一个新进程的诞生（四）从一次定时器滴答来看进程调度

Original 闪客 低并发编程 2022-02-23 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，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。

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

这一讲，我们从一次定时器滴答出发，看看一次 Linux 0.11 的进程调度的全过程。

Let's Go!

还记得我们在 第18回 | 大名鼎鼎的进程调度就是从这里开始的 **sched_init** 的时候，开启了**定时器**吧？这个定时器每隔一段时间就会向 CPU 发起一个中断信号。

这个间隔时间被设置为 10 ms，也就是 100 Hz。

schedule.c

```
#define HZ 100
```

发起的中断叫**时钟中断**，其中断向量号被设置为了 **0x20**。

还记得我们在 **sched_init** 里设置的时钟中断和对应的中断处理函数吧？

schedule.c

```
set_intr_gate(0x20, &timer_interrupt);
```

这样，当时钟中断，也就是 0x20 号中断来临时，CPU 会查找中断向量表中 0x20 处的函数地址，即中断处理函数，并跳转过去执行。

这个中断处理函数就是 **timer_interrupt**，是用汇编语言写的。

system_call.s

```
_timer_interrupt:
    ...
    // 增加系统滴答数
    incl _jiffies
    ...
    // 调用函数 do_timer
    call _do_timer
    ...
```

这个函数做了两件事，一个是将**系统滴答数**这个变量 **jiffies** 加一，一个是调用了另一个函数 **do_timer**。

sched.c

```
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();
}
```

do_timer 最重要的部分就是上面这段代码，非常简单。

首先将当先进程的时间片 -1，然后判断：

如果时间片仍然大于零，则什么都不做直接返回。

如果时间片已经为零，则调用 schedule()，很明显，这就是进行进程调度的主干。

```
void schedule(void) {
    int i, next, c;
    struct task_struct ** p;
    ...
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
}
```

别看这么一大坨，我做个不严谨的简化，你就明白了

```
void schedule(void) {
    int next = get_max_counter_and_runnable_thread();
    refresh_all_thread_counter();
    switch_to(next);
}
```

看到没，就剩这么点了。

很简答，这个函数就做了三件事：

1. 拿到剩余时间片（counter的值）最大且在 runnable 状态（state = 0）的进程号 next。

2. 如果所有 runnable 进程时间片都为 0，则将所有进程（注意不仅仅是 runnable 的进程）的 counter 重新赋值（counter = counter/2 + priority），然后再次执行步骤 1。

3. 最后拿到了一个进程号 next，调用了 switch_to(next) 这个方法，就切换到了这个进程去执行了。

看 switch_to 方法，是用内联汇编语句写的。

sched.h

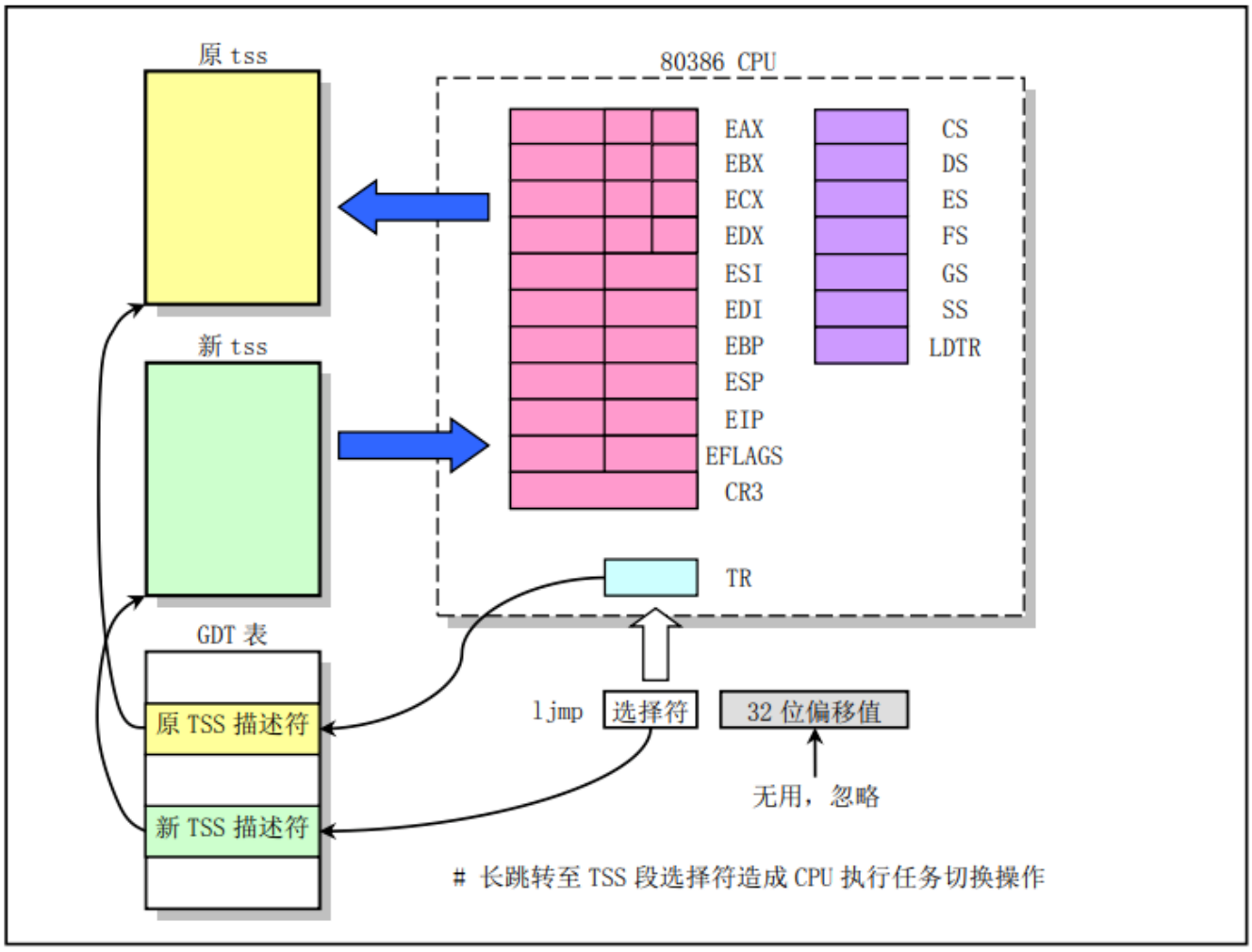
```
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
    "movw %%dx,%1\n\t" \
    "xchgl %%ecx,_current\n\t" \
    "ljmp %0\n\t" \
    "cmpl %%ecx,_last_task_used_math\n\t" \
    "jne 1f\n\t" \
    "clts\n" \
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

这段话就是进程切换的最最最最底层的代码了。

看不懂没关系，其实主要就干了一件事，就是 ljmp 到新进程的 tss 段处。

啥意思?

CPU 规定，如果 ljmp 指令后面跟的是一个 tss 段，那么，会由硬件将当前各个寄存器的值保存在当前进程的 tss 中，并将新进程的 tss 信息加载到各个寄存器。



上图来源于《Linux内核完全注释V5.0》

这个图在完全注释这本书里里画的非常清晰，我就不重复造轮子了。

简单说就是，**保存当前进程上下文，恢复下一个进程的上下文，跳过去**!

看，不知不觉，我们上一讲和本讲开头提到的那些进程数据结构的字段，就都用上了。

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

至此，我们梳理完了一个进程切换的整条链路，来回顾一下。

----- 流水账开始 -----

罪魁祸首的，就是那个每 10ms 触发一次的定时器滴答。

而这个滴答将会给 CPU 产生一个时钟中断信号。

而这个中断信号会使 CPU 查找中断向量表，找到操作系统写好的一个时钟中断处理函数 do_timer。

do_timer 会首先将当前进程的 counter 变量 -1，如果 counter 此时仍然大于 0，则就此结束。

但如果 counter = 0 了，就开始进行进程的调度。

进程调度就是找到所有处于 RUNNABLE 状态的进程，并找到一个 counter 值最大的进程，把它丢进 switch_to 函数的入参里。

switch_to 这个终极函数，会保存当前进程上下文，恢复要跳转到的这个进程的上下文，同时使得 CPU 跳转到这个进程的偏移地址处。

接着，这个进程就舒舒服服地运行了起来，等待着下一次时钟中断的来临。

----- 流水账结束 -----

好了，这两回我们自己设计了一遍进程调度，又看了一次 Linux 0.11 的进程调度的全过程。有了这两回做铺垫，我们下一回就该非常自信地回到我们的主流程，开始看我们心心念念的 **fork** 函数！

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

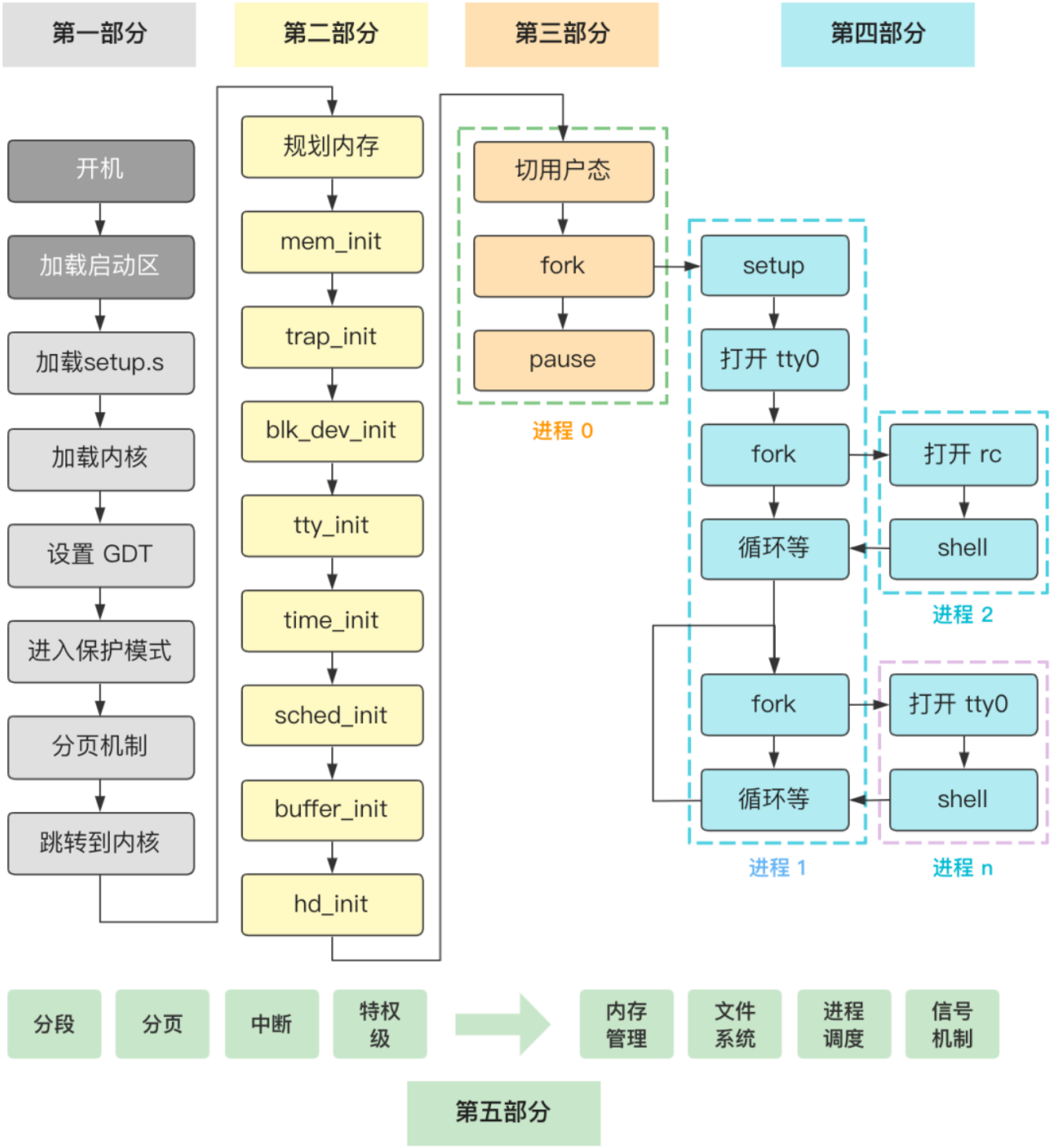
欲知后事如何，且听下回分解。

------- 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（三）如果让你来设计进程调度

下一篇
让我们一起来写本书？

Read more

People who liked this content also liked

单片机的程序结束后都干嘛去了？
唐敏技能大师工作室

time包的单调时钟处理
Golang菜鸟

单片机“面向对象”
IoT Inn