CFS 中的调度器数据结构

调度类

CFS调度器是在Linux2.6.23引入的，在当时就提出了调度类概念，调度类就是将调度策略模块化，有种面向对象的感觉。先来看下调度类的数据结构，调度类是通过struct sched_class数据结构表示

```
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int
flags);

    /*
     * It is the responsibility of the pick_next_task() method that will
     * return the next task to call put_prev_task() on the @prev task or
     * something equivalent.
     *
     * May return RETRY_TASK when it finds a higher prio class has runnable
     * tasks.
     */
    struct task_struct * (*pick_next_task)(struct rq *rq,
                           struct task_struct *prev,
                           struct rq_flags *rf);
    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
    void (*set_curr_task)(struct rq *rq);
    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);

```

next 是用来指向下一级的一个调度类，内核中为每个调度策略提供了一个调度类，这些调度类是通过next成员链接到一起
enqueue_task: 用来将一个进程添加到就绪队列中，同时会增加它的可运行的进程数
dequeue_task: 用来将一个进程从就绪队列移除，同时减少可运行进程的数量
check_preempt_curr: 用来检测当一个进程的状态设置为runnable时，检查当前进程是否可以发生抢占
pick_next_task: 在运行队列中选择下一个最合适的进程来运行
put_prev_task: 获得当前进程之前的那个进程
set_curr_task: 用来设置当前进程的调度状态等
task_tick: 在每个时钟tick的时候会调度各个调度类中的tick函数

Linux内核都提供了那些调度类

extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;

Linux内核定义了五种调度类，而且每种调度有对应的调度策略，而每种调度策略有会对应调度哪些进程。
同时也提供了六种调度策略。下表示调度策略和调度类之间的关系

调度类　调度策略 调度对象
stop_sched_class（停机调度类）　无　停机的进程
dl_sched_class（限期调度类）　SCHED_DEADLINE　dl进程
rt_sched_class（实时调度类）　SCHED_RR 或者 SCHED_FIFO 实时进程
fair_sched_class（公平调度类）　SCHED_NORMAL 或者 SCHED_BATCH 普通进程
idle_sched_class（空闲调度类）　SCHED_IDLE idle进程
同时这些调度类之间是有优先级关系的。

如果定义了SMP，则最高优先级的是stop调度类。调度类的优先级关系

stop_sched_class > dl_sched_class > rt_sched_class > fair_sched_class > idle_shced_class

调度实体

struct sched_entity {
　/* For load-balancing: */
　struct load_weight　　load;
　unsigned long　　runnable_weight;
　struct rb_node　　run_node;
　struct list_head　group_node;
　unsigned int　　on_rq;

```
u64             exec_start;
u64             sum_exec_runtime;
u64             vruntime;
u64             prev_sum_exec_runtime;

u64             nr_migrations;

struct sched_statistics     statistics;
```

从Linux2.6.23开始引入了调度实体的概念，调度实体封装了进程的一些重要的信息。在之前的O(1)算法中调度的单位都是task_struct，而在Linux2.6.23引入调度模块化后，调度的单位成为调度实体sched_entity

load就是此进程的权重
run_node:CFS调度是通过红黑树来管理进程的，这个是红黑树的节点
on_rq: 此值为1时，代表此进程在运行队列中
exec_start: 记录这个进程在CPU上开始执行任务的时间
sum_exec_runtime: 记录这个进程总的运行时间
vruntime: 代表的是进程的虚拟运行时间
prev_sum_exec_runtime: 记录前面一个进程的总的运行时间
nr_migrations: 负载均衡时进程的迁移次数
statistics：进程的统计信息

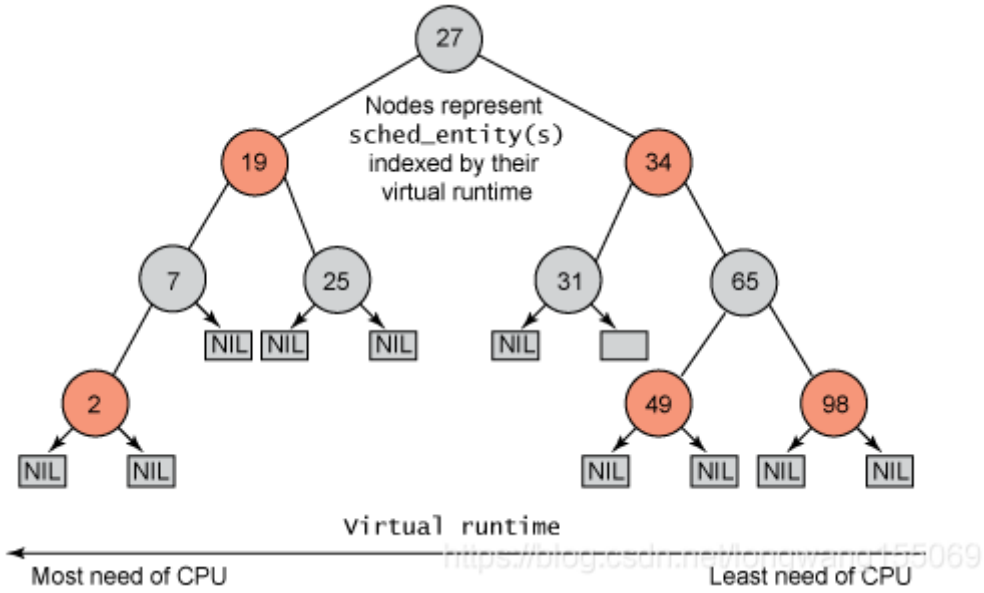
红黑树

树左边节点的值永远比树右边接的值小。

在O(n)和O(1)的调度器中运行队列都是通过数组链表来管理的，而在CFS调度中抛弃了之前的数据结构，采用了以时间为键值的一棵红黑树。其中的时间键值就是进程的vruntime。

CFS维护了一个以时间为排序的红黑树，所有的红黑树节点都是通过进程的se.vruntime来作为key来进行排序。CFS每次调度的时候总是选择这棵红黑树最左边的节点，然后来调度它。随着时间的推移，之前在最左边的节点随着运行，进程的vruntime也随之增大，这些进程慢慢的会添加到红黑树的右边。循环往复这个树上的所有进程都会被调度到，从而达到的公平。

同时CFS也会维护这棵树上最小的vruntime的值cfs.min_vruntime，而且这个值是单调递增的。此值用来跟踪运行队列中最小的vruntime的值。



运行队列

系统中每个CPU上有有一个运行队列struct rq数据结构，这个struct rq是个PER-CPU的，每个CPU上都要这样的一个运行队列，可以可以防止多个CPU去并发访问一个运行队列。

```
/*

 * This is the main, per-CPU runqueue data structure.
   *

 * Locking rule: those places that want to lock multiple runqueues

 * (such as the load balancing or the thread migration code), lock

 * acquire operations must be ordered by ascending &runqueue.
   */
   struct rq {

       unsigned int     nr_running;

   /* capture load from *all* tasks on this CPU: */
   struct load_weight   load;
   unsigned long        nr_load_updates;
   u64          nr_switches;

   struct cfs_rq        cfs;
   struct rt_rq     rt;
   struct dl_rq     dl;
   可
```

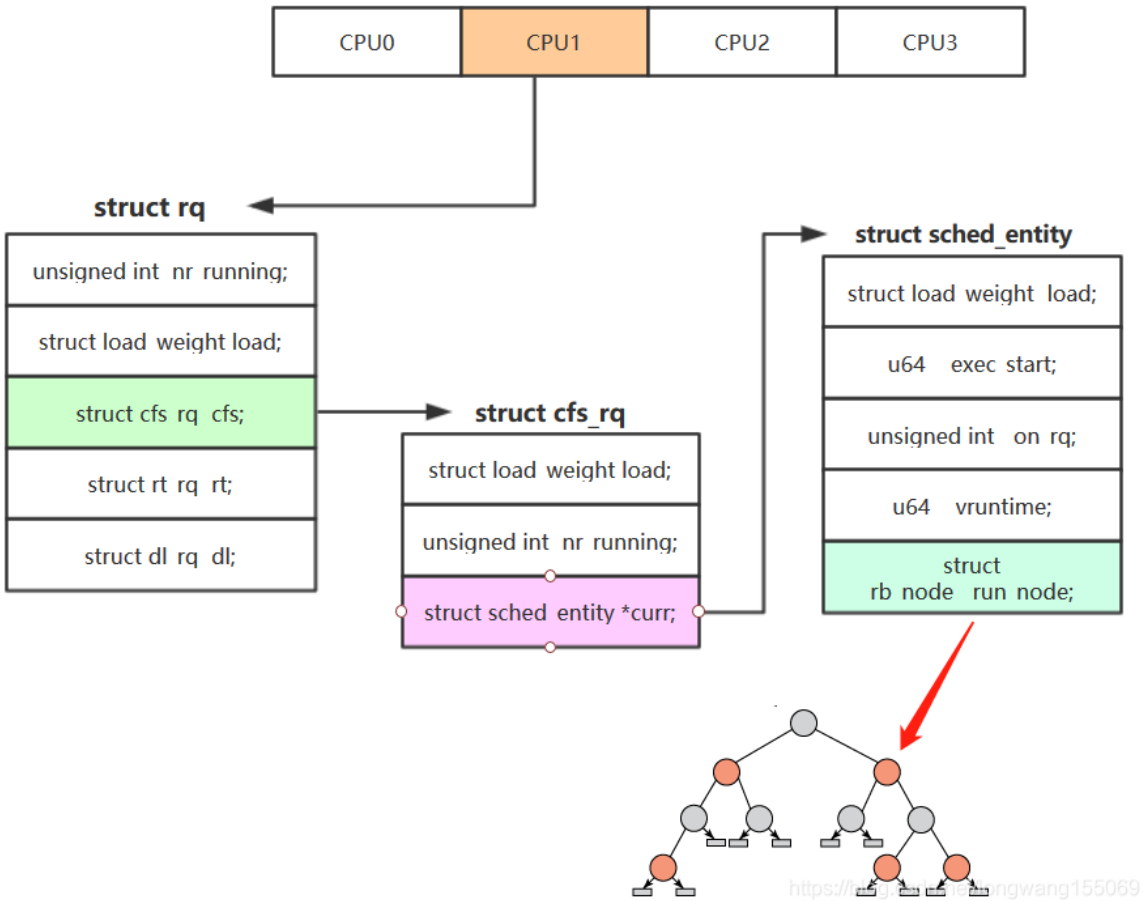
可以从注释看到struct rq是一个per-cpu的变量。

nr_running: 代表这个运行队列上总的运行进程数
load: 在这个CPU上所有进程的权重，这个CPU上可能运行的进程有实时进程，普通进程等
nr_switches: 进程切换的统计数
struct cfs_rq: 就是CFS调度类的一个运行队列
struct rt_rq: 代表的是rt调度类的运行队列
struct dl_rq: 代表的是dl调度类的运行队列
可以得出的一个结论是，一个struct rq中包括了各种类型的进程，有DL的，有实时的，有普通的。通过将不同进程的挂到不同的运行队列中管理。

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight  load;
    unsigned long       runnable_weight;
    unsigned int        nr_running;
    unsigned int        h_nr_running;
u64         exec_clock;
u64         min_vruntime;
```

从注释上看struct cfs_rq代表的是CFS调度策略对应的运行队列

load: 是这个CFS_rq的权重，包含着CFS就绪队列中的所有进程
nr_running: 代表的是这个CFS运行队列中可运行的进程数
min_vruntime: 此值代表的是CFS运行队列中所有进程的最小的vruntime
看下运行队列的关系图



每个CPU中都存在一个struct rq运行队列，struct rq中根据进程调度策略分为不同的运行队列，比如普通进程就会挂载到cfs_rq中，在struct cfs_rq中则定义了每一个调度实体，每一个调度实体根据vruntime的值添加到红黑树的节点中。