

17 | 调度（下）：抢占式调度是如何发生的？

2019-05-06 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 08:50 大小 8.10M




上一节，我们讲了主动调度，就是进程运行到一半，因为等待 I/O 等操作而主动让出 CPU，然后就进入了我们的“进程调度第一定律”。所有进程的调用最终都会走 `_schedule` 函数。那这个定律在这一节还是要继续起作用。

抢占式调度

上一节我们讲的主动调度是第一种方式，第二种方式，就是抢占式调度。什么情况下会发生抢占呢？

最常见的现象就是一个进程执行时间太长了，是时候切换到另一个进程了。那怎么衡量一个进程的运行时间呢？在计算机里面有一个时钟，会过一段时间触发一次时钟中断，通知操作系统，时间又过去一个时钟周期，这是个很好的方式，可以查看是否需要抢占的时间点。


时钟中断处理函数会调用 `scheduler_tick()`，它的代码如下：

 复制代码

```
1 void scheduler_tick(void)
2 {
3     int cpu = smp_processor_id();
4     struct rq *rq = cpu_rq(cpu);
5     struct task_struct *curr = rq->curr;
6     .....
7     curr->sched_class->task_tick(rq, curr, 0);
8     cpu_load_update_active(rq);
9     calc_global_load_tick(rq);
10    .....
11 }
```

这个函数先取出当前 `cpu` 的运行队列，然后得到这个队列上当前正在运行中的进程的 `task_struct`，然后调用这个 `task_struct` 的调度类的 `task_tick` 函数，顾名思义这个函数就是来处理时钟事件的。

如果当前运行的进程是普通进程，调度类为 `fair_sched_class`，调用的处理时钟的函数为 `task_tick_fair`。我们来看一下它的实现。

 复制代码

```
1 static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
2 {
3     struct cfs_rq *cfs_rq;
4     struct sched_entity *se = &curr->se;
5
6
7     for_each_sched_entity(se) {
8         cfs_rq = cfs_rq_of(se);
9         entity_tick(cfs_rq, se, queued);
10    }
11    .....
12 }
```

根据当前进程的 `task_struct`，找到对应的调度实体 `sched_entity` 和 `cfs_rq` 队列，调用 `entity_tick`。

```

1 static void
2 entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
3 {
4     update_curr(cfs_rq);
5     update_load_avg(curr, UPDATE_TG);
6     update_cfs_shares(curr);
7     .....
8     if (cfs_rq->nr_running > 1)
9         check_preempt_tick(cfs_rq, curr);
10 }

```

在 `entity_tick` 里面，我们又见到了熟悉的 `update_curr`。它会更新当前进程的 `vruntime`，然后调用 `check_preempt_tick`。顾名思义就是，检查是否是时候被抢占了。

```

1 static void
2 check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
3 {
4     unsigned long ideal_runtime, delta_exec;
5     struct sched_entity *se;
6     s64 delta;
7
8
9     ideal_runtime = sched_slice(cfs_rq, curr);
10    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
11    if (delta_exec > ideal_runtime) {
12        resched_curr(rq_of(cfs_rq));
13        return;
14    }
15    .....
16    se = __pick_first_entity(cfs_rq);
17    delta = curr->vruntime - se->vruntime;
18    if (delta < 0)
19        return;
20    if (delta > ideal_runtime)
21        resched_curr(rq_of(cfs_rq));
22 }

```


`check_preempt_tick` 先是调用 `sched_slice` 函数计算出的 `ideal_runtime`，他是一个调度周期中，这个进程应该运行的实际时间。

sum_exec_runtime 指进程总共执行的实际时间，prev_sum_exec_runtime 指上次该进程被调度时已经占用的实际时间。每次在调度一个新的进程时都会把它的 se->prev_sum_exec_runtime = se->sum_exec_runtime，所以 sum_exec_runtime - prev_sum_exec_runtime 就是这次调度占用实际时间。如果这个时间大于 ideal_runtime，则应该被抢占了。

除了这个条件之外，还会通过 __pick_first_entity 取出红黑树中最小的进程。如果当前进程的 vruntime 大于红黑树中最小的进程的 vruntime，且差值大于 ideal_runtime，也应该被抢占了。

当发现当前进程应该被抢占，不能直接把它踢下来，而是把它标记为应该被抢占。为什么呢？因为进程调度第一定律呀，一定要等待正在运行的进程调用 __schedule 才行啊，所以这里只能先标记一下。


标记一个进程应该被抢占，都是调用 resched_curr，它会调用 set_tsk_need_resched，标记进程应该被抢占，但是此时此刻，并不真的抢占，而是打上一个标签 TIF_NEED_RESCHED。

 复制代码

```
1 static inline void set_tsk_need_resched(struct task_struct *tsk)
2 {
3     set_tsk_thread_flag(tsk, TIF_NEED_RESCHED);
4 }
```

另外一个可能抢占的场景是**当一个进程被唤醒的时候**。

我们前面说过，当一个进程在等待一个 I/O 的时候，会主动放弃 CPU。但是当 I/O 到来的时候，进程往往会被唤醒。这个时候是一个时机。当被唤醒的进程优先级高于 CPU 上的当前进程，就会触发抢占。try_to_wake_up() 调用 ttwu_queue 将这个唤醒的任务添加到队列当中。ttwu_queue 再调用 ttwu_do_activate 激活这个任务。ttwu_do_activate 调用 ttwu_do_wakeup。这里面调用了 check_preempt_curr 检查是否应该发生抢占。如果应该发生抢占，也不是直接踢走当前进程，而也是将当前进程标记为应该被抢占。

 复制代码

```
1 static void ttwu_do_wakeup(struct rq *rq, struct task_struct *p, int wake_flags,
```

```
2             struct rq_flags *rf)
3 {
4     check_preempt_curr(rq, p, wake_flags);
5     p->state = TASK_RUNNING;
6     trace_sched_wakeup(p);
```

到这里，你会发现，抢占问题只做完了一半。就是标识当前运行中的进程应该被抢占了，但是真正的抢占动作并没有发生。

抢占的时机


真正的抢占还需要时机，也就是需要那么一个时刻，让正在运行中的进程有机会调用一下 `__schedule`。

你可以想象，不可能某个进程代码运行着，突然要去调用 `__schedule`，代码里面不可能这么写，所以一定要规划几个时机，这个时机分为用户态和内核态。

用户态的抢占时机

对于用户态的进程来讲，从系统调用中返回的那个时刻，是一个被抢占的时机。

前面讲系统调用的时候，64 位的系统调用的链路位 `do_syscall_64->syscall_return_slowpath->prepare_exit_to_usermode->exit_to_usermode_loop`，当时我们还没关注 `exit_to_usermode_loop` 这个函数，现在我们来看一下。


 复制代码

```
1 static void exit_to_usermode_loop(struct pt_regs *regs, u32 cached_flags)
2 {
3     while (true) {
4         /* We have work to do. */
5         local_irq_enable();
6
7
8         if (cached_flags & _TIF_NEED_RESCHED)
9             schedule();
10     .....
11     }
12 }
```

现在我们看到在 `exit_to_usermode_loop` 函数中，上面打的标记起了作用，如果被打了 `_TIF_NEED_RESCHED`，调用 `schedule` 进行调度，调用的过程和上一节解析的一样，会选择一个进程让出 CPU，做上下文切换。

对于用户态的进程来讲，从中断中返回的那个时刻，也是一个被抢占的时机。

在 `arch/x86/entry/entry_64.S` 中有中断的处理过程。又是一段汇编语言代码，你重点领会它的意思就行，不要纠结每一行都看懂。

 复制代码

```
1 common_interrupt:
2     ASM_CLAC
3     addq    $-0x80, (%rsp)
4     interrupt do_IRQ
5 ret_from_intr:
6     popq    %rsp
7     testb   $3, CS(%rsp)
8     jz      retint_kernel
9 /* Interrupt came from user space */
10 GLOBAL(retint_user)
11     mov     %rsp,%rdi
12     call    prepare_exit_to_usermode
13     TRACE_IRQS_IRETQ
14     SWAPGS
15     jmp     restore_regs_and_iret
16 /* Returning to kernel space */
17 retint_kernel:
18 #ifdef CONFIG_PREEMPT
19     bt      $9, EFLAGS(%rsp)
20     jnc     1f
21 0:      cmpl $0, PER_CPU_VAR(__preempt_count)
22     jnz     1f
23     call    preempt_schedule_irq
24     jmp     0b
```

中断处理调用的是 `do_IRQ` 函数，中断完毕后分为两种情况，一个是返回用户态，一个是返回内核态。这个通过注释也能看出来。

咱们先来看看返回用户态这一部分，先不管返回内核态的那部分代码，`retint_user` 会调用 `prepare_exit_to_usermode`，最终调用 `exit_to_usermode_loop`，和上面的逻辑一样，发现有标记则调用 `schedule()`。

内核态的抢占时机

用户态的抢占时机讲完了，接下来我们看内核态的抢占时机。

对内核态的执行中，被抢占的时机一般发生在在 `preempt_enable()` 中。

在内核态的执行中，有的操作是不能被中断的，所以在进行这些操作之前，总是先调用 `preempt_disable()` 关闭抢占，当再次打开的时候，就是一次内核态代码被抢占的机会。

就像下面代码中展示的一样，`preempt_enable()` 会调用 `preempt_count_dec_and_test()`，判断 `preempt_count` 和 `TIF_NEED_RESCHED` 看是否可以被抢占。如果可以，就调用 `preempt_schedule->preempt_schedule_common->__schedule` 进行调度。还是满足进程调度第一定律的。

 复制代码

```
1 #define preempt_enable() \
2 do { \
3     if (unlikely(preempt_count_dec_and_test())) \
4         __preempt_schedule(); \
5 } while (0)
6
7
8 #define preempt_count_dec_and_test() \
9     ({ preempt_count_sub(1); should_resched(0); })
10
11
12 static __always_inline bool should_resched(int preempt_offset)
13 {
14     return unlikely(preempt_count() == preempt_offset &&
15                    tif_need_resched());
16 }
17
18
19 #define tif_need_resched() test_thread_flag(TIF_NEED_RESCHED)
20
21
22 static void __sched notrace preempt_schedule_common(void)
23 {
24     do {
25         .....
26         __schedule(true);
27         .....
28     } while (need_resched())
```

在内核态也会遇到中断的情况，当中断返回的时候，返回的仍然是内核态。这个时候也是一个执行抢占的时机，现在我们来上面中断返回的代码中返回内核的那部分代码，调用的是 `preempt_schedule_irq`。

 复制代码

```
1  asmlinkage __visible void __sched preempt_schedule_irq(void)
2  {
3      .....
4      do {
5          preempt_disable();
6          local_irq_enable();
7          __schedule(true);
8          local_irq_disable();
9          sched_preempt_enable_no_resched();
10     } while (need_resched());
11     .....
12 }
```

`preempt_schedule_irq` 调用 `__schedule` 进行调度。还是满足进程调度第一定律的。

总结时刻

好了，抢占式调度就讲到这里了。我这里画了一张脑图，将整个进程的调度体系都放在里面。

这个脑图里面第一条就是总结了进程调度第一定律的核心函数 `__schedule` 的执行过程，这是上一节讲的，因为要切换的东西比较多，需要你详细了解每一部分是如何切换的。

第二条总结了标记为可抢占的场景，第三条是所有的抢占发生的时机，这里是真正验证了进程调度第一定律的。



课堂练习

通过对于内核中进程调度的分析，我们知道，时间对于调度是很重要的，你知道 Linux 内核是如何管理和度量时间的吗？

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，**反复研读**。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。



趣谈 Linux 操作系统


像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

上一篇 16 | 调度（中）：主动调度是如何发生的？

下一篇 18 | 进程的创建：如何发起一个新项目？

精选留言 (10)

写留言



why

2019-05-10

6

- 抢占式调度
- 两种情况: 执行太久, 需切换到另一进程; 另一个高优先级进程被唤醒
 - 执行太久: 由时钟中断触发检测, 中断处理调用 scheduler_tick
 - 取当前进程 task_struct->task_tick_fair()->取 sched_entity cfs_rq 调用 entity_tick()...

展开



杨怀

2019-05-06

4

老师您好，我喜欢边调试边阅读代码，代码是死的但是跑起来是活的变的，linux内核代码有没有好的调试方式，或者添加打印日志的方式；另外时钟中断是怎么触发的呢，我记得cpu里面没有时钟这个物理设备的，应该有类似单片机晶振这个东西去无限循环执行指令的，这个也不会有时钟中断呀



CHEN

2019-05-06

2

Linux内核通过时钟中断管理和度量时间.

Linux在初始化时会使用一个init_irq()函数设定定时周期(IRQ:Interrupt Request),

time_init()中调用setup_irq()设置时间中断向量irq 0; 中断服务程序是

timer_interrupt(), 会调用另一个函数do_timer_interrupt(),do_timer_interrupt还会调用do_timer更新系统时间。do_timer中的工作包括，让全局变量jiffies增加1，并且调用...

展开



焰火

2019-05-07

1

进程调度第一定律总结的太棒了。

另外有个问题想问下老师：我把整个调度系统想成一个进程，这个调度进程来实现task调度？如果是这样的，Linux如果跑在单CPU上，多进程是怎么调度的呢？

展开 ∨



兴文

2019-05-30



如果用户进程一直在用户态执行，没有发生系统调用和中断，就不会触发scheduler操作，那这个进程是不是一直占有CPU啊？

不胖二十斤
绝不换头像

如是

2019-05-24



老师中断是怎么处理的，难道不会用到cpu吗？

展开 ∨

作者回复: 会用cpu的



周平

2019-05-16



管理的时间或者说度量的时间是否就是系统时钟，就像MCU中的时钟源一样呢？



wwj

2019-05-15



物理内存统一管理 本身也是程序 他的内存如何管理

展开 ∨

作者回复: 物理内存的管理程序也是程序，也分代码部分和数据部分，代码部分当然在内核代码段里面了，系统启动的时候就加载了。数据部分大部分分配在直接映射区，也会分配页表，页表在哪里呢？页表的根在代码段的那个区域里面。



tiankonghe...

2019-05-06



学习了

展开 ▾



安排

2019-05-06



task_struct中有指向调度类的指针，第15课调度（上）还有疑问不知道这个指针有什么用，在这一节找到了答案。

老师讲的比那些内核书上讲的好太多了。

作者回复: 这样夸奖

