

## 15 | 调度（上）：如何制定项目管理流程？

2019-05-01 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 17:33 大小 16.08M



前几节，我们介绍了 `task_struct` 数据结构。它就像项目管理系统一样，可以帮项目经理维护项目运行过程中的各类信息，但这并不意味着项目管理工作就完事大吉了。`task_struct` 仅仅能够解决“看到”的问题，咱们还要解决如何制定流程，进行项目调度的问题，也就是“做到”的问题。

公司的人员总是有限的。无论接了多少项目，公司不可能短时间增加很多人手。有的项目比较紧急，应该先进行排期；有的项目可以缓缓，但是也不能让客户等太久。所以这个过程非常复杂，需要平衡。

对于操作系统来讲，它面对的 CPU 的数量是有限的，干活儿都是它们，但是进程数目远远超过 CPU 的数目，因而就需要进行进程的调度，有效地分配 CPU 的时间，既要保证进程的最快响应，也要保证进程之间的公平。这也是一个非常复杂的、需要平衡的事情。

## 调度策略与调度类


在 Linux 里面，进程大概可以分成两种。

一种称为**实时进程**，也就是需要尽快执行返回结果的那种。这就好比我们是一家公司，接到的客户项目需求就会有很多种。有些客户的项目需求比较急，比如一定要在一两个月内完成的这种，客户会加急加钱，那这种客户的优先级就会比较高。

另一种是**普通进程**，大部分的进程其实都是这种。这就好比，大部分客户的项目都是普通的需求，可以按照正常流程完成，优先级就没实时进程这么高，但是人家肯定也有确定的交付日期。


那很显然，对于这两种进程，我们的调度策略肯定是不同的。

在 `task_struct` 中，有一个成员变量，我们叫**调度策略**。

 复制代码


```
1 unsigned int policy;
```

它有以下几个定义：

 复制代码

```
1 #define SCHED_NORMAL      0
2 #define SCHED_FIFO        1
3 #define SCHED_RR          2
4 #define SCHED_BATCH        3
5 #define SCHED_IDLE         5
6 #define SCHED_DEADLINE     6
```

配合调度策略的，还有我们刚才说的**优先级**，也在 `task_struct` 中。

 复制代码

```
1 int prio, static_prio, normal_prio;
2 unsigned int rt_priority;
```

优先级其实就是一个数值，对于实时进程，优先级的范围是 0 ~ 99；对于普通进程，优先级的范围是 100 ~ 139。数值越小，优先级越高。从这里可以看出，所有的实时进程都比普通进程优先级要高。毕竟，谁让人家加钱了呢。

## 实时调度策略

对于调度策略，其中 `SCHED_FIFO`、`SCHED_RR`、`SCHED_DEADLINE` 是实时进程的调度策略。

虽然大家都是加钱加急的项目，但是也不能乱来，还是需要有个办事流程才行。

例如，**`SCHED_FIFO`**就是交了相同钱的，先来先服务，但是有的加钱多，可以分配更高的优先级，也就是说，高优先级的进程可以抢占低优先级的进程，而相同优先级的进程，我们遵循先来先得。

另外一种策略是，交了相同钱的，轮换着来，这就是**`SCHED_RR` 轮流调度算法**，采用时间片，相同优先级的任务当用完时间片会被放到队列尾部，以保证公平性，而高优先级的任务也是可以抢占低优先级的任务。

还有一种新的策略是**`SCHED_DEADLINE`**，是按照任务的 `deadline` 进行调度的。当产生一个调度点的时候，DL 调度器总是选择其 `deadline` 距离当前时间点最近的那个任务，并调度它执行。

## 普通调度策略

对于普通进程的调度策略有，`SCHED_NORMAL`、`SCHED_BATCH`、`SCHED_IDLE`。


既然大家的项目都没有那么紧急，就应该按照普通的项目流程，公平地分配人员。

`SCHED_NORMAL` 是普通的进程，就相当于咱们公司接的普通项目。

`SCHED_BATCH` 是后台进程，几乎不需要和前端进行交互。这有点像公司在接项目同时，开发一些可以复用的模块，作为公司的技术积累，从而使得在之后接新项目的时候，能够减少工作量。这类项目可以默默执行，不要影响需要交互的进程，可以降低他的优先级。

SCHED\_IDLE 是特别空闲的时候才跑的进程，相当于咱们学习训练类的项目，比如咱们公司很长时间没有接到外在项目了，可以弄几个这样的项目练练手。

上面无论是 policy 还是 priority，都设置了一个变量，变量仅仅表示了应该这样这样干，但事情总要有有人去干，谁呢？在 task\_struct 里面，还有这样的成员变量：

 复制代码

```
1 const struct sched_class *sched_class;
```

调度策略的执行逻辑，就封装在这里面，它是真正干活的那个。

sched\_class 有几种实现：

stop\_sched\_class 优先级最高的任务会使用这种策略，会中断所有其他线程，且不会被其他任务打断；

dl\_sched\_class 就对应上面的 deadline 调度策略；

rt\_sched\_class 就对应 RR 算法或者 FIFO 算法的调度策略，具体调度策略由进程的 task\_struct->policy 指定；

fair\_sched\_class 就是普通进程的调度策略；

idle\_sched\_class 就是空闲进程的调度策略。

这里实时进程的调度策略 RR 和 FIFO 相对简单一些，而且由于咱们平时常遇到的都是普通进程，在这里，咱们就重点分析普通进程的调度问题。普通进程使用的调度策略是 fair\_sched\_class，顾名思义，对于普通进程来讲，公平是最重要的。

## 完全公平调度算法

在 Linux 里面，实现了一个基于 CFS 的调度算法。CFS 全称 Completely Fair Scheduling，叫完全公平调度。听起来很“公平”。那这个算法的原理是什么呢？我们来看看。

首先，你需要记录下进程的运行时间。CPU 会提供一个时钟，过一段时间就触发一个时钟中断。就像咱们的表滴答一下，这个我们叫 Tick。CFS 会为每一个进程安排一个虚拟运行

时间 `vruntime`。如果一个进程在运行，随着时间的增长，也就是一个个 `tick` 的到来，进程的 `vruntime` 将不断增大。没有得到执行的进程 `vruntime` 不变。


显然，那些 `vruntime` 少的，原来受到了不公平的对待，需要给它补上，所以会优先运行这样的进程。

这有点像让你把一筐球平均分到  $N$  个口袋里面，你看着哪个少，就多放一些；哪个多了，就先不放。这样经过多轮，虽然不能保证球完全一样多，但是也差不多公平。

你可能会说，不还有优先级呢？如何给优先级高的进程多分时间呢？

这个简单，就相当于  $N$  个口袋，优先级高的袋子大，优先级低的袋子小。这样球就不能按照个数分配了，要按照比例来，大口袋的放了一半和小口袋放了一半，里面的球数目虽然差很多，也认为是公平的。

在更新进程运行的统计量的时候，我们其实就可以看出这个逻辑。

 复制代码

```
1  /*
2   * Update the current task's runtime statistics.
3   */
4  static void update_curr(struct cfs_rq *cfs_rq)
5  {
6      struct sched_entity *curr = cfs_rq->curr;
7      u64 now = rq_clock_task(rq_of(cfs_rq));
8      u64 delta_exec;
9      .....
10     delta_exec = now - curr->exec_start;
11     .....
12     curr->exec_start = now;
13     .....
14     curr->sum_exec_runtime += delta_exec;
15     .....
16     curr->vruntime += calc_delta_fair(delta_exec, curr);
17     update_min_vruntime(cfs_rq);
18     .....
19 }
20
21
22 /*
23  * delta /= w
24  */
25 static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
```

```
26 {
27     if (unlikely(se->load.weight != NICE_0_LOAD))
28         /* delta_exec * weight / lw.weight */
29         delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
30     return delta;
31 }
```

在这里得到当前的时间，以及这次的时间片开始的时间，两者相减就是这次运行的时间 `delta_exec`，但是得到的这个时间其实是实际运行的时间，需要做一定的转化才作为虚拟运行时间 `vruntime`。转化方法如下：

虚拟运行时间 `vruntime` += 实际运行时间 `delta_exec` \* `NICE_0_LOAD` / 权重

这就是说，同样的实际运行时间，给高权重的算少了，低权重的算多了，但是当选取下一个运行进程的时候，还是按照最小的 `vruntime` 来的，这样高权重的获得的实际运行时间自然就多了。这就相当于给一个体重 (权重) 200 斤的胖子吃两个馒头，和给一个体重 100 斤的瘦子吃一个馒头，然后说，你们两个吃的是一样多。这样虽然总体胖子比瘦子多吃了一倍，但是还是公平的。

## 调度队列与调度实体

看来 CFS 需要一个数据结构来对 `vruntime` 进行排序，找出最小的那个。这个能够排序的数据结构不但需要查询的时候，能够快速找到最小的，更新的时候也需要能够快速调整排序，要知道 `vruntime` 可是经常在变的，变了再插入这个数据结构，就需要重新排序。

能够平衡查询和更新速度的是树，在这里使用的是红黑树。

红黑树的的节点是应该包括 `vruntime` 的，称为调度实体。

在 `task_struct` 中有这样的成员变量：


```
struct sched_entity se;
struct sched_rt_entity rt;
struct sched_dl_entity dl;
```

这里有实时调度实体 `sched_rt_entity`，Deadline 调度实体 `sched_dl_entity`，以及完全公平算法调度实体 `sched_entity`。

看来不光 CFS 调度策略需要有这样一个数据结构进行排序，其他的调度策略也同样有自己的数据结构进行排序，因为任何一个策略做调度的时候，都是要区分谁先运行谁后运行。

而进程根据自己是实时的，还是普通的类型，通过这个成员变量，将自己挂在某一个数据结构里面，和其他的进程排序，等待被调度。如果这个进程是个普通进程，则通过 `sched_entity`，将自己挂在这棵红黑树上。

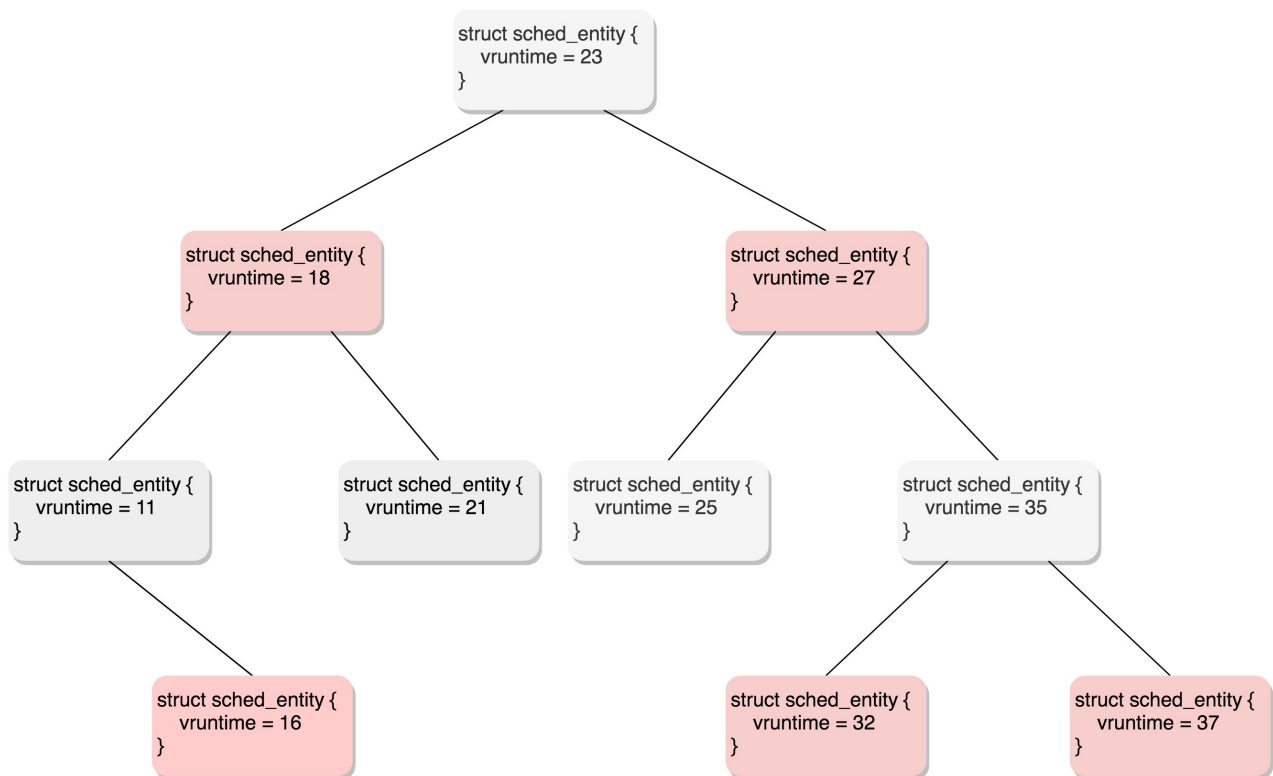
对于普通进程的调度实体定义如下，这里面包含了 `vruntime` 和权重 `load_weight`，以及对于运行时间的统计。

 复制代码

```
1 struct sched_entity {
2     struct load_weight      load;
3     struct rb_node          run_node;
4     struct list_head        group_node;
5     unsigned int            on_rq;
6     u64                     exec_start;
7     u64                     sum_exec_runtime;
8     u64                     vruntime;
9     u64                     prev_sum_exec_runtime;
10    u64                     nr_migrations;
11    struct sched_statistics   statistics;
12    .....
13 };
```

下图是一个红黑树的例子。






所有可运行的进程通过不断地插入操作最终都存储在以时间为顺序的红黑树中，vruntime 最小的在树的左侧，vruntime 最多的在树的右侧。CFS 调度策略会选择红黑树最左边的叶子节点作为下一个将获得 cpu 的任务。

这棵红黑树放在那里呢？就像每个软件工程师写代码的时候，会将任务排成队列，做完一个做下一个。

CPU 也是这样的，每个 CPU 都有自己的 struct rq 结构，其用于描述在此 CPU 上所运行的所有进程，其包括一个实时进程队列 rt\_rq 和一个 CFS 运行队列 cfs\_rq，在调度时，调度器首先会先去实时进程队列找是否有实时进程需要运行，如果没有才会去 CFS 运行队列找是否有进行需要运行。

 复制代码

```
1 struct rq {
2     /* runqueue lock: */
3     raw_spinlock_t lock;
4     unsigned int nr_running;
5     unsigned long cpu_load[CPU_LOAD_IDX_MAX];
6     .....
7     struct load_weight load;
8     unsigned long nr_load_updates;
9     u64 nr_switches;
```




```

10
11
12     struct cfs_rq cfs;
13     struct rt_rq rt;
14     struct dl_rq dl;
15     .....
16     struct task_struct *curr, *idle, *stop;
17     .....
18 };

```

对于普通进程公平队列 cfs\_rq, 定义如下:

 复制代码

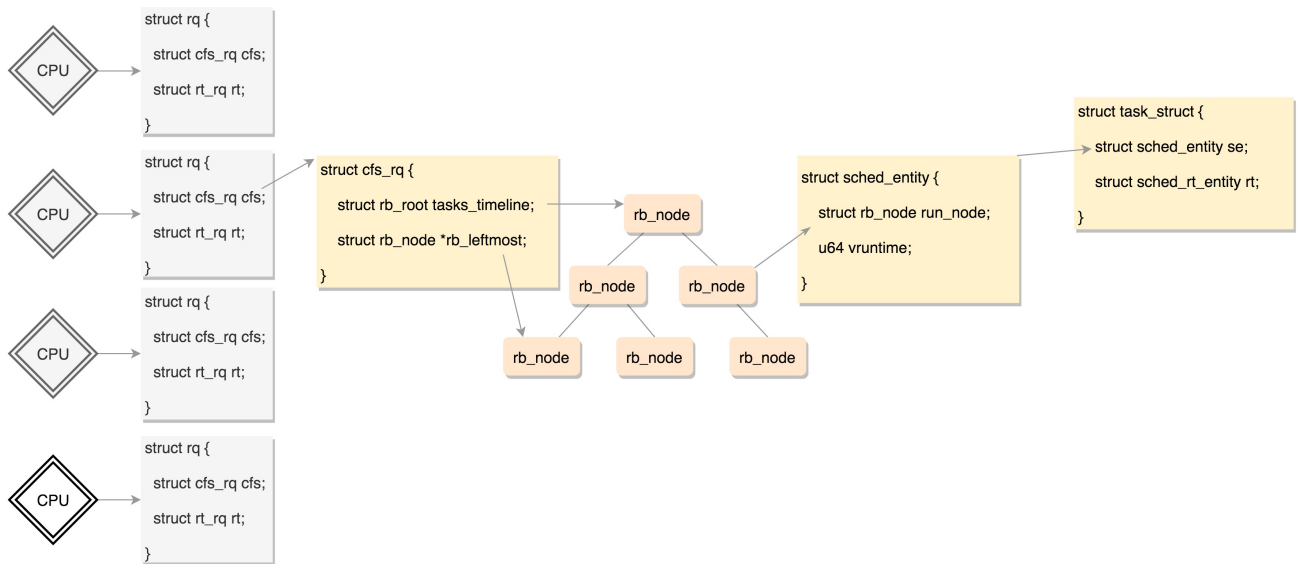
```

1  /* CFS-related fields in a runqueue */
2  struct cfs_rq {
3      struct load_weight load;
4      unsigned int nr_running, h_nr_running;
5
6
7      u64 exec_clock;
8      u64 min_vruntime;
9  #ifndef CONFIG_64BIT
10     u64 min_vruntime_copy;
11 #endif
12     struct rb_root tasks_timeline;
13     struct rb_node *rb_leftmost;
14
15
16     struct sched_entity *curr, *next, *last, *skip;
17     .....
18 };

```

这里面 rb\_root 指向的就是红黑树的根节点, 这个红黑树在 CPU 看起来就是一个队列, 不断的取下一个应该运行的进程。rb\_leftmost 指向的是最左面的节点。

到这里终于凑够数据结构了, 上面这些数据结构的关系如下图:



## 调度类是如何工作的？

凑够了数据结构，接下来我们来看调度类是如何工作的。

调度类的定义如下：

[复制代码](#)

```

1 struct sched_class {
2     const struct sched_class *next;
3
4
5     void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
6     void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
7     void (*yield_task) (struct rq *rq);
8     bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);
9
10
11     void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
12
13
14     struct task_struct * (*pick_next_task) (struct rq *rq,
15                                             struct task_struct *prev,
16                                             struct rq_flags *rf);
17     void (*put_prev_task) (struct rq *rq, struct task_struct *p);
18
19
20     void (*set_curr_task) (struct rq *rq);
21     void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
22     void (*task_fork) (struct task_struct *p);
23     void (*task_dead) (struct task_struct *p);
24
25

```


```

26     void (*switched_from) (struct rq *this_rq, struct task_struct *task);
27     void (*switched_to) (struct rq *this_rq, struct task_struct *task);
28     void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio
29     unsigned int (*get_rr_interval) (struct rq *rq,
30                                     struct task_struct *task);
31     void (*update_curr) (struct rq *rq)

```

这个结构定义了很多种方法，用于在队列上操作任务。这里请大家注意第一个成员变量，是一个指针，指向下一个调度类。

上面我们讲了，调度类分为下面这几种：


 复制代码

```

1 extern const struct sched_class stop_sched_class;
2 extern const struct sched_class dl_sched_class;
3 extern const struct sched_class rt_sched_class;
4 extern const struct sched_class fair_sched_class;
5 extern const struct sched_class idle_sched_class;

```

它们其实是放在一个链表上的。这里我们以调度最常见的操作，**取下一个任务**为例，来解析一下。可以看到，这里面有一个 `for_each_class` 循环，沿着上面的顺序，依次调用每个调度类的方法。


 复制代码

```

1  /*
2   * Pick up the highest-prio task:
3   */
4  static inline struct task_struct *
5  pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
6  {
7      const struct sched_class *class;
8      struct task_struct *p;
9      .....
10     for_each_class(class) {
11         p = class->pick_next_task(rq, prev, rf);
12         if (p) {
13             if (unlikely(p == RETRY_TASK))
14                 goto again;
15             return p;
16         }
17     }

```


这就说明，调度的时候是从优先级最高的调度类到优先级低的调度类，依次执行。而对于每种调度类，有自己的实现，例如，CFS 就有 `fair_sched_class`。

 复制代码

```
1 const struct sched_class fair_sched_class = {
2     .next                = &idle_sched_class,
3     .enqueue_task        = enqueue_task_fair,
4     .dequeue_task        = dequeue_task_fair,
5     .yield_task          = yield_task_fair,
6     .yield_to_task        = yield_to_task_fair,
7     .check_preempt_curr  = check_preempt_wakeup,
8     .pick_next_task       = pick_next_task_fair,
9     .put_prev_task        = put_prev_task_fair,
10    .set_curr_task        = set_curr_task_fair,
11    .task_tick            = task_tick_fair,
12    .task_fork            = task_fork_fair,
13    .prio_changed         = prio_changed_fair,
14    .switched_from        = switched_from_fair,
15    .switched_to          = switched_to_fair,
16    .get_rr_interval      = get_rr_interval_fair,
17    .update_curr          = update_curr_fair,
18 };
```

对于同样的 `pick_next_task` 选取下一个要运行的任务这个动作，不同的调度类有自己的实现。`fair_sched_class` 的实现是 `pick_next_task_fair`，`rt_sched_class` 的实现是 `pick_next_task_rt`。

我们会发现这两个函数是操作不同的队列，`pick_next_task_rt` 操作的是 `rt_rq`，`pick_next_task_fair` 操作的是 `cfs_rq`。

 复制代码

```
1 static struct task_struct *
2 pick_next_task_rt(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
3 {
4     struct task_struct *p;
5     struct rt_rq *rt_rq = &rq->rt;
6     .....
7 }
```

```

8
9
10 static struct task_struct *
11 pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
12 {
13     struct cfs_rq *cfs_rq = &rq->cfs;
14     struct sched_entity *se;
15     struct task_struct *p;
16     .....
17 }

```

这样整个运行的场景就串起来了，在每个 CPU 上都有一个队列 `rq`，这个队列里面包含多个子队列，例如 `rt_rq` 和 `cfs_rq`，不同的队列有不同的实现方式，`cfs_rq` 就是用红黑树实现的。

当有一天，某个 CPU 需要找下一个任务执行的时候，会按照优先级依次调用调度类，不同的调度类操作不同的队列。当然 `rt_sched_class` 先被调用，它会在 `rt_rq` 上找下一个任务，只有找不到的时候，才轮到 `fair_sched_class` 被调用，它会在 `cfs_rq` 上找下一个任务。这样保证了实时任务的优先级永远大于普通任务。

下面我们仔细看一下 `sched_class` 定义的与调度有关的函数。

`enqueue_task` 向就绪队列中添加一个进程，当某个进程进入可运行状态时，调用这个函数；

`dequeue_task` 将一个进程从就绪队列中删除；

`pick_next_task` 选择接下来要运行的进程；

`put_prev_task` 用另一个进程代替当前运行的进程；

`set_curr_task` 用于修改调度策略；

`task_tick` 每次周期性时钟到的时候，这个函数被调用，可能触发调度。

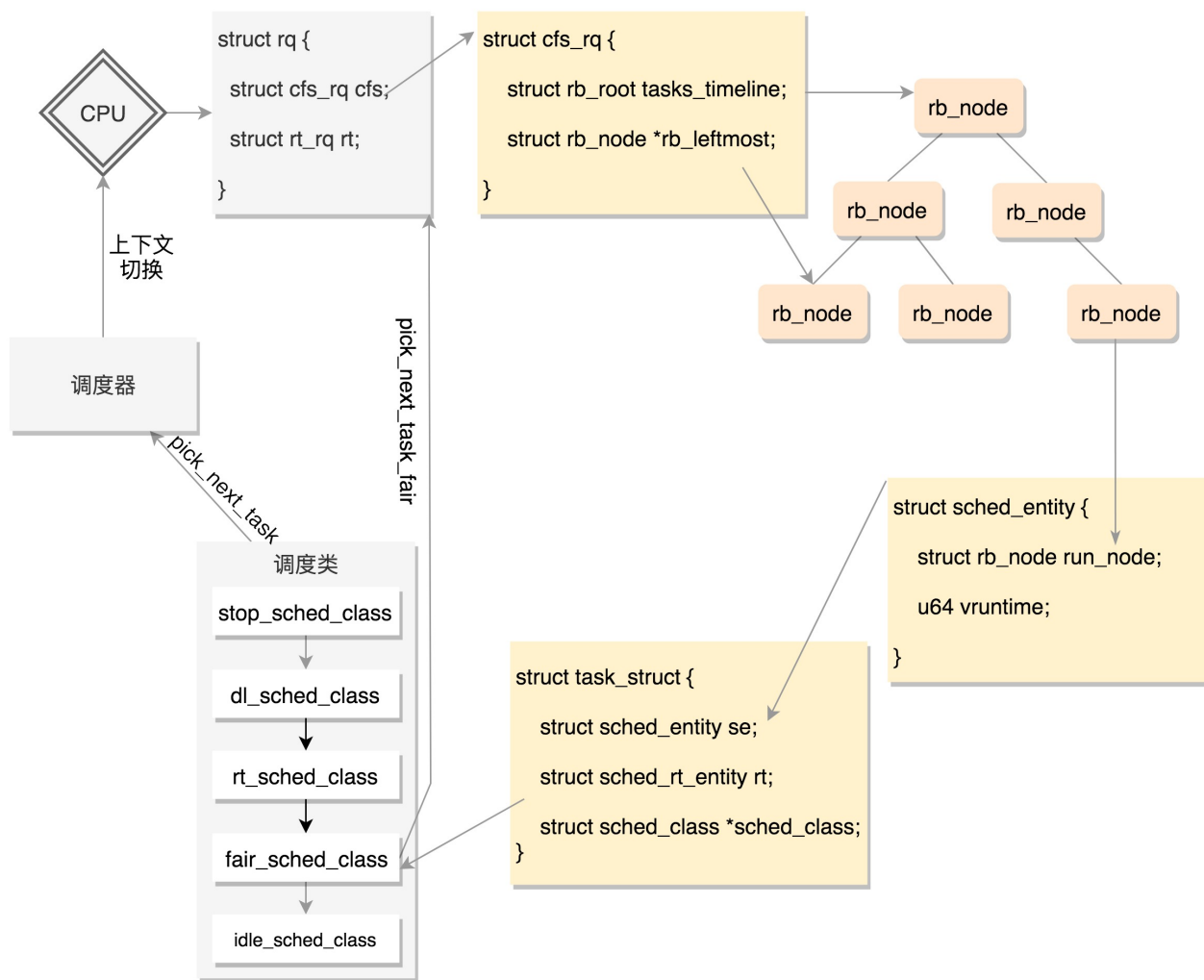
在这里面，我们重点看 `fair_sched_class` 对于 `pick_next_task` 的实现 `pick_next_task_fair`，获取下一个进程。调用路径如下：`pick_next_task_fair->pick_next_entity->__pick_first_entity`。

```
1 struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
2 {
3     struct rb_node *left = rb_first_cached(&cfs_rq->tasks_timeline);
4
5
6     if (!left)
7         return NULL;
8
9
10    return rb_entry(left, struct sched_entity, run_node);
```

从这个函数的实现可以看出，就是从红黑树里面取最左面的节点。

## 总结时刻

好了，这一节我们讲了调度相关的数据结构，还是比较复杂的。一个 CPU 上有一个队列，CFS 的队列是一棵红黑树，树的每一个节点都是一个 sched\_entity，每个 sched\_entity 都属于一个 task\_struct，task\_struct 里面有指针指向这个进程属于哪个调度类。



在调度的时候，依次调用调度类的函数，从 CPU 的队列中取出下一个进程。上面图中的调度器、上下文切换这一节我们没有讲，下一节我们讲讲基于这些数据结构，如何实现调度。

## 课堂练习

这里讲了进程调度的策略和算法，你知道如何通过 API 设置进程和线程的调度策略吗？你可以写个程序尝试一下。

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。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 进程数据结构（下）：项目多了就需要项目管理系统

下一篇 16 | 调度（中）：主动调度是如何发生的？

## 精选留言 (21)

写留言



why

2019-05-01

16

- 调度策略与调度类

- 进程包括两类：实时进程(优先级高); 普通进程
- 两种进程调度策略不同: task\_struct->policy 指明采用哪种调度策略(有6种策略)
- 优先级配合调度策略, 实时进程(0-99); 普通进程(100-139)
- 实时调度策略, 高优先级可抢占低优先级进程...

展开 ▾



Keep-Movi...

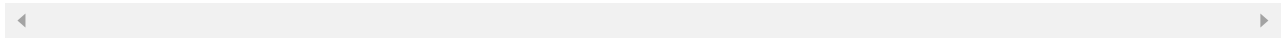
2019-05-01

4

本节讲的是进程的调度，那线程的调度是什么样的呢？Linux调度的基本单位是进程还是线

程呢？

作者回复: 进程和线程都是task，一起调度



青石

2019-05-05

👍 3

# 查看当前进程的调度策略

\$ chrt -p 31636

pid 31636 的当前调度策略: SCHED\_OTHER

pid 31636 的当前调度优先级: 0

...

展开 ▾



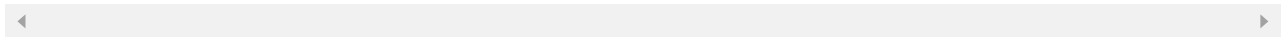
why

2019-05-01

👍 3

如果是新建的进程如何处理, 它 vruntime 总是最小的, 总被调度直到与其他进程相当.

作者回复: 每次新进程创建完毕后，都会试图先让新的抢占一次



叫啥好捏

2019-05-01

👍 2

大佬假期也不休息么

展开 ▾



杨领well

2019-05-22

👍 1

建议:

1、源码在第一行标记出具体出处 (某某文件某某行)

2、文中出现的源码尽量将其标签 (如: again) 的位置标记出来。

展开 ▾





拾贝壳的孩...

2019-05-12



1

如果优先队列一直有任务,普通队列的task一直得不到处理,操作系统会怎么做呢?

我现在知道这个问题的答案了。这种现象叫做饿死。我当时想到这个问题时其实不知道这个概念。

为了防止这种现象的发生,操作系统在一定的周期会重置所有task的优先级,这样就...  
展开 ▾

作者回复: 对的, 饿死, cpu一直转, 低优先级的没响应



Helios

2019-05-07



1

“每次新进程创建完毕后, 都会试图先让新的抢占一次”。

那如果不断有新的任务进来, 那岂不是对以前的任务不公平了呢, 尤其是那些快运行完(可能就差几个时间片了)的任务。



焰火

2019-05-02



1

一个task 分配给一个cpu执行后, 就不会再被其他cpu 执行了吧?

展开 ▾

作者回复: 是的, 同一个时刻同一个cpu只能给一个进程用



刘強

2019-05-02



1

感觉这个sched\_class结构体类似面向对象中的基类啊,通过函数指针类型的成员指向不同的函数, 实现了多态。

作者回复: 是的





**Geek\_WMK**

2019-05-27



@why是小灰吧

展开 ∨



**nora**

2019-05-17



老师，请教一下 虚拟运行时间  $vruntime += \text{实际运行时间} \times \text{NICE\_0\_LOAD} / \text{权重}$  中NICE\_0\_LOAD是什么意思呢？cfs调度实体是红黑树，那实时进程呢，是普通队列嘛？

展开 ∨



**一步**

2019-05-11



虚拟运行时间  $vruntime += \text{实际运行时间} \times \text{NICE\_0\_LOAD} / \text{权重}$   
同样的实际运行时间，给高权重的算少了，低权重的算多了

这里的权重不是优先级吧，那么权重和优先级是怎么个对应关系呢？

展开 ∨



**郑晨Cc**

2019-05-10



CFS对vruntime 的排序为啥要用复杂的红黑树呢？他只需要知道最小的vruntime和插入新的vruntime，也不需要随机的查询，按理说用小顶堆更合适的啊



**一步**

2019-05-10



对于实时进程，优先级的范围是 0 ~ 99；对于普通进程，优先级的范围是 100~139  
这里为什么 实时进程 的范围是100，而普通进程的优先级范围才是40？

展开 ∨



**nora**

2019-05-08



所以Linux下其实进程和线程一样的，都会占用task，故Linux下线程并不会比进程轻量

级?

---



**Helios**

2019-05-07



“每次新进程创建完毕后，都会试图先让新的抢占一次”。因为vruntime比较小，那如果一直有新的任务进来，是不是对

---



**拾贝壳的孩...**

2019-05-07



如果优先队列一直有任务,普通队列的task一直得不到处理，操作系统会怎么做呢?

---



**leon**

2019-05-04



Linux的最小调度单位是线程，每个进程又至少都有一个线程，当我们在谈Linux调度的时候为啥不说是线程调度？所谓的进程调度是说的也是调度的里面的线程吗？那是不是就没有所谓的进程调度了呢？

展开 ∨

作者回复: 是的，叫任务调度吧



**小龙的城堡**

2019-05-04



老师，有两个疑问：1.cfs模式下，在选取下一个要执行的任务为什么不用优先级队列，而用红黑树，我觉得无非就是取下一个运行时间最少的任务，没有查询的需求；2.课后习题是不是要用到有关设置进城类型的系统调用才能完成，毕竟调度发生在内核态。多谢老师！