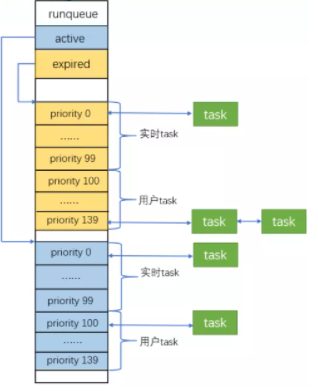
**调度的发展历史**

| **字段** | **版本** |
| --- | --- |
| O(n) 调度器 | linux0.11 - 2.4 |
| O(1) 调度器 | linux2.6 |
| CFS调度器 | linux2.6至今 |

* O(n) 调度器是在内核2.4以及更早期版本采用的算法，其调度算法非常简单和直接，就绪队列是个全局列表，从就绪队列中查找下一个最佳任务，由于每次在寻找下一个任务时需要遍历系统中所有的任务（全局列表），因此被称为 O(n) 调度器（时间复杂度）。
* 内核2.6采用了O(1) 调度器，让每个CPU维护一个自己的就绪队列，从而减少了锁的竞争。就绪队列由两个优先级数组组成，分别是**active优先级数组**和**expired优先级数组**。每个优先级数组包含140个优先级队列，也就是每个优先级对应一个队列，其中前100个对应实时进程，后40个对应普通进程。如下图所示：



这样设计的好处，调度器选择下一个被调度任务就变得高效和简单多了，只需要在active优先级数组中选择优先级高，并且队列中有可运行的任务即可。这里使用位图来定义该队列中是否有可运行的任务，如果有，则位图中相应的位就会被置1。这样选择下一个被调用任务的时间就变成了查询位图的操作。

但上面的算法有个问题，一个高优先级多线程的应用会比低优先级单线程的应用获得更多的资源，这就会导致一个调度周期内，低优先级的应用可能一直无法响应，直到高优先级应用结束。**CFS调度器**就是站在一视同仁的角度解决了这个问题，保证在一个调度周期内每个任务都有执行的机会，执行时间的长短，取决于任务的权重。下面详细看下CFS调度器是如何动态调整任务的运行时间，达到公平调度的。

-------------------------

CFS是Completely Fair Scheduler简称，即完全公平调度器。

linux在2.6.23内核中引入scheduling class的概念，将调度器模块化，系统中可以有多种调度器，使用不同策略调度不同类型的进程：

1. DL（deadline） Scheduler 采用sched\_deadline策略，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。
2. RT（实时） Scheduler 采用sched\_rr和sched\_fifo策略
3. CFS Scheduler 采用sched\_normal和sched\_batch策略
4. IDEL（闲置） Scheduler 采用sched\_idle策略

这样一来，CFS调度器就不关心实时进程了，专注于普通进程就可以了。

CFS( Completely Fair Scheduler )完全公平调度器，从实现思想上和之前的O(1)/O(n)很不一样。

O(1)和O(n)都将CPU资源划分为时间片，采用了固定额度分配机制，在每个调度周期进程可使用的时间片是确定的，调度周期结束被重新分配。

CFS摒弃了固定时间片分配，采用动态时间片分配，本次调度中进程可占用的时间与进程总数、总CPU时间、进程权重等均有关系，每个调度周期的值都可能会不一样。

CFS调度器从进程优先级出发，它建立了优先级prio和权重weight之间的映射关系，把优先级转换为权重来参与后续的计算：

左边星号内是nice值，普通进程的优先级范围是[100,139]，prio整体减小120就和代码左边的注释对上了，也就是nice值的范围[-20,19]

const int sched\_prio\_to\_weight[40] = {  
 */\* -20 \*/*     88761,     71755,     56483,     46273,     36291,  
 */\* -15 \*/*     29154,     23254,     18705,     14949,     11916,  
 */\* -10 \*/*      9548,      7620,      6100,      4904,      3906,  
 */\*  -5 \*/*      3121,      2501,      1991,      1586,      1277,  
 */\*   0 \*/*      1024,       820,       655,       526,       423,  
 */\*   5 \*/*       335,       272,       215,       172,       137,  
 */\*  10 \*/*       110,        87,        70,        56,        45,  
 */\*  15 \*/*        36,        29,        23,        18,        15,  
};

因此sched\_prio=0（调度优先级nice）相当于static\_prio=120（静态优先级）。

比如现有进程A sched\_prio=0，进程B sched\_prio=-5，通过sched\_prio\_to\_weight的映射：

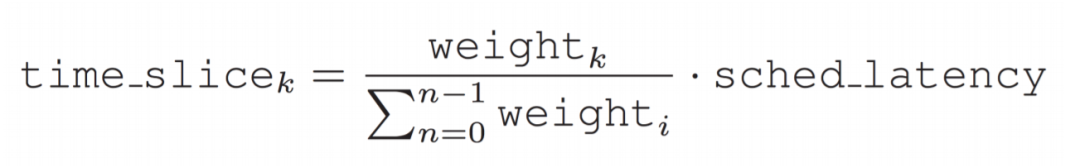
进程A 权重weight=1024，进程B权重 weight = 3121

进程A的CPU占比 = 1024/(1024+3121)= 24.7%

进程B的CPU占比 = 3121/(1024+3121) = 75.3%

假如CPU总时间是10ms，那么根据A占用2.47ms，B占用7.53ms

在CFS中引入sysctl\_sched\_latency(调度延迟)作为一个调度周期，真实的CPU时间表示为：



**实际运行时间 = 调度周期 \* 进程权重 / 所有进程权重之和**

显然这样根据权重计算后的各个进程的运行时间是不等的，也就违背了"完全公平"思想，于是CFS引入了虚拟运行时间(virtual runtime)。

虚拟运行时间

每个进程的物理运行时间时肯定不能一样的，CFS调度器只要保证的就是进程的虚拟运行时间相等即可。

**知识点：**前面已知CFS调度器用nice值表示优先级，取值范围是[-20, 19]，nice和权重是一一对应的关系。数值越小代表优先级越大，同时也意味着权重值越大（如前表所示）。

公式中以1024权重为基准值计算得来，1024权重对应nice值为0，其权重被称为NICE\_0\_LOAD。默认下，大部分进程的权重基本都是NICE\_0\_LOAD。

**虚拟运行时间**

虚拟时间vriture\_runtime和实际时间（wall time）转换公式如下：

**虚拟运行时间 = 实际运行时间 \* NICE\_0\_LOAD / 进程权重**

**= (调度周期 \* 进程权重 / 所有进程权重之和) \* NICE\_0\_LOAD /** **进程权重**

**= 调度周期 \* 1024 / 所有进程总权重**

* 例如现有进程A sched\_prio=0，进程B sched\_prio=-5：
* 调度延迟=10ms，A的运行时间=2.47ms B的运行时间=7.53ms，也就是wall\_time
* nice\_0\_weight表示sched\_prio=0的权重为1024
* 进程A的虚拟时间：2.47\*1024/1024=2.47ms
* 进程B的虚拟时间：7.53\*1024/3121=2.47ms
* 经过这样映射，A和B的虚拟时间就相等了。

从公式可以看出，**在一个调度周期刚开始的时候，所有进程的虚拟运行时间是相同的。**

虚拟运行时间与优先级的衰减因子有关，也就是**”1/进程权重”**随着nice值增大而增大，同时其作为分母也加速了低优先级进程的衰减。

nice=0 虚拟运行时间 = 物理运行时间

nice>0 虚拟运行时间 > 物理运行时间

nice<0 虚拟运行时间 < 物理运行时间

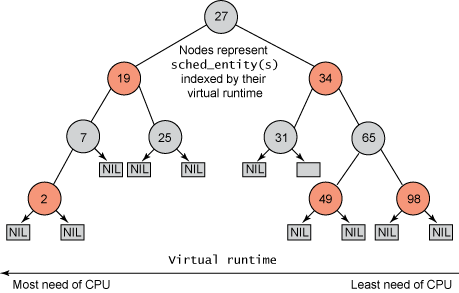
**而我们在进程调度时，需要找到虚拟运行时间最小的进程调度运行，一开始是随机任意进程都一样。**

**然后我们设一个变量保存进程已经使用的虚拟运行时间，将其作为vruntime时间。**

为了能够快速找到虚拟运行时间最小的进程，Linux 内核使用红黑树来保存可运行的进程。

**于是会看到：随着任务的执行，进程的运行时间增加，因此它的vruntime也会变大，它会在红黑树中向右移动。而虚拟时间数值最小的放到最左的叶子节点，这个节点就是下一个被pick的进程了。**

CFS跟踪调度实体sched\_entity的虚拟运行时间vruntime，将sched\_entity通过enqueue\_entity()和dequeue\_entity()来进行红黑树的出队入队，vruntime少的调度实体sched\_entity排列到红黑树的左边。



如上图所示，红黑树的左节点比父节点小，而右节点比父节点大。所以查找最小节点时，只需要获取红黑树的最左节点即可。

相关步骤如下：

1. 每个sched\_latency周期内，根据各个任务的权重值，可以计算出运行时间runtime；
2. 运行时间runtime可以转换成虚拟运行时间vruntime；
3. 根据虚拟运行时间的大小，插入到CFS红黑树中，虚拟运行时间少的调度实体放置到左边；
4. 在下一次任务调度的时候，选择虚拟运行时间少的调度实体来运行（pick\_next\_task从就绪队列中选择最适合运行的调度实体，即虚拟时间最小的调度实体）；

**CFS 数据结构**

* task\_struct: 任务描述符，包含很多进程相关的信息，例如，优先级、进程状态以及调度实体等。

struct task\_struct {  
    ...  
    struct sched\_entity se;  
    ...  
}

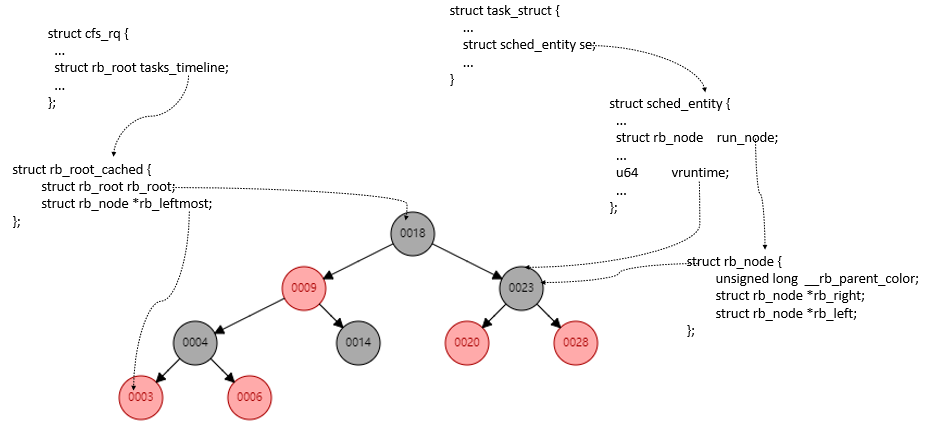
* cfs\_rq：跟踪就绪队列信息以及管理就绪态调度实体，并维护一棵按照虚拟时间排序的红黑树。tasks\_timeline->rb\_root是红黑树的根，tasks\_timeline->rb\_leftmost指向红黑树中最左边的调度实体，即虚拟时间最小的调度实体。

struct cfs\_rq {  
  ...  
  struct rb\_root\_cached tasks\_timeline  
  ...  
};

* sched\_entity：可被内核调度的实体。每个就绪态的调度实体sched\_entity包含插入红黑树中使用的节点rb\_node，同时vruntime成员记录已经运行的虚拟时间。

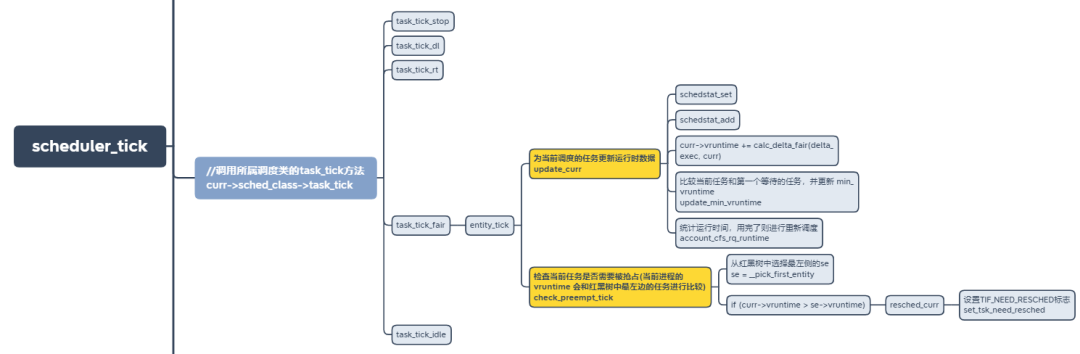
struct sched\_entity {  
  ...  
  struct rb\_node    run\_node;        
  ...  
  u64          vruntime;                
  ...  
};

这些数据结构的关系如下图所示：



**CFS 算法实现**

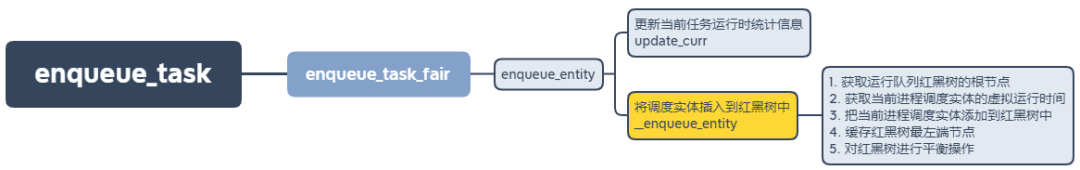
1. **时钟中断 scheduler\_tick 更新虚拟运行时间，检查是否需要抢占。**



* 更新运行时的各类统计信息，比如vruntime， 运行时间、负载值、权重值等。
* 检查是否需要抢占，主要是比较运行时间是否耗尽，以及vruntime的差值是否大于运行时间等。

1. **任务出队入队**

当任务进入可运行状态时，用 enqueue\_task\_fair 将调度实体放入到红黑树中，完成入队操作；当任务退出可运行状态时，用 dequeue\_task\_fair 将调度实体从红黑树中移除，完成出队操作；队操作。



调用 \_\_enqueue\_entity 函数后，就可以把进程调度实体插入到运行队列的红黑树中。同时会把红黑树最左端的节点缓存到运行队列的 rb\_leftmost 字段中，用于快速获取下一个可运行的进程。

1. **从 cfs\_rq 中获取下一个可运行的任务**

每当进程任务切换的时候，也就是schedule函数执行时，调度器都需要选择下一个将要执行的任务。在CFS调度器中，是通过 pick\_next\_task\_fair 函数完成的，其本质是从就绪队列中选择最适合运行的调度实体（虚拟时间最小的调度实体）。

