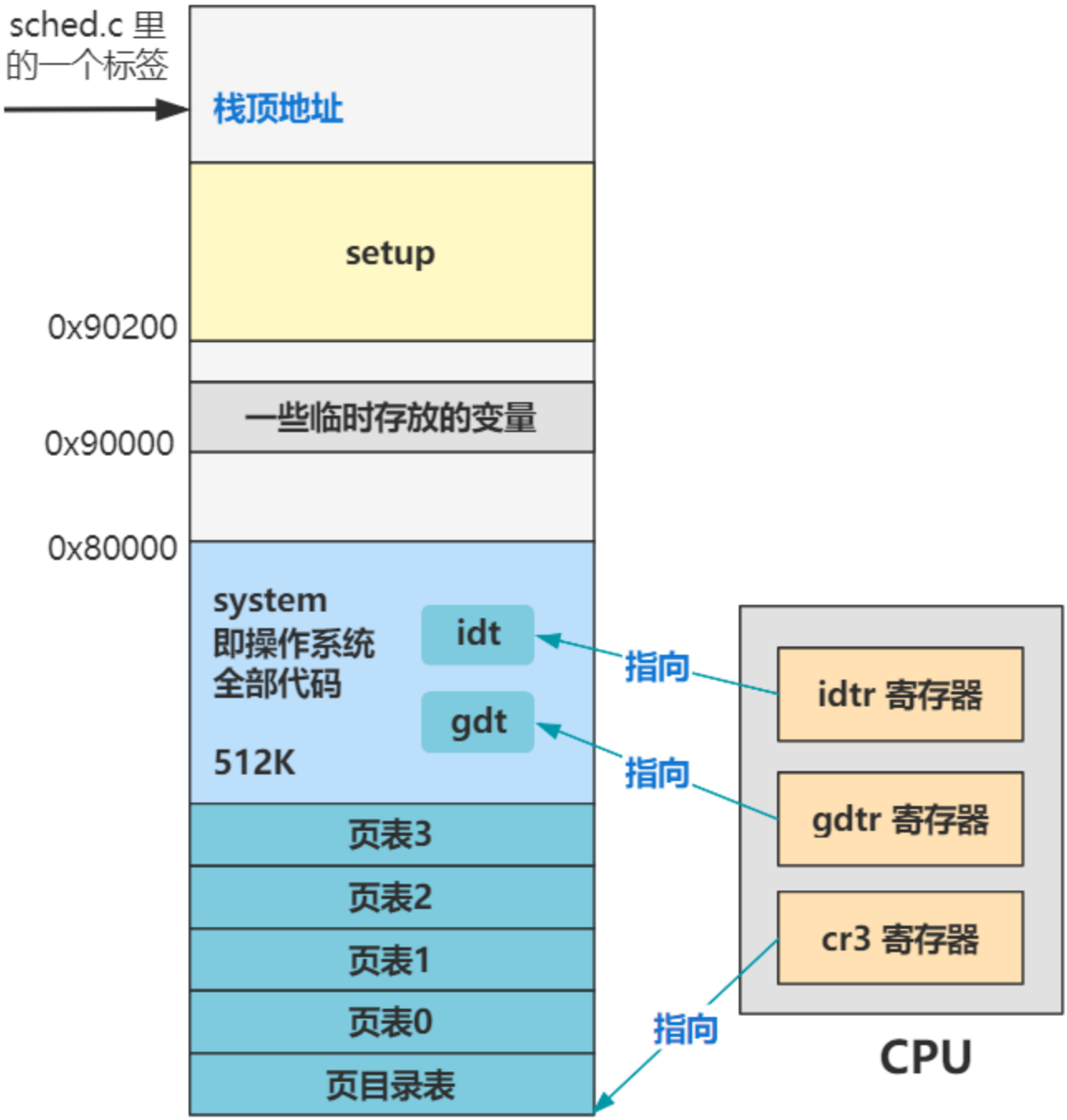
Linux 0.11 就是这样做的，每个进程的结构 task_struct 里面，有一个叫 **tss** 的结构，存储的就是 CPU 这些**寄存器**的信息。

```
struct task_struct {
    ...
    struct tss_struct tss;
}

struct tss_struct {
    long    back_link;  /* 16 high bits zero */
    long    esp0;
    long    ss0;        /* 16 high bits zero */
    long    esp1;
    long    ss1;        /* 16 high bits zero */
    long    esp2;
    long    ss2;        /* 16 high bits zero */
    long    cr3;
    long    eip;
    long    eflags;
    long    eax,ecx,edx,ebx;
    long    esp;
    long    ebp;
    long    esi;
    long    edi;
    long    es;     /* 16 high bits zero */
    long    cs;     /* 16 high bits zero */
    long    ss;     /* 16 high bits zero */
    long    ds;     /* 16 high bits zero */
    long    fs;     /* 16 high bits zero */
    long    gs;     /* 16 high bits zero */
    long    ldt;        /* 16 high bits zero */
    long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
    struct i387_struct i387;
};
```

这里提个细节。

你发现 tss 结构里还有个 **cr3** 不？它表示 cr3 寄存器里存的值，而 cr3 寄存器是指向页目录表首地址的。



那么指向不同的页目录表，整个页表结构就是完全不同的一套，那么线性地址到物理地址的映射关系就有能力做到不同。

也就是说，在我们刚刚假设的理想情况下，不同程序用不同的内存地址可以做到内存互不干扰。

但是有了这个 cr3 字段，就完全可以无需由各个进程自己保证不和其他进程使用的内存冲突，因为只要建立不同的映射关系即可，由操作系统来建立不同的页目录表并替换 cr3 寄存器即可。

这也可以理解为，保存了**内存映射的上下文信息**。

当然 Linux 0.11 并不是通过替换 cr3 寄存器来实现内存互不干扰的，它的实现更为简单，这是后话了。

运行时间信息

如何判断一个进程该让出 CPU 了，切换到下一个进程呢？

总不能是每次时钟中断时都切换一次吧？一来这样不灵活，二来这完全依赖时钟中断的频率，有点危险。

所以一个好的办法就是，给进程一个属性，叫**剩余时间片**，每次时钟中断来了之后都 **-1**，如果减到 0 了，就触发切换进程的操作。

在 Linux 0.11 里，这个属性就是 **counter**。

```
struct task_struct {
    ...
    long counter;
    ...
    struct tss_struct tss;
}
```

而他的用法也非常简单，就是每次中断都判断一下是否到 0 了。

```
void do_timer(long cpl) {
    ...
    // 当前线程还有剩余时间片，直接返回
    if ((--current->counter)>0) return;
    // 若没有剩余时间片，调度
    schedule();
}
```

如果还没到 0，就直接返回，相当于这次时钟中断什么也没做，仅仅是给当前进程的时间片属

性做了 -1 操作。

如果已经到 0 了，就触发**进程调度**，选择下一个进程并使 CPU 跳转到那里运行。

进程调度的逻辑就是在 **schedule** 函数里，怎么调，我们先不管。

优先级

上面那个 counter 一开始的时候该是多少呢？而且随着 counter 不断递减，减到 0 时，下一轮回中这个 counter 应该赋予什么值呢？

其实这俩问题都是一个问题，就是 **counter 的初始化**问题，也需要有一个属性来记录这个值。

往宏观想一下，这个值越大，那么 counter 就越大，那么每次轮到这个进程时，它在 CPU 中运行的时间就越长，也就是这个进程比其他进程得到了更多 CPU 运行的时间。

那我们可以把这个值称为**优先级**，是不是很形象。

```
struct task_struct {
    ...
    long counter;
    long priority;
    ...
    struct tss_struct tss;
}
```

每次一个进程初始化时，都把 counter 赋值为这个 priority，而且当 counter 减为 0 时，下一次分配时间片，也赋值为这个。

其实叫啥都行，反正就是这么用的，就叫优先级吧。

进程状态

其实我们有了上面那三个信息，就已经可以完成进程的调度了。

甚至如果你的操作系统让所有进程都得到同样的运行时间，连 counter 和 priority 都不用记录，就操作系统自己定一个固定值一直递减，减到 0 了就随机切一个新进程。

这样就仅仅维护好寄存器的上下文信息 tss 就好了。

但我们总要不断优化以适应不同场景的用户需求的，那我们再优化一个细节。

很简单的一个场景，一个进程中有一个读取硬盘的操作，发起读请求后，要等好久才能得到硬盘的中断信号。

那这个时间其实该进程再占用着 CPU 也没用，此时就可以选择主动放弃 CPU 执行权，然后再把自己的状态标记为等待中。

意思是告诉进程调度的代码，先别调度我，因为我还在等硬盘的中断，现在轮到我了也没用，把机会给别人吧。

那这个状态可以记录一个属性了，叫 **state**，记录了此时**进程的状态**。

```
struct task_struct {
    long state;
    long counter;
    long priority;
    ...
    struct tss_struct tss;
}
```

而这个进程的状态在 Linux 0.11 里有这么五种。

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE           3
#define TASK_STOPPED          4
```

好了，目前我们这几个字段，就已经可以完成简单的进程调度任务了。

有表示状态的 **state**，表示剩余时间片的 **counter**，表示优先级的 **priority**，和表示上下文信息的 **tss**。

其他字段我们需要用到的时候再说，今天只是头脑风暴一下进程调度设计的思路。

我们看一下 Linux 0.11 中进程结构的全部，心里先有个数，具体干嘛的先别管，就记住我们刚刚头脑风暴的那四个字段就行了。

```
struct task_struct {
/* these are hardcoded - don't touch */
    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long signal;
    struct sigaction sigaction[32];
    long blocked;   /* bitmap of masked signals */
/* various fields */
    int exit_code;
    unsigned long start_code,end_code,end_data,brk,start_stack;
    long pid,father,pgrp,session,leader;
    unsigned short uid,euid,suid;
    unsigned short gid,egid,sgid;
    long alarm;
    long utime,stime,cutime,cstime,start_time;
    unsigned short used_math;
/* file system info */
    int tty;        /* -1 if no tty, so it must be signed */
    unsigned short umask;
    struct m_inode * pwd;
    struct m_inode * root;
    struct m_inode * executable;
    unsigned long close_on_exec;
    struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
    struct desc_struct ldt[3];
/* tss for this task */
    struct tss_struct tss;
};
```

看吧，其实也没多少咯～

好了，今天我们完全由自己从零到有设计出了进程调度的大体流程，以及它需要的数据结构。

我们知道了进程调度的开始，要从一次定时器滴答来触发，通过时钟中断处理函数走到进程调度函数，然后去进程的结构 task_struct 中取出所需的数据，进行策略计算，并挑选出下一个可以得到 CPU 运行的进程，跳转过去。

那么下一讲，我们从一次时钟中断出发，看看一次 Linux 0.11 的进程调度的全过程。有了这两回做铺垫，之后再看主流程中的 fork 代码，将会非常清晰！

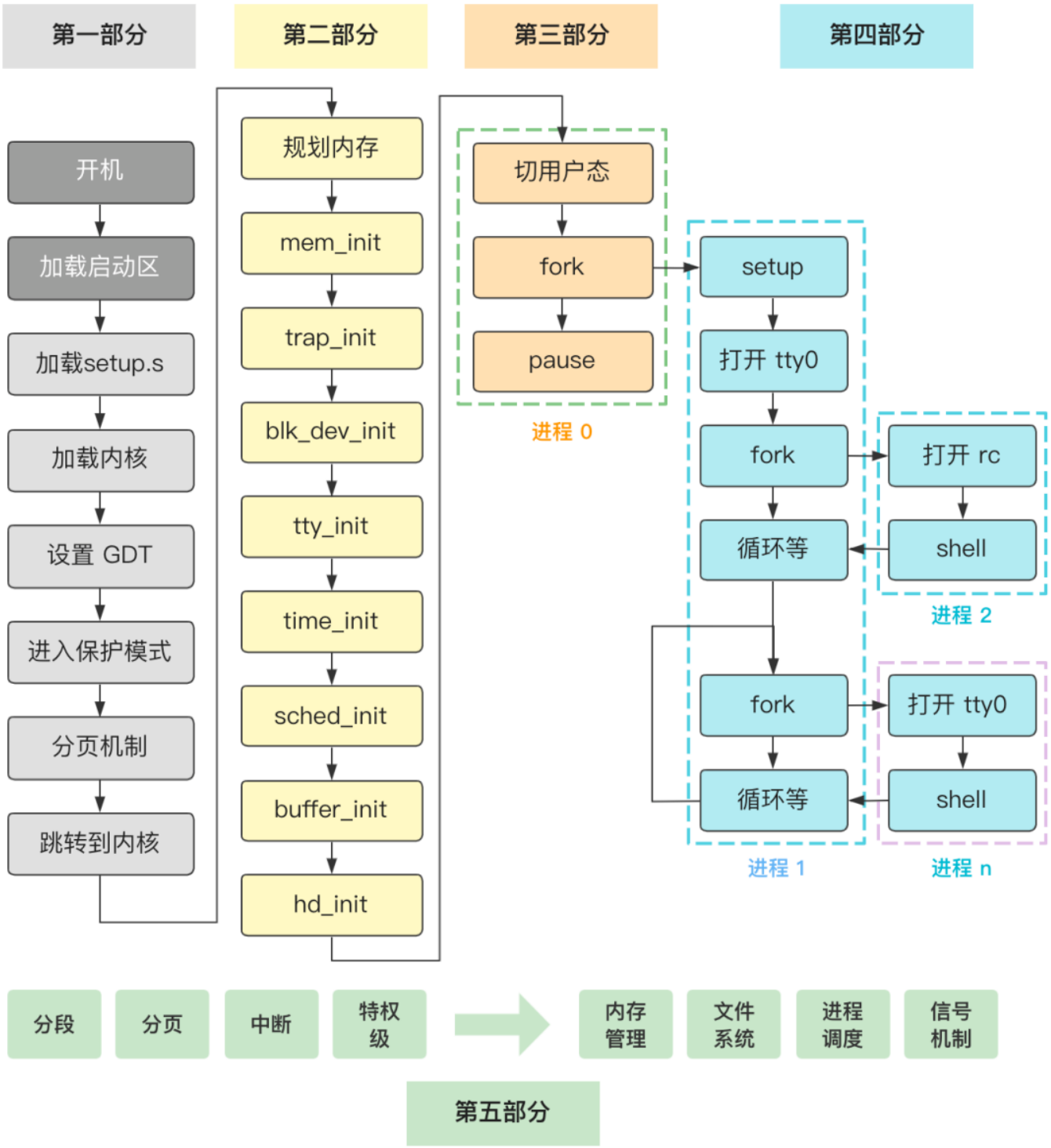
欲知后事如何，且听下回分解。

------- 关于本系列的完整内容 -------

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**在看**我也会很开心，我相信星火燎原的力量，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
一个新进程的诞生（二）从内核态到用户态

下一篇
一个新进程的诞生（四）从一次定时器滴答来看进程调度

Modified on 2022-02-16

Read more

People who liked this content also liked

一个逻辑完备的线程池
程序喵大人

13600个字，给你解释清楚 JVM对象销毁
爱穿格子衫的程序猿

基于线程池的线上服务性能优化
高性能架构探索