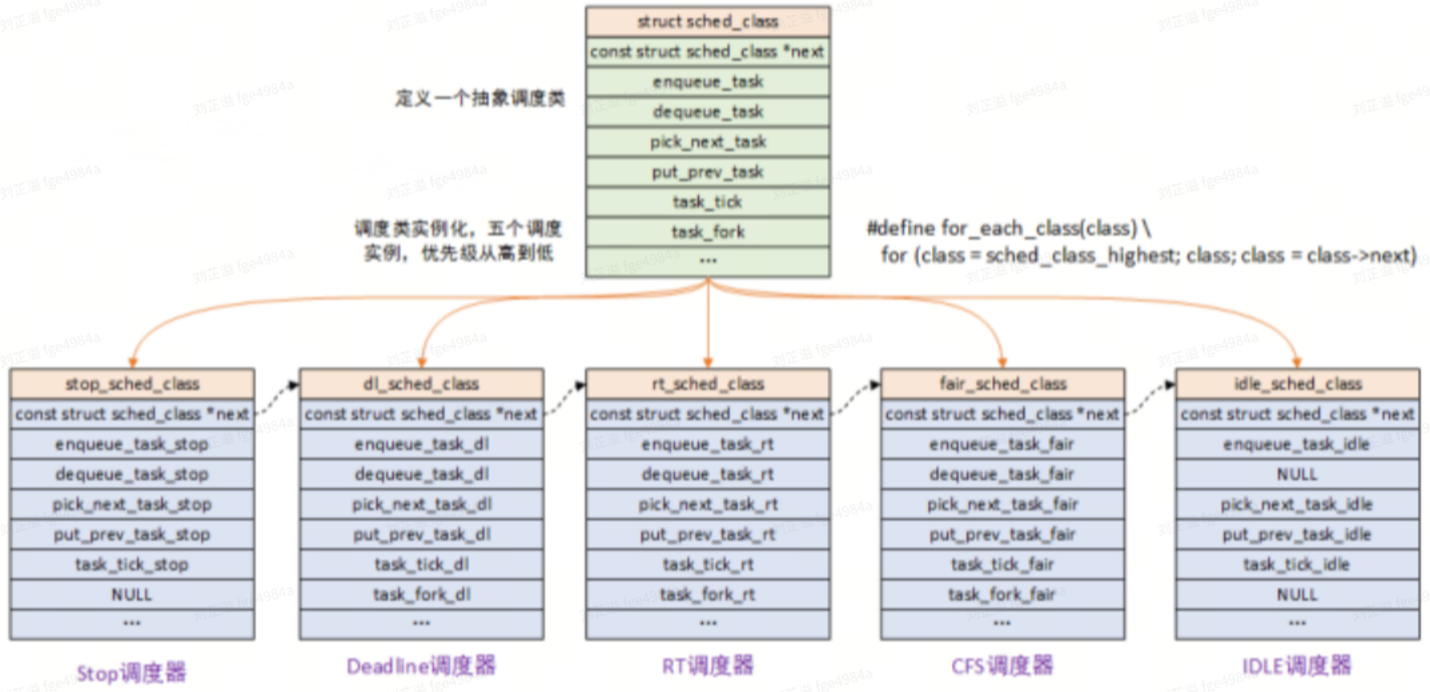
CFS调度

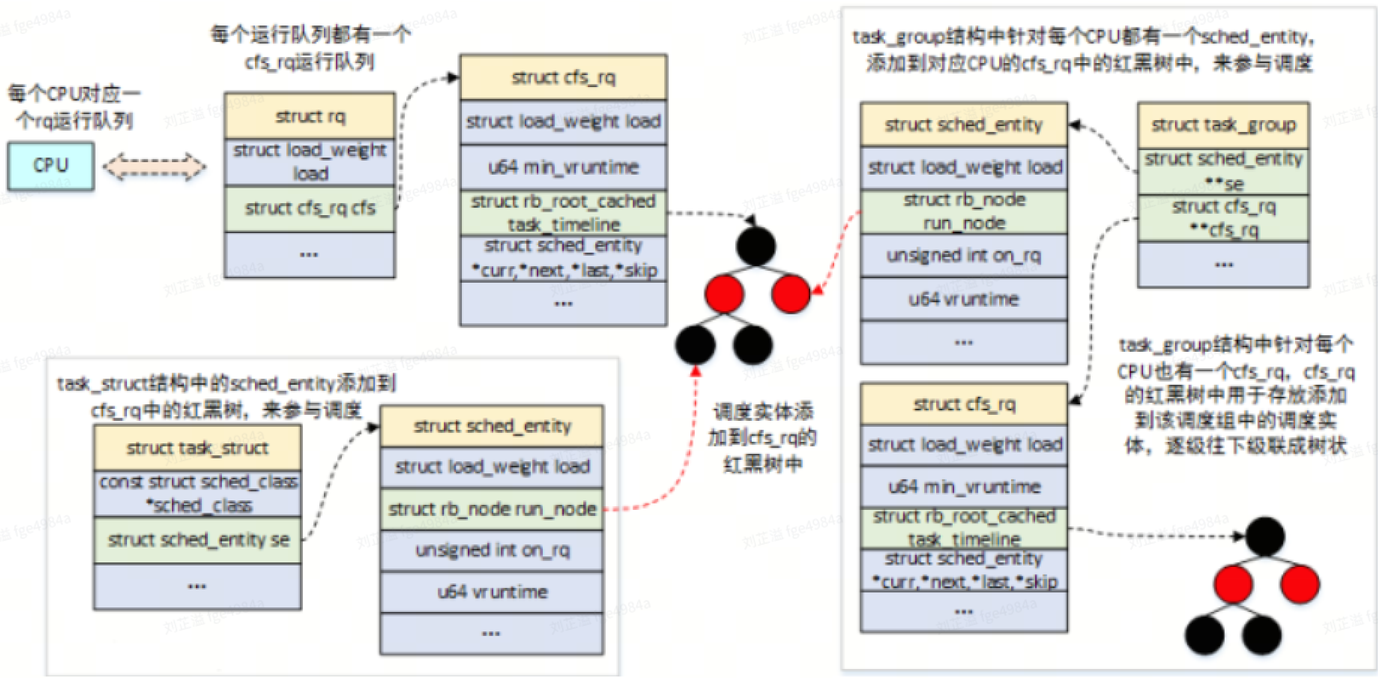
调度器类

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

  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup,
            bool head);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
  void (*yield_task) (struct rq *rq);

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

  struct task_struct * (*pick_next_task) (struct rq *rq);
  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);
  void (*task_fork) (struct task_struct *p);

  void (*switched_from) (struct rq *this_rq, struct task_struct *task,
             int running);
  void (*switched_to) (struct rq *this_rq, struct task_struct *task,
           int running);
  void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
           int oldprio, int running);

  unsigned int (*get_rr_interval) (struct rq *rq,
           struct task_struct *task);
};
```



rq/cfs_rq/task_group/sched_entity

1. struct rq：每个 CPU 都有一个对应的运行队列；
2. struct cfs_rq：CFS 运行队列，**该结构中包含了 struct rb_root_cached 红黑树，用于链接调度实体 struct sched_entity**。rq 运行队列中对应了一个 CFS 运行队列，此外，在 task_group 结构中也会为每个 CPU 再维护一个 CFS 运行队列；
3. struct task_struct：任务的描述符，包含了进程的所有信息，**该结构中的 struct sched_entity，用于参与 CFS 的调度；**
4. struct task_group：组调度，Linux 支持将任务分组来对 CPU 资源进行分配管理，该结构中为系统中的每个 CPU 都分配了 struct sched_entity 调度实体和 struct cfs_rq 运行队列，其中 struct sched_entity 用于参与 CFS 的调度；
5. struct sched_entity：调度实体，这个也是 CFS 调度管理的对象了；



sched_entity

```c
struct sched_entity {
  struct load_weight  load;    /* for load-balancing */
  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      last_wakeup;
  u64      avg_overlap;

  u64      nr_migrations;

  u64      start_runtime;
  u64      avg_wakeup;
//...
#ifdef CONFIG_FAIR_GROUP_SCHED
  struct sched_entity *parent;
  /* rq on which this entity is (to be) queued: */
  struct cfs_rq   *cfs_rq;
  /* rq "owned" by this entity/group: */
  struct cfs_rq   *my_q;
#endif
};
```

cfs_rq

```c
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight load;                //CFS运行队列的负载权重值
    unsigned int nr_running, h_nr_running; //nr_running: 运行的调度实体数（参与时间片计算）

    u64 exec_clock;     //运行时间
    u64 min_vruntime;  //最少的虚拟运行时间，调度实体入队出队时需要进行增减处理
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif

    struct rb_root_cached tasks_timeline;  //红黑树，用于存放调度实体

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr, *next, *last, *skip; //分别指向当前运行的调度实体、下一个调度的调度实体、CFS运行队列中排最后的调度实体、跳过运行的调度实体

#ifdef    CONFIG_SCHED_DEBUG
    unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
    /*
     * CFS load tracking
     */
    struct sched_avg avg;              //计算负载相关
    u64 runnable_load_sum;
    unsigned long runnable_load_avg;   //基于PELT的可运行平均负载
#ifdef CONFIG_FAIR_GROUP_SCHED
    unsigned long tg_load_avg_contrib; //任务组的负载贡献
    unsigned long propagate_avg;
#endif
    atomic_long_t removed_load_avg, removed_util_avg;
#ifndef CONFIG_64BIT
    u64 load_last_update_time_copy;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    /*
     *   h_load = weight * f(tg)
```

```c
     *
     * Where f(tg) is the recursive weight fraction assigned to this group.
     */
    unsigned long h_load;
    u64 last_h_load_update;
    struct sched_entity *h_load_next;
#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
    struct rq *rq;    /* cpu runqueue to which this cfs_rq is attached */ //指向CFS运行队列所属的CPU RQ运行队列

    /*
     * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
     * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
     * (like users, containers etc.)
     *
     * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
     * list is used during load balance.
     */
    int on_list;
    struct list_head leaf_cfs_rq_list;
    struct task_group *tg;    /* group that "owns" this runqueue */ //CFS运行队列所属的任务组

#ifdef CONFIG_CFS_BANDWIDTH
    int runtime_enabled;    //CFS运行队列中使用CFS带宽控制
    u64 runtime_expires;    //到期的运行时间
    s64 runtime_remaining; //剩余的运行时间

    u64 throttled_clock, throttled_clock_task; //限流时间相关
    u64 throttled_clock_task_time;
    int throttled, throttle_count;    //throttled: 限流, throttle_count: CFS运行队列限流次数
    struct list_head throttled_list; //运行队列限流链表节点，用于添加到cfs_bandwidth结构中的cfttle_cfs_rq链表中
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

runtime与vruntime

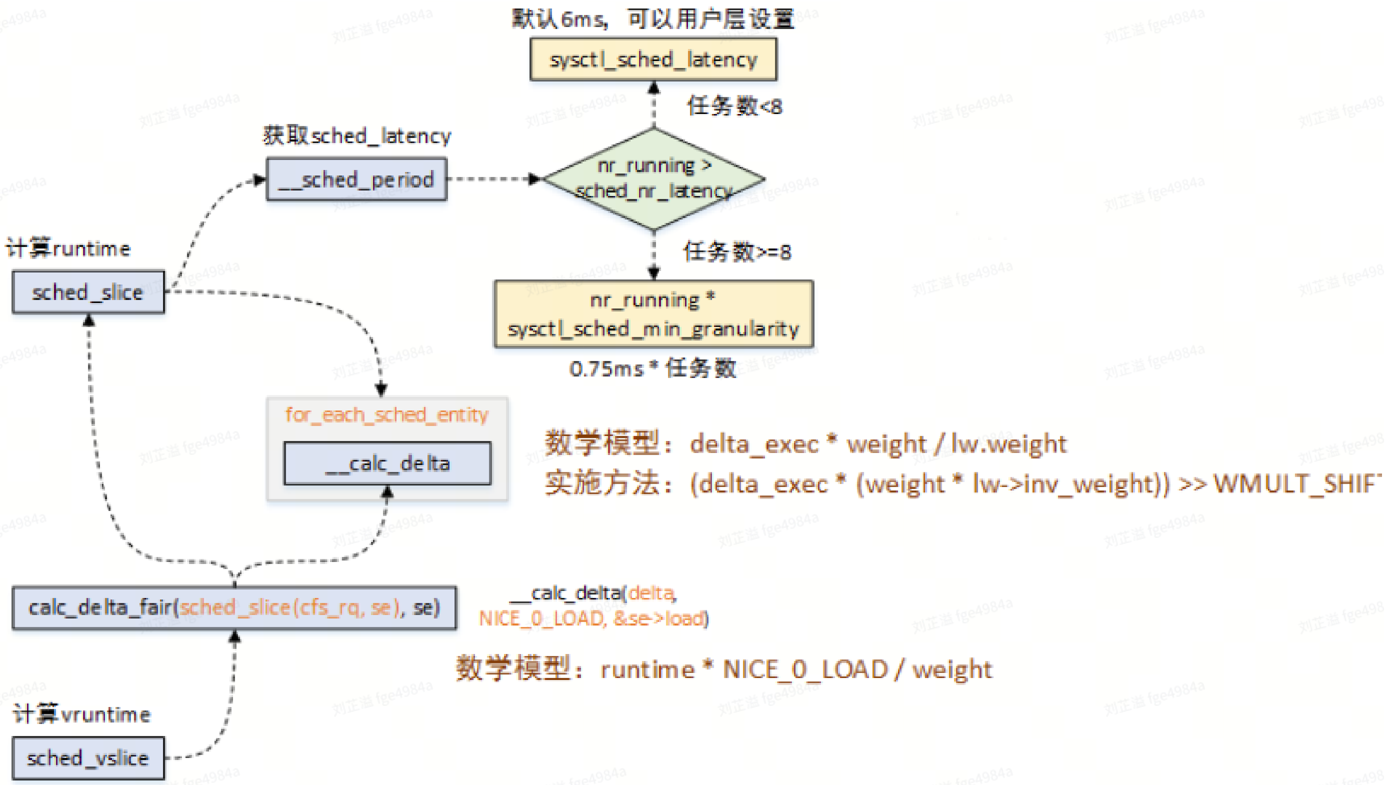
1. Linux 内核默认的 sysctl_sched_latency 是 6ms，这个值用户态可设。sched_period 用于保证可运行任务都能至少运行一次的时间间隔；

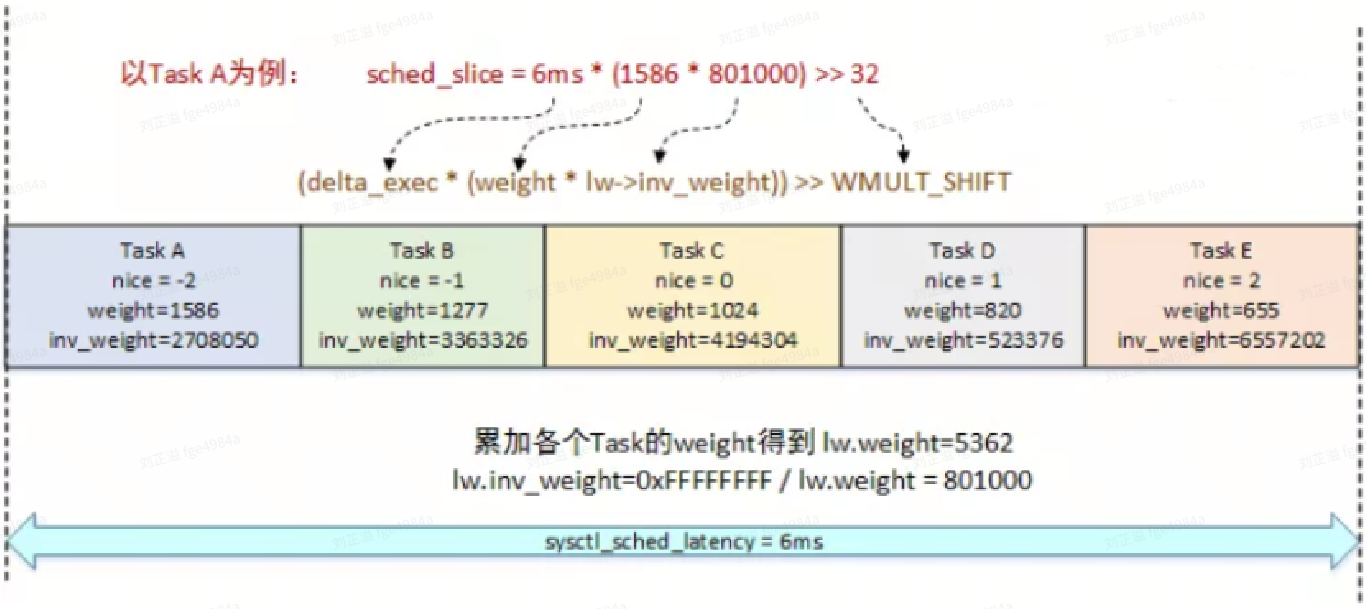
2. 当可运行任务大于 8 个的时候，sched_period 的计算则需要根据任务个数乘以最小调度颗粒值，这个值系统默认为 0.75ms;
3. 每个任务的运行时间计算，是用 sched_period 值，去乘以该任务在整个 CFS 运行队列中的权重占比;
4. 虚拟运行的时间 = 实际运行时间 * NICE_0_LOAD / 该任务的权重;

函数调用

```
/*
 * delta /= w
 */
static inline unsigned long
calc_delta_fair(unsigned long delta, struct sched_entity *se)
{
  if (unlikely(se->load.weight != NICE_0_LOAD))
    delta = calc_delta_mine(delta, NICE_0_LOAD, &se->load);

  return delta;
}

/*
 * The idea is to set a period in which each task runs once.
 *
 * When there are too many tasks (sysctl_sched_nr_latency) we have to stretch
 * this period because otherwise the slices get too small.
 *
 * p = (nr <= nl) ? l : l*nr/nl
 */
static u64 __sched_period(unsigned long nr_running)
{
  u64 period = sysctl_sched_latency;
  unsigned long nr_latency = sched_nr_latency;

  if (unlikely(nr_running > nr_latency)) {
    period = sysctl_sched_min_granularity;
    period *= nr_running;
  }

  return period;
}

/*
 * We calculate the wall-time slice from the period by taking a part
 * proportional to the weight.
```

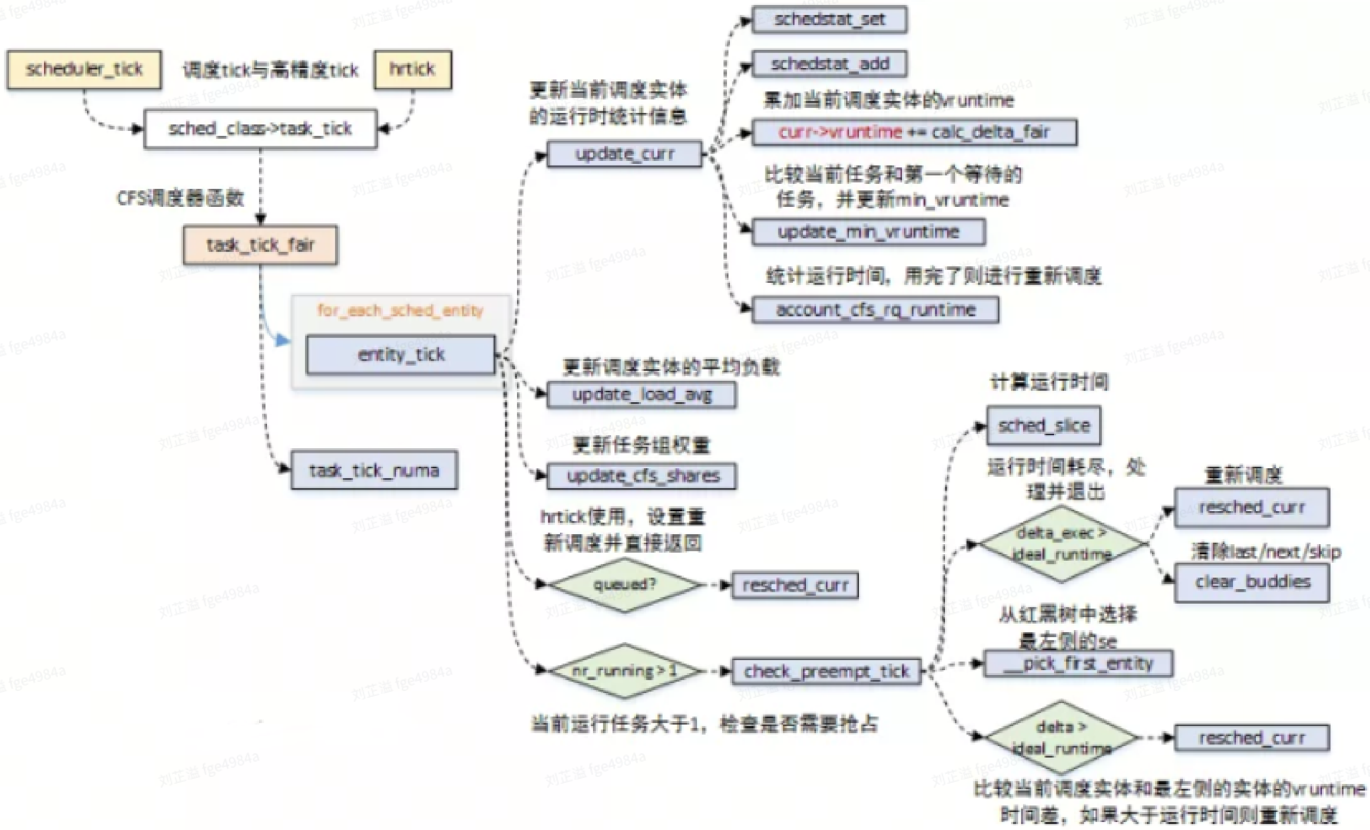
```
 *
 * s = p*P[w/rw]
 */
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
  u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);

  for_each_sched_entity(se) {
    struct load_weight *load;
    struct load_weight lw;

    cfs_rq = cfs_rq_of(se);
    load = &cfs_rq->load;

    if (unlikely(!se->on_rq)) {
      lw = cfs_rq->load;

      update_load_add(&lw, se->load.weight);
      load = &lw;
    }
    slice = calc_delta_mine(slice, se->load.weight, load);
  }
  return slice;
}

/*
 * We calculate the vruntime slice of a to be inserted task
 *
 * vs = s/w
 */
static u64 sched_vslice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
  return calc_delta_fair(sched_slice(cfs_rq, se), se);
}
```

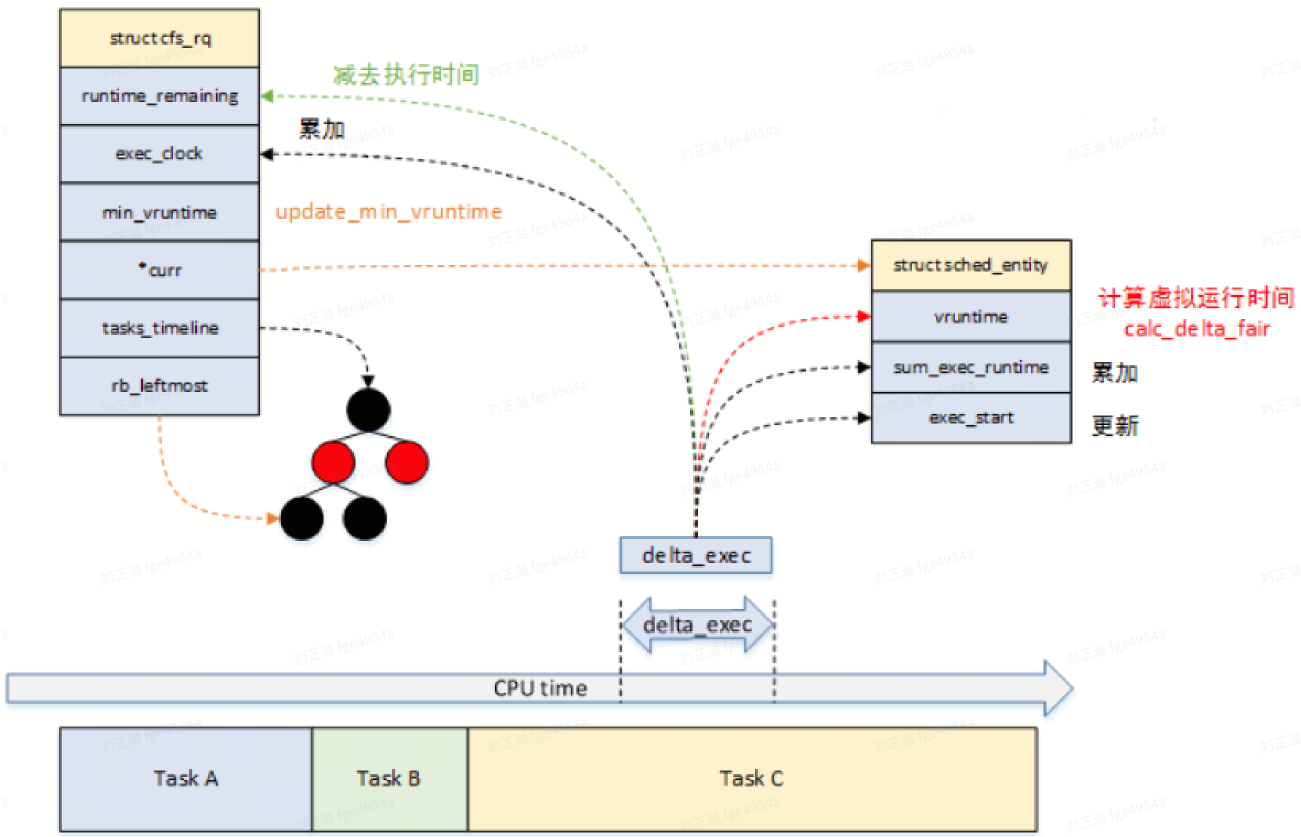




CFS调度tick

- task_tick_fair->entity->tick->update_curr->_update_curr






```
/*
 * scheduler tick hitting a task of our scheduling class:
 */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int
queued)
```

```
{
  struct cfs_rq *cfs_rq;
  struct sched_entity *se = &curr->se;

  for_each_sched_entity(se) {
    cfs_rq = cfs_rq_of(se);
    entity_tick(cfs_rq, se, queued);
  }
}

static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
  /*
   * Update run-time statistics of the 'current'.
   */
  update_curr(cfs_rq);

#ifdef CONFIG_SCHED_HRTICK
  /*
   * queued ticks are scheduled to match the slice, so don't bother
   * validating it and just reschedule.
   */
  if (queued) {
    resched_task(rq_of(cfs_rq)->curr);
    return;
  }
  /*
   * don't let the period tick interfere with the hrtick preemption
   */
  if (!sched_feat(DOUBLE_TICK) &&
      hrtimer_active(&rq_of(cfs_rq)->hrtick_timer))
    return;
#endif

  if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
    check_preempt_tick(cfs_rq, curr);
}

/*
 * Update the current task's runtime statistics. Skip current tasks that
 * are not in our scheduling class.
 */
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
        unsigned long delta_exec)
{
```

```c
	unsigned long delta_exec_weighted;

	schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));

	curr->sum_exec_runtime += delta_exec;
	schedstat_add(cfs_rq, exec_clock, delta_exec);
	delta_exec_weighted = calc_delta_fair(delta_exec, curr);

	curr->vruntime += delta_exec_weighted;
	update_min_vruntime(cfs_rq);
}


static void update_curr(struct cfs_rq *cfs_rq)
{
	struct sched_entity *curr = cfs_rq->curr;
	u64 now = rq_of(cfs_rq)->clock;
	unsigned long delta_exec;

	if (unlikely(!curr))
		return;

	/*
	 * Get the amount of time the current task was running
	 * since the last time we changed load (this cannot
	 * overflow on 32 bits):
	 */
	delta_exec = (unsigned long)(now - curr->exec_start);
	if (!delta_exec)
		return;

	__update_curr(cfs_rq, curr, delta_exec);
	curr->exec_start = now;

	if (entity_is_task(curr)) {
		struct task_struct *curtask = task_of(curr);

		trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
		cpuacct_charge(curtask, delta_exec);
		account_group_exec_runtime(curtask, delta_exec);
	}
}
```

任务出队入队

1. 当任务进入可运行状态时，需要将调度实体放入到红黑树中，完成入队操作；

2. 当任务退出可运行状态时，需要将调度实体从红黑树中移除，完成出队操作；
3. CFS 调度器，使用 **enqueue_entity 函数将任务入队到 CFS 队列，使用 dequeue_entity 函数将任务从 CFS 队列中出队操作**

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
  /*
   * Update the normalized vruntime before updating min_vruntime
   * through callig update_curr().
   */
  if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATE))
    se->vruntime += cfs_rq->min_vruntime;

  /*
   * Update run-time statistics of the 'current'.
   */
  update_curr(cfs_rq);
  account_entity_enqueue(cfs_rq, se);

  if (flags & ENQUEUE_WAKEUP) {
    place_entity(cfs_rq, se, 0);
    enqueue_sleeper(cfs_rq, se);
  }

  update_stats_enqueue(cfs_rq, se);
  check_spread(cfs_rq, se);
  if (se != cfs_rq->curr)
    __enqueue_entity(cfs_rq, se);
}

/*
 * Enqueue an entity into the rb-tree:
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
  struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
  struct rb_node *parent = NULL;
  struct sched_entity *entry;
  s64 key = entity_key(cfs_rq, se);
  int leftmost = 1;

  /*
   * Find the right place in the rbtree:
   */
```

```c
  while (*link) {
    parent = *link;
    entry = rb_entry(parent, struct sched_entity, run_node);
    /*
     * We dont care about collisions. Nodes with
     * the same key stay together.
     */
    if (key < entity_key(cfs_rq, entry)) {
      link = &parent->rb_left;
    } else {
      link = &parent->rb_right;
      leftmost = 0;
    }
  }

  /*
   * Maintain a cache of leftmost tree entries (it is frequently
   * used):
   */
  if (leftmost)
    cfs_rq->rb_leftmost = &se->run_node;

  rb_link_node(&se->run_node, parent, link);
  rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

寻找下一个任务

- __pick_next_entity（）：返回红黑树最左侧节点，该节点是vruntime最小的节点
- 使用rb_leftmost缓存最小vruntime节点

```c
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
  struct rb_node *left = cfs_rq->rb_leftmost;

  if (!left)
    return NULL;

  return rb_entry(left, struct sched_entity, run_node);
}
```