# lab3

## 1.数据结构

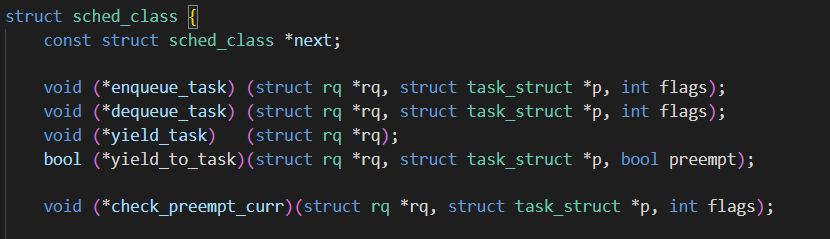
### 1.1fair\_sched\_class

结构体详细解释如图。主要定义了所有的cfs调度算法所用到的函数以及调度算法切换用到的函数。

其中.next是指向下一种调度算法实例的指针，便于调度算法切换。



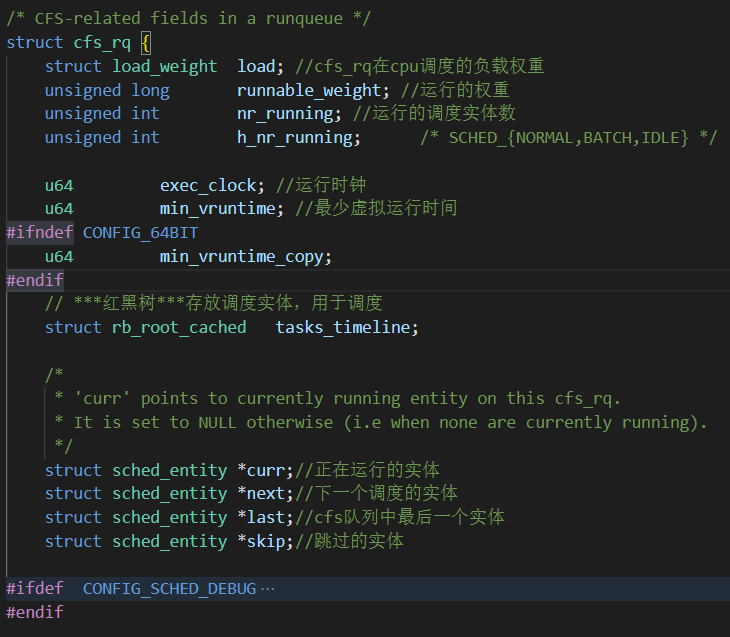
openeuler（linux）中定义了一个调度类sched\_class，定义了调度算法需要的钩子函数（函数指针）。所有的调度实例都是这个类的实例化。这个类的声明在kernel/sched/sched.h文件里。



### 1.2cfs\_rq

cfs的运行进程调度队列。可以看到其中最重要的是rb\_root\_cached红黑树结构体定义的任务时间线tasks\_timeline。

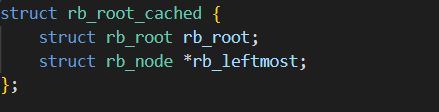
还有定义了几个关键实体：运行实体，下一个实体，最后一个实体，放弃cpu的跳过实体。也就是说调度的对象就是sched\_entity实体结构体代表的进程。



### 1.3红黑树相关数据结构

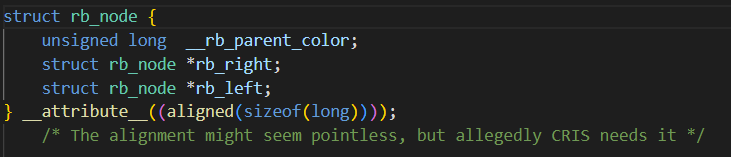
#### 1.3.1rb\_root\_cache

定义了根节点，最左边节点。



#### 1.3.2rb\_node

节点结构体定义了：父节点颜色，左右子节点。



### 1.4sched\_entity

调度实体。主要记录了运行相关的时间信息：开始执行时间exec\_start，执行总时间sun\_exec\_runtime等。

定义了加入红黑树的节点run\_node。

还有群调度相关的参数：该实体所属群的父实体；实体属于的cfs\_rq和实体群自己的cfs\_rq。



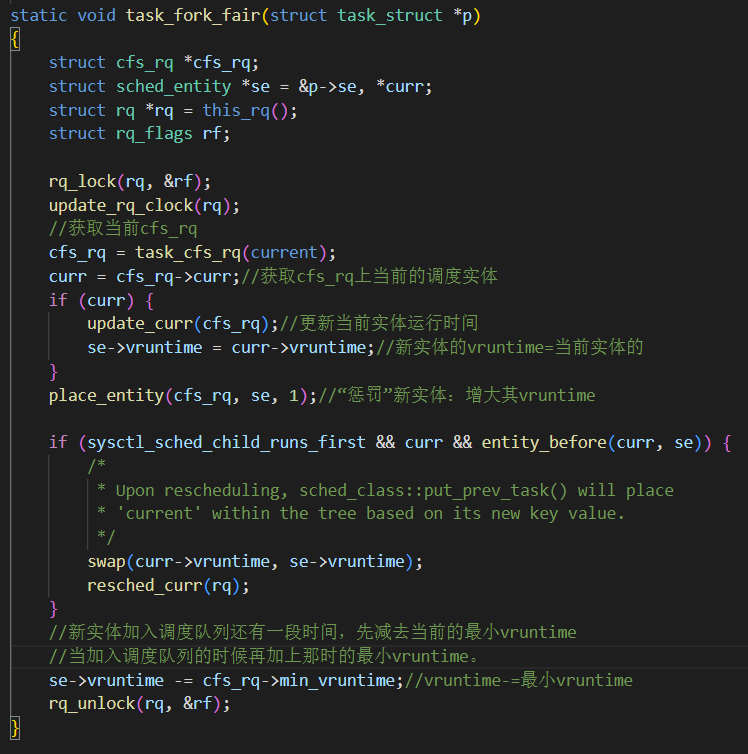
## 函数

除了调度函数schedule()之外，cfs调度算法主要的函数均来自cfs结构体fair\_sched\_class中的定义。

### 2.1cfs加入新节点

#### 2.1.1task\_fork\_fair

对新task的调度实体se，赋值其vruntime，以准备将其加入到调度队列里。



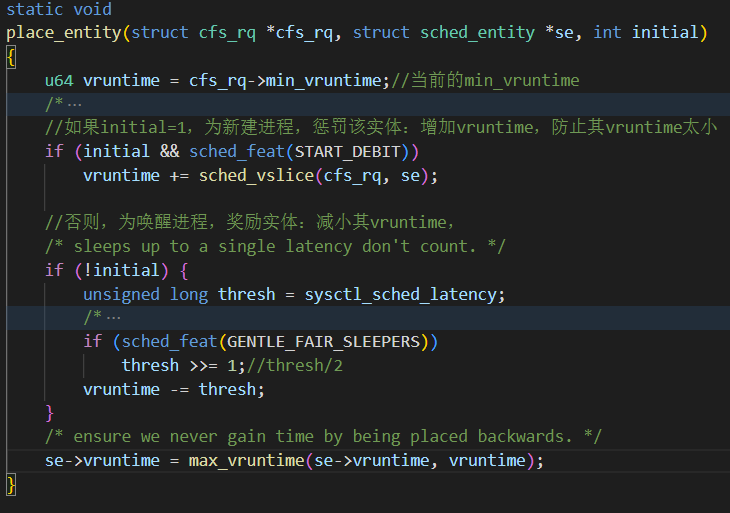
这里并没有直接把进程加入调度队列，只是计算其vruntime为加入调度队列作准备。所以最后se->vruntime-=cfs\_rq->min\_vruntime的目的是减去现在的min\_vruntime，然后在真正加入调度队列的时候再加上那时候的min\_vruntime，这样如果在此期间min\_vruntime改变了也会保证在加入的时候使用了最新的min\_vruntime。

#### 2.1.2place\_entity

根据进程是新建的还是唤醒的对其vruntime作一定调整。

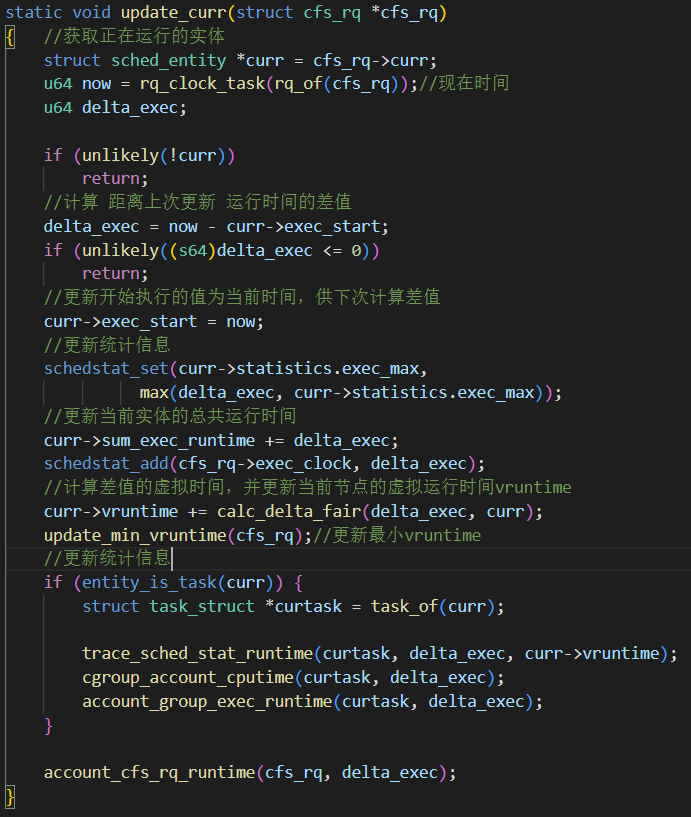
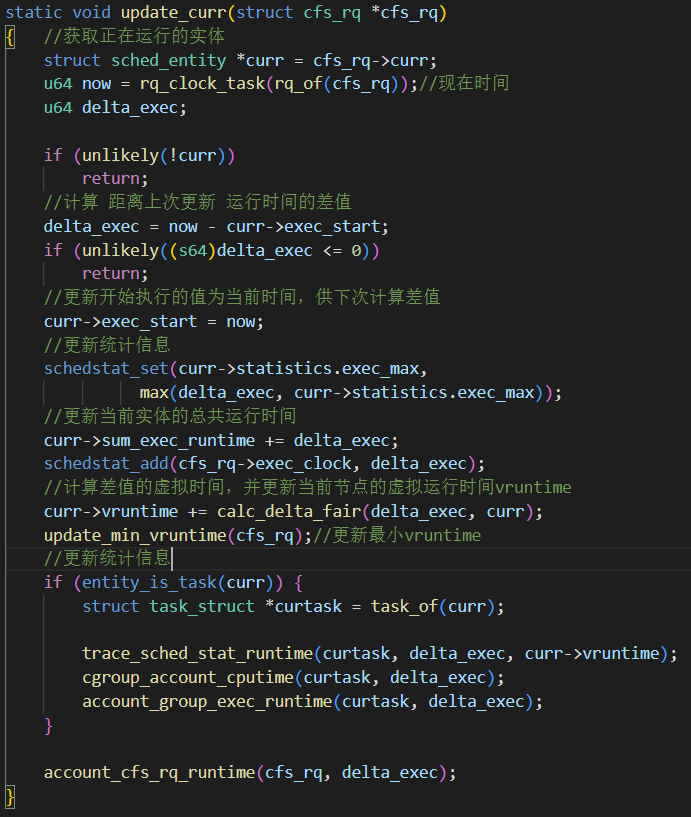
如果进程是新建的，为了防止其vruntime太小而一直占用cpu，对其惩罚，增加vruntime。

如果是唤醒进程，奖励，减小vruntime。



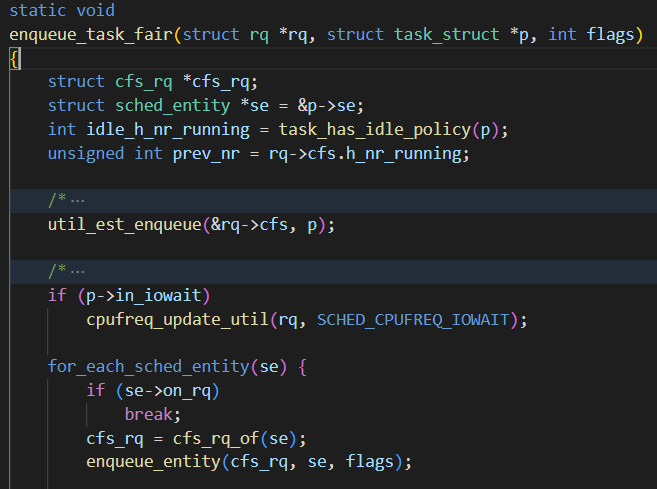
#### 2.1.3update\_curr

更新当前进程的运行时间，记录和更新统计信息。

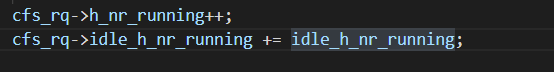


#### 2.1.4enquene\_task\_fair

新进程创建之后由wake\_up\_new\_task调用active\_task函数调用本函数，将新进程的实体加入到红黑树中。

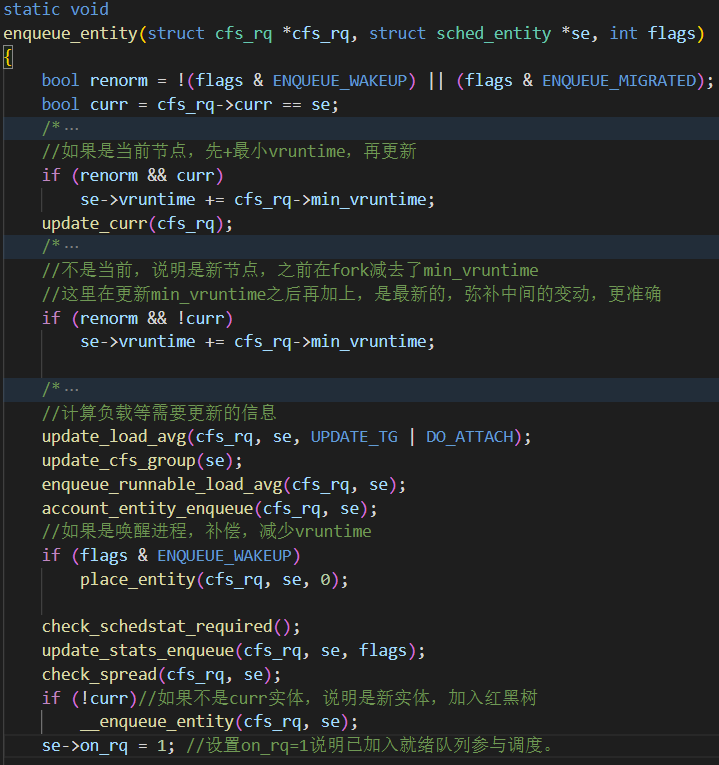


可以看到主要调用enquene\_entity来把实体加入cfs\_rq。然后更新了一下统计信息：可运行实体和空闲可运行实体



#### 2.1.5enquene\_entity

将一个实体加入就绪队列。



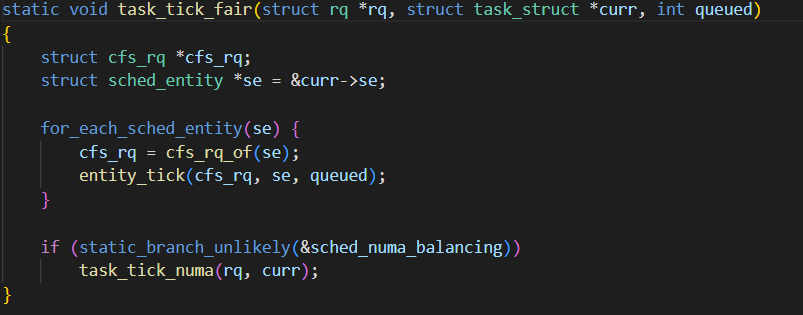
之前task\_fork\_fair的时候减去了当时的min\_vruntime，这里在update\_curr之后加上新的min\_vruntime，则虚拟时间是最新的，不会因为这之间的min\_vruntime改变而有太大误差。

最后如果是新实体，调用\_\_enquene\_entity加入红黑树；并且se->on\_rq设置为1，表示加入了就绪队列。至此一个新进程的调度实体成功添加到了就绪队列。

### 2.2cfs调度

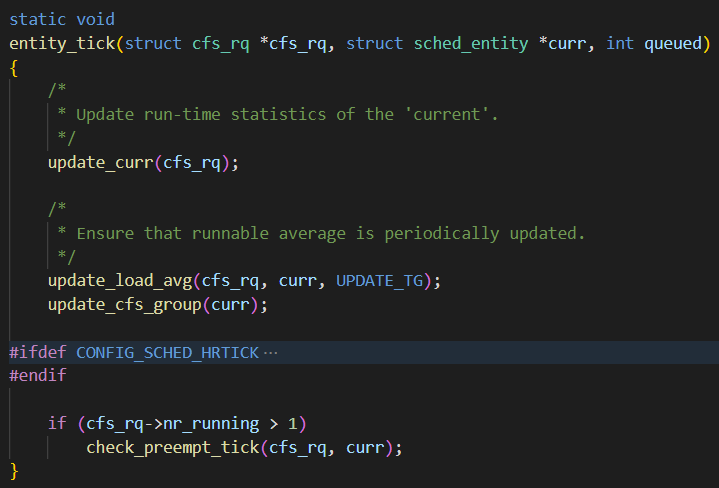
#### 2.2.1task\_tick\_fair

进程运行的每个tick，检查要不要调度。主要调用entity\_tick进一步看当前实体要不要调度。



#### 2.2.2entity\_tick

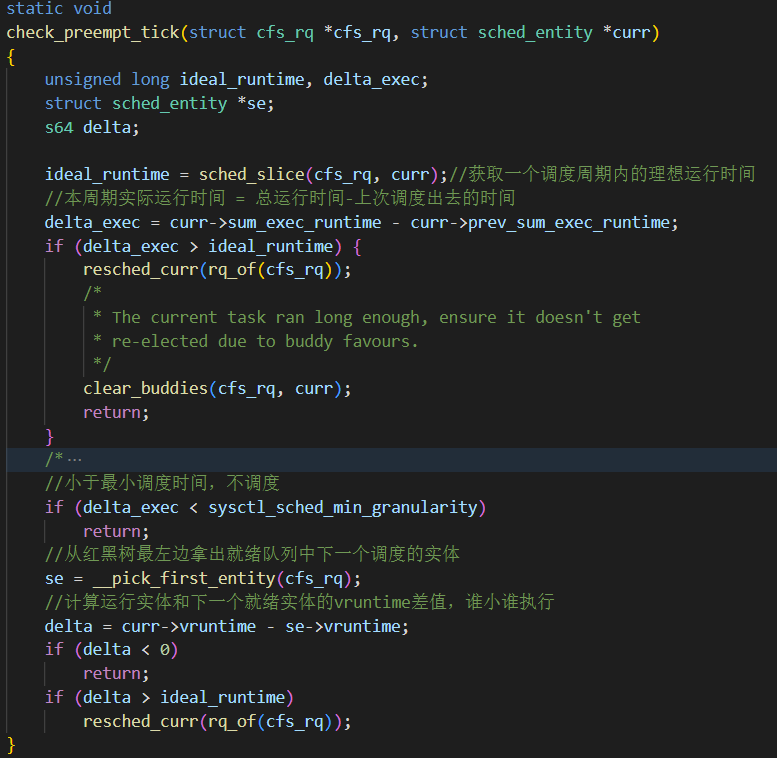
在每个tick，先更新当前运行实体的运行时间等信息updata\_curr。然后进入check\_preempt\_tick检查当前tick要不要切换进程。



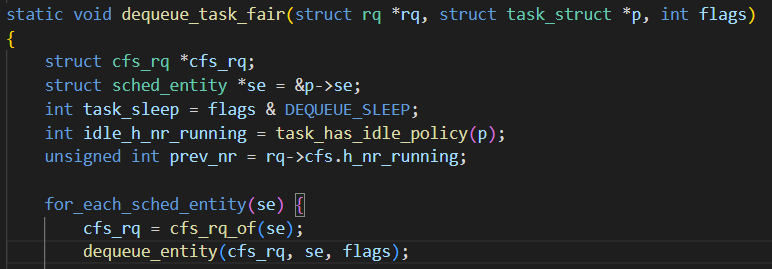
#### 2.2.3check\_preempt\_tick

通过比较运行时间判断要不要抢占并切换当前运行进程。

如果需要调度，则调用resched\_curr()，将当前进程设置为需要即刻调度。

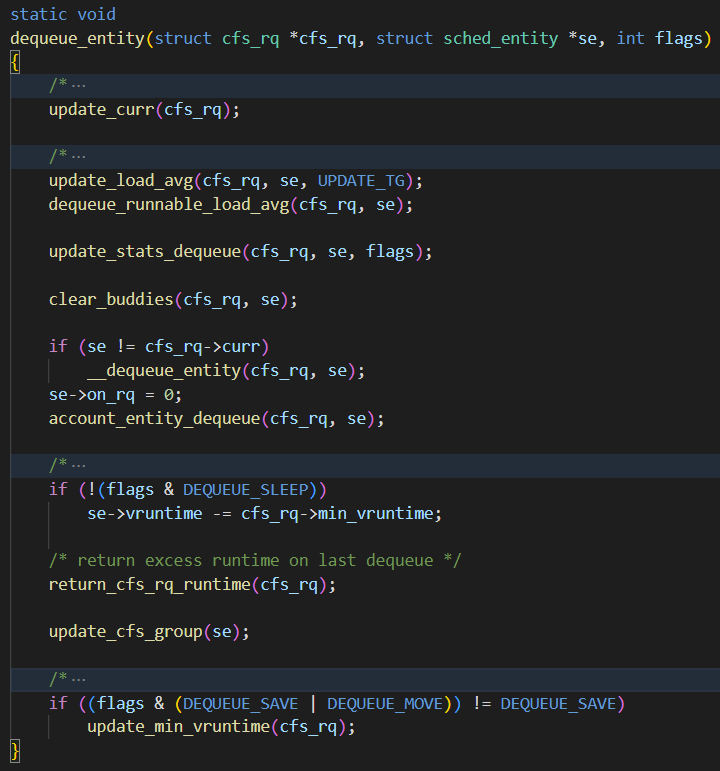


#### 2.2.4dequene\_task\_fair



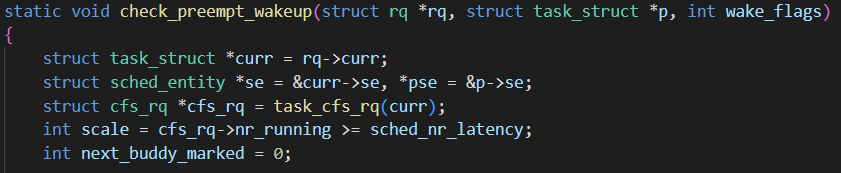
#### 2.2.5dequeue\_entity

实体移除调度队列和加入是完全相反的逻辑。先更新一下权重，可执行实体数量--，把实体从红黑树上去除，期间更新curr和min\_vruntime。

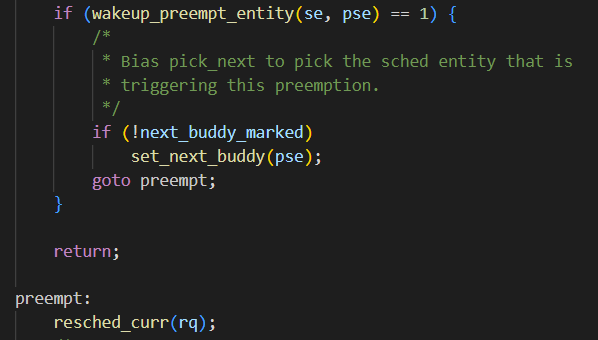


#### 2.2.6check\_preempt\_wakeup

被wake\_up\_new\_task中调用的check\_preempt\_curr调用。判断是否抢占。



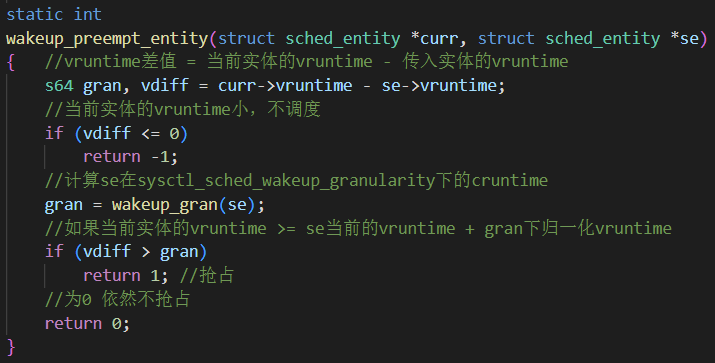
如果wakeup\_preempt\_entity返回值为1，则说明需要抢占，进入preempt段，调用resched\_curr把当前实体设置为需要立刻调度。



#### 2.2.7wakeup\_preempt\_entity

判断当前实体curr要不要被唤醒实体se抢占。判断还是基于vruntime。

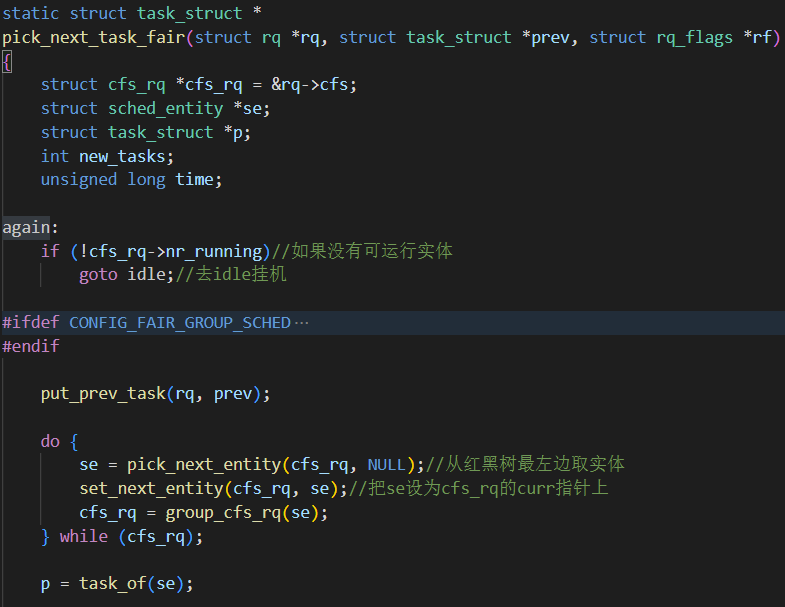
这个函数主要被pick\_next\_entity调用来判断启用哪个实体作为下一个实体。另外被check\_preempt\_wakeup用于判断新实体要不要抢占执行。



#### 2.2.8pick\_next\_task\_fair

公平的机制建立在红黑树结构体上，所以公平的调度下一个实体也就是从红黑树最左边取下节点，放入cfs\_curr指针。

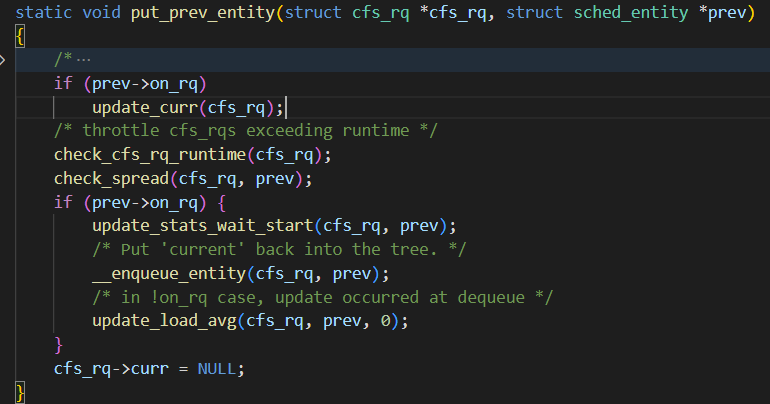
pick\_next\_task\_fair主要被\_\_schedle函数调用，选取下一个调度的实体。



#### 2.2.9put\_prev\_entity

由put\_pre\_task调用或者由pick\_next\_task\_fair调用，将curr指针设置为null；同时更新一些统计信息，为调度下一个实体做准备工作。

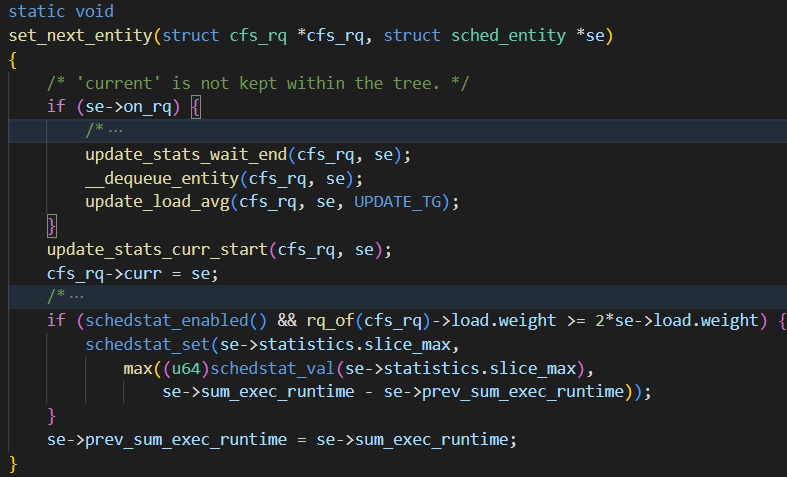
如果是由pick\_next\_task\_fair下一步调用set\_next\_enity把curr指针设置为pick\_next\_entity选择的实体真正完成进程调度。



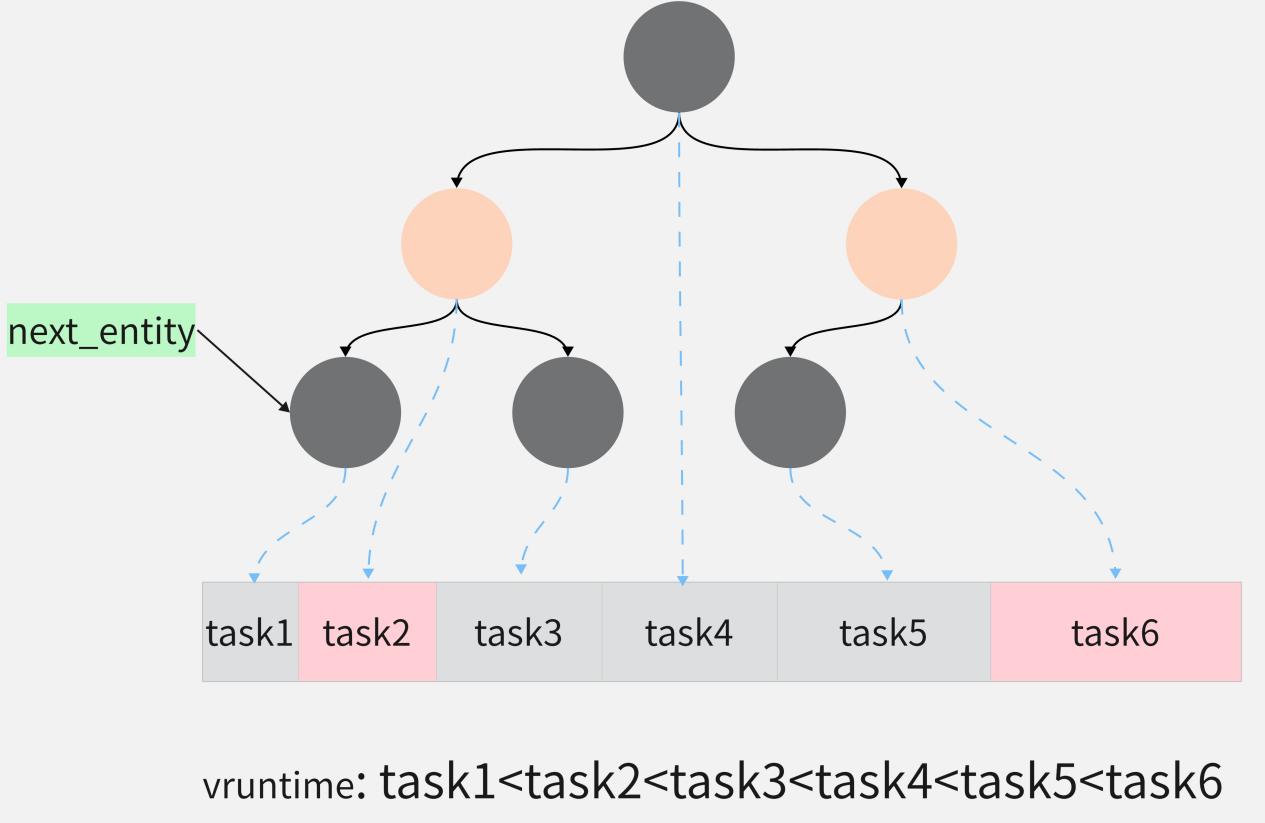
#### 2.2.10set\_next\_entity

将pick\_next\_entity的实体设置为curr。先将需要设置为curr的实体从cfs就绪队列里出列，然后设置为curr。期间更新一些统计信息。

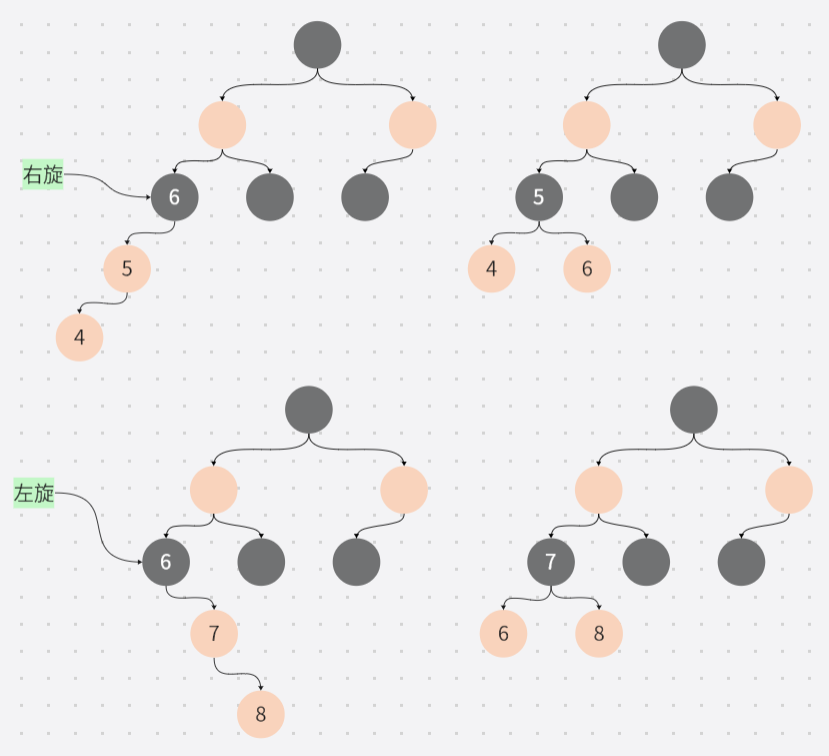
然后记录一下之前的curr节点的prev总执行时间。



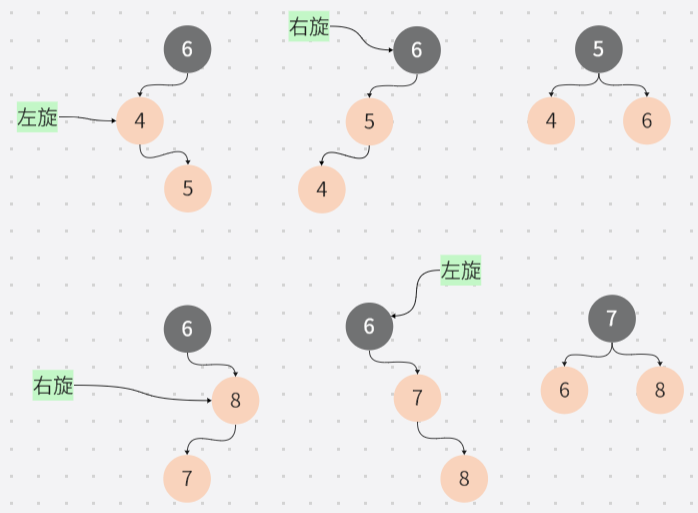
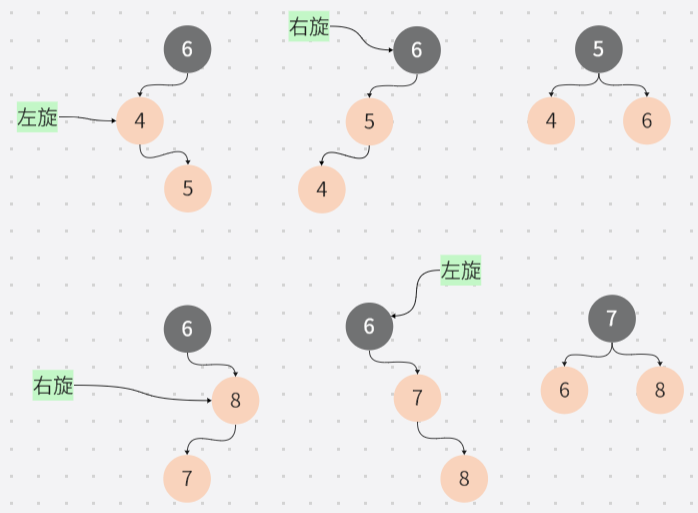
## 红黑树



1. 红黑树只保存就绪态的任务的调度实体，配合就绪队列如上图组织。
2. 红黑树的作用：
   1. 当**执行调度**的时候，调用pick\_next\_entity的时候，选出当前vruntime最小的就绪态任务next\_entity，作为下一个执行的任务。
   2. 当**加入或者删除实体**的时候，维护红黑树的性质不变。
3. **调整**过程：红黑树的调整分为插入和删除，对应着\_\_enqueue一个调度实体和\_\_dequeue一个调度实体的情况。具体的调整过程见图。
   1. 插入：
      1. LL,RR不上溢

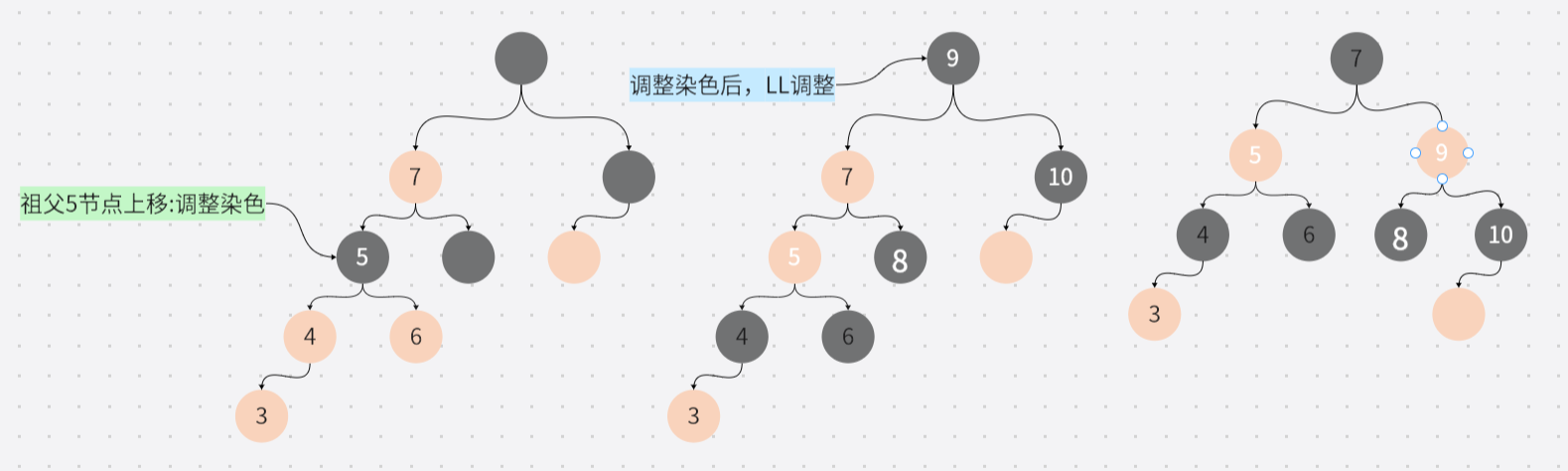


* + 1. LR,RL不上溢：分别对父节点进行一次左旋，右旋变成LL和RR情况，再和1中一样调整即可。



* + 1. 上溢：上溢的调整非常简单，只需要把祖父节点在等效为B+树的情况下上移一层，也就是只染色，不做任何改变，染色之后相当于祖父是新加入的节点再根据LL,LR，RL,RR也就是之前的情况调整即可。

可以看到这里调整染色后实际上是祖父和父亲节点的LL情况，调整的方法和LL一样，都是右旋一次即可。



* 1. 删除

删除的时候分为三种情况1：删除节点没有子节点，直接删除；2：删除节点只有一个子节点，用子节点替换；3：删除节点有两个子节点，任选一个子节点的值覆盖父节点的值，然后删除原来的子节点，这时候仍有三种情况，如此递推下去，知道出现情况1或2即可。

删除后的调整：删除红节点不需要调整，删除黑节点会导致不是每个子树的黑色节点高度都相同，需要调整。调整的思路和插入是相同的。

1. 选择红黑树作为cfs数据结构的原因。
   1. cfs算法每次选择vruntime最小的就绪态任务作为下一个调度的任务，所以需要尽快的选出最小调度实体并出队列。红黑树维护就绪队列，可以O(1)的选出min\_vruntime实体（单独定义了选择最左节点的函数），并且O(logN)的删除该节点。
   2. 为什么不用AVG树或者查找二叉树？
      1. AVG树和红黑树性能差不多，但维护成本更高：删除节点的时候最坏可能需要logN次旋转。
      2. 查找二叉树虽然维护成本最低，但是对于特殊数据分布——数据初始或者加入时有序，查找性能会退化到O(N)。
      3. 就绪队列是需要**频繁查找并增删**的。均衡查找性能和维护成本，红黑树无疑是更好的选择：查找的情况，红黑树没有退化情况，其查找性能稳定在O(logN)；删除节点的时候红黑树不需要像AVG树那样严格平衡，最多只需要旋转3次。
   3. 为什么不用堆？
      1. 就绪队列需要**频繁增删**，堆结构看上去删除一个数据只需要维护堆结构，也就是O(log(N))，当当前进程**被抢占**，删除堆顶元素(也就是vruntime最小的元素)的时候是这样的；但是要考虑到有可能进程**主动休眠**(例如进程需要调用i/o等，自己申请阻塞自己)，这种从就绪队列删除的情况需要先查找得到节点再删除和维护堆，时间复杂度是O(N)，这种情况时间复杂度大于红黑树的O(logN)。也就是说，从性能上，由于数据的频繁增删，红黑树的性能要好于堆结构。
      2. linux对堆结构的存储是连续数组空间，这样的连续空间可能成为性能瓶颈。

## 函数调用流程图

见文件：lab3函数调用流程图.pdf