CFS算法基本思想

1 CFS调度算法思想概述

CFS的主要思想如下：

- 根据普通进程的优先级nice值来定一个比重(weight)，该比重用来计算进程的实际运行时间到虚拟运行时间(vruntime)的换算；不言而喻优先级高的进程运行更多的时间和优先级低的进程运行更少的时间在vruntime上是等价的；
- 根据rq->cfs_rq中进程的数量计算一个总的period周期，每个进程再根据自己的weight占整个的比重来计算自己的理想运行时间(ideal_runtime)，在scheduler_tick()中判断如果进程的实际运行时间(exec_runtime)已经达到理想运行时间(ideal_runtime)，则进程需要被调度test_tsk_need_resched(curr)。有了period，那么cfs_rq中所有进程在period以内必会得到调度；
- 与此同时,设定一个调度周期（*sched_latency_ns*），目标是让每个进程在这个周期内至少有机会运行一次，换一种说法就是每个进程等待CPU的时间最长不超过这个调度周期
- *另一个参数sched_min_granularity_ns*发挥作用的一个场景是，CFS把调度周期sched_latency按照进程的数量平分，给每个进程平均分配CPU时间片（当然要按照 nice值加权,即根据weight），但是如果进程数量太多的话，就会造成

CPU时间片太小，如果小于 sched_min_granularity_ns的话就以sched_min_granularity_ns为准；而调度周期也随之不再遵守sched_latency_ns，而是以 (sched_min_granularity_ns * 进程数量) 的乘积为准。

- *参数sched_min_granularity_ns*发挥作用的另一个场景是:参数限定了一个唤醒进程要抢占当前进程之前必须满足的条件：只有当该唤醒进程的vruntime比当前进程的vruntime小、并且两者差距 (vdiff)大于sched_wakeup_granularity_ns的情况下，才可以抢占，否则不可以。这个参数越大，发生唤醒抢占就越不容易。
- 根据进程的虚拟运行时间(vruntime)，把rq->cfs_rq中的进程组织成一个红黑树(平衡二叉树)，那么在pick_next_entity时树的最左节点就是运行时间最少的进程，是最好的需要调度的候选人；

既然task是通过vruntime时间来组织红黑树，并且调度算法也是通过最左边的叶子节点(最小的vruntime)来调度task的。我们先看看vruntime怎么计算？

2 vruntime怎么计算的

每个进程的vruntime = runtime * (NICE_0_LOAD/nice_n_weight)
我们能够看到优先级对应的权重设定如下：
nice(0)=NICE_0_LOAD = 1024，nice(1)=nice(0)/1.25,nice(-1)=nice(0)*1.25，如下表所示：

```
/* 该表的主要思想是，高一个等级的weight是低一个等级的 1.25 倍 */
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int 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,
};
```

NICE_0_LOAD(1024)在schedule计算中是一个非常神奇的数字，它的含义就是基准"1"。因为kernel不能表示小数，所以把1放大称为1024。

TICK_NSEC周期更新vruntime方式如下：

scheduler_tick---->task_tick_fair---->enqueue_tick----->update_curr，如下：

```
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;

    sched_clock_tick();

    raw_spin_lock(&rq->lock);
    /*设置rq的window_start时间*/
    walt_set_window_start(rq);
    /*更新task的runnable time, pre runnable time以及当前task的demand时间*/
    walt_update_task_ravg(rq->curr, rq, TASK_UPDATE,
            walt_ktime_clock(), 0);
    update_rq_clock(rq);  /*更新rq运行时间*/
#ifdef CONFIG_INTEL_DWS
    if (sched_feat(INTEL_DWS))
        update_rq_runnable_task_avg(rq);
#endif
    /*task 每个周期内vruntime虚拟运行时间的更新*/
    curr->sched_class->task_tick(rq, curr, 0);
    /*load是作为负载均衡使用的，后面会单独讲load的衰减和计算*/
    update_cpu_load_active(rq);  /*更新cpu load*/
    calc_global_load_tick(rq);  /*更新系统负载*/
    raw_spin_unlock(&rq->lock);

    perf_event_task_tick();

#ifdef CONFIG_SMP
    rq->idle_balance = idle_cpu(cpu);
    trigger_load_balance(rq);
#endif
    rq_last_tick_reset(rq);

    if (curr->sched_class == &fair_sched_class)
        check_for_migration(rq, 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;
    struct sched_domain *sd;
    /*这是计算se（调度实体）的 vruntime*/
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }

```

```
    if (static_branch_unlikely(&sched_numa_balancing))
        task_tick_numa(rq, curr);

#ifdef CONFIG_64BIT_ONLY_CPU
    trace_sched_load_per_bitwidth(rq->cpu, weighted_cpuload(rq->cpu),
        weighted_cpuload_32bit(rq->cpu));
#endif

#ifdef CONFIG_SMP
    rq->misfit_task = !task_fits_max(curr, rq->cpu);
    rcu_read_lock();
    sd = rcu_dereference(rq->sd);
    /*更加cpu是否过载，rq里面是否有Misfit task来判断sched domain是否过载，具体怎么
    判断过载有专门文章讲过*/
    if (sd) {
        if (cpu_overutilized(task_cpu(curr)))
            set_sd_overutilized(sd);

        if (rq->misfit_task && sd->parent)
            set_sd_overutilized(sd->parent);
    }
    rcu_read_unlock();
#endif

}

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);   /*更新运行时间和虚拟运行时间*/

    /*
     * Ensure that runnable average is periodically updated.
     */
    update_load_avg(curr, UPDATE_TG);   /*更新entity的load,PELT算法*/
    update_cfs_shares(curr);
…………………………………….
    /*当rq里面处于runnable的task超过一个时候，check是否需要调度。*/
    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
}
/*计算vruntime*/
/*
 * Update the current task's runtime statistics.
 */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;
    /*curr运行时间*/
    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;
    /*重新标记运行启动时间，方便下个tick，计算运行时间*/
    curr->exec_start = now;

    schedstat_set(curr->statistics.exec_max,
              max(delta_exec, curr->statistics.exec_max));
    /*累加的运行时间*/
```

```
    curr->sum_exec_runtime += delta_exec;
    /*更新proc/schedstat数据*/
    schedstat_add(cfs_rq, exec_clock, delta_exec);
    /*每个tick周期, 更新每一个task的vruntime*/
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);

    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);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}

/*
 * delta /= w
 */
/*计算虚拟运行时间*/
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

    return delta;
}

```

上面仅仅是根据实际的运行时间, 在根据进程的权重更新进程的虚拟运行时间,那么系统具体是怎么根据权重来计算vruntime的?

我们知道:

```
static void set_load_weight(struct task_struct *p)
{    /*获取task的优先级*/
     int prio = p->static_prio - MAX_RT_PRIO;
     struct load_weight *load = &p->se.load;

     /*
      * SCHED_IDLE tasks get minimal weight:
      *//*设置idle thread的优先级权重*/
     if (idle_policy(p->policy)) {
         load->weight = scale_load(WEIGHT_IDLEPRIO);
         load->inv_weight = WMULT_IDLEPRIO;
         return;
     }
     /*设置正常进程优先级的权重,*/
     load->weight = scale_load(prio_to_weight[prio]);
     /*进程权重的倒数,数值为2^32/weight*/
     load->inv_weight = prio_to_wmult[prio];
     /*上面两个数值都可以通过查表获取的*/
}
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
```

```
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int 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,
};

/*
 * Inverse (2^32/x) values of the prio_to_weight[] array, precalculated.
 *
 * In cases where the weight does not change often, we can use the
 * precalculated inverse to speed up arithmetics by turning divisions
 * into multiplications:
 */  /*进程权重的倒数,数值为2^32/weight*/
static const u32 prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
/*计算虚拟运行时间*/
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

    return delta;
}
/*
 * delta_exec * weight / lw.weight
 *   OR
 * (delta_exec * (weight * lw->inv_weight)) >> WMULT_SHIFT
 *
 * Either weight := NICE_0_LOAD and lw \e prio_to_wmult[], in which case
 * we're guaranteed shift stays positive because inv_weight is guaranteed to
 * fit 32 bits, and NICE_0_LOAD gives another 10 bits; therefore shift >=
22.
 *
 * Or, weight =< lw.weight (because lw.weight is the runqueue weight), thus
 * weight/lw.weight <= 1, and therefore our shift will also be positive.
 */
```

```
static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct load_weight *lw)
{
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;

    __update_inv_weight(lw);

    if (unlikely(fact >> 32)) {
        while (fact >> 32) {
            fact >>= 1;
            shift--;
        }
    }

    /* hint to use a 32x32->64 mul */
    fact = (u64)(u32)fact * lw->inv_weight;

    while (fact >> 32) {
        fact >>= 1;
        shift--;
    }
    return mul_u64_u32_shr(delta_exec, fact, shift);
}
```

可以知道核心计算函数是__calc_delta,是怎么计算的.我们知道
vruntime = runtime * (NICE_0_LOAD/nice_n_weight)
= (runtime*NICE_0_LOAD*inv_weight) >> shift
=(runtime * NICE_0_LOAD * 2^32 / nice_n_weight) >> 2^32即为__calc_delta函数的精髓所在.
由于2^32/nice_n_weight预先做了计算并生成了计算table即prio_to_wmult[]
既然知道了vruntime的数值，那么scheduler是如何来决定是否调度某个task的？

3 理想运行时间/period

period目的是计算一段时间内，各个task的可以运行的时间，即分配的时间片。在cfs_rq里面的分配的时间片是根据task的weight计算得来的。
在上面的代码分析过程中，我们看到，如果cfs_rq的nr_running>1，则检测是否有task需要调度：scheduler_tick--->task_tick_fair--->entity_tick--->check_preempt_tick:

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;
    struct sched_entity *se;
    s64 delta;
    /*计算period和计算ideal_runtime*/
```

```
        ideal_runtime = sched_slice(cfs_rq, curr);
        /*计算实际运行时间*/
        delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
       /*  如果实际运行时间已经超过ideal_time,
              当前进程需要被调度, 设置TIF_NEED_RESCHED标志
         */
        if (delta_exec > ideal_runtime) {
            resched_curr(rq_of(cfs_rq));
            /*
             * The current task ran long enough, ensure it doesn't get
             * re-elected due to buddy favours.
             */
            clear_buddies(cfs_rq, curr);
            return;
        }

        /*
         * Ensure that a task that missed wakeup preemption by a
         * narrow margin doesn't have to wait for a full slice.
         * This also mitigates buddy induced latencies under load.
         */ /*运行时间小于最小颗粒度保证, 直接返回, 此task继续运行*/
        if (delta_exec < sysctl_sched_min_granularity)
            return;
        /*挑选红黑树最左边的节点进行运行*/
        se = __pick_first_entity(cfs_rq);
        delta = curr->vruntime - se->vruntime;
        /*判断当前task与pick的task的虚拟运行时间的大小*/
        if (delta < 0)
            return;
         /*这个没搞明白为何是delta去比较???、*/
         if (delta > ideal_runtime)
             resched_curr(rq_of(cfs_rq));
    }
```

sched_slice是怎么计算ideal_runtime的?

```
    /*
     * 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(slice, se->load.weight, load);
    }

    return slice;
}
/*
 * The idea is to set a period in which each task runs once.
 *
 * When there are too many tasks (sched_nr_latency) we have to stretch
 * this period because otherwise the slices get too small.
 *
 * p = (nr <= nl) ? l : l*nr/nl
 */  /*如果cfs_rq队列的nr_running的数量大于8个，则sched_period时间变成nr_running
  *0.75ms,否则为6ms，nr_running数量一般很少超过8个，反正我没有见过*/
static u64 __sched_period(unsigned long nr_running)
{
    if (unlikely(nr_running > sched_nr_latency))
        return nr_running * sysctl_sched_min_granularity;
    else
        return sysctl_sched_latency;
}
/*
 * Targeted preemption latency for CPU-bound tasks:
 * (default: 6ms * (1 + ilog(ncpus)), units: nanoseconds)
 *
 * NOTE: this latency value is not the same as the concept of
 * 'timeslice length' - timeslices in CFS are of variable length
 * and have no persistent notion like in traditional, time-slice
 * based scheduling concepts.
 *
 * (to see the precise effective timeslice length of your workload,
 *  run vmstat and monitor the context-switches (cs) field)
 */
unsigned int sysctl_sched_latency = 6000000ULL;
/*
 * Minimal preemption granularity for CPU-bound tasks:
 * (default: 0.75 msec * (1 + ilog(ncpus)), units: nanoseconds)
 */
unsigned int sysctl_sched_min_granularity = 750000ULL;

/*
 * is kept at sysctl_sched_latency / sysctl_sched_min_granularity
 */
static unsigned int sched_nr_latency = 8;

```

4 进程怎么插入到红黑树的

按照进程的vruntime组成了红黑树，调用路径：

enqueue_task---->enqueue_task_fair--->enqueue_entity---->__enqueue_entity:

```
/*在core.c文件*/
static inline void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
{
    update_rq_clock(rq);
    if (!(flags & ENQUEUE_RESTORE))
        sched_info_queued(rq, p);
#ifdef CONFIG_INTEL_DWS
    if (sched_feat(INTEL_DWS))
        update_rq_runnable_task_avg(rq);
#endif
    p->sched_class->enqueue_task(rq, p, flags);
}

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

    /*
     * Find the right place in the rbtree:
     */
    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.
         */
        /*比较se的vruntime，小的会放到rb tree的rb_left*/
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }

    /*
```

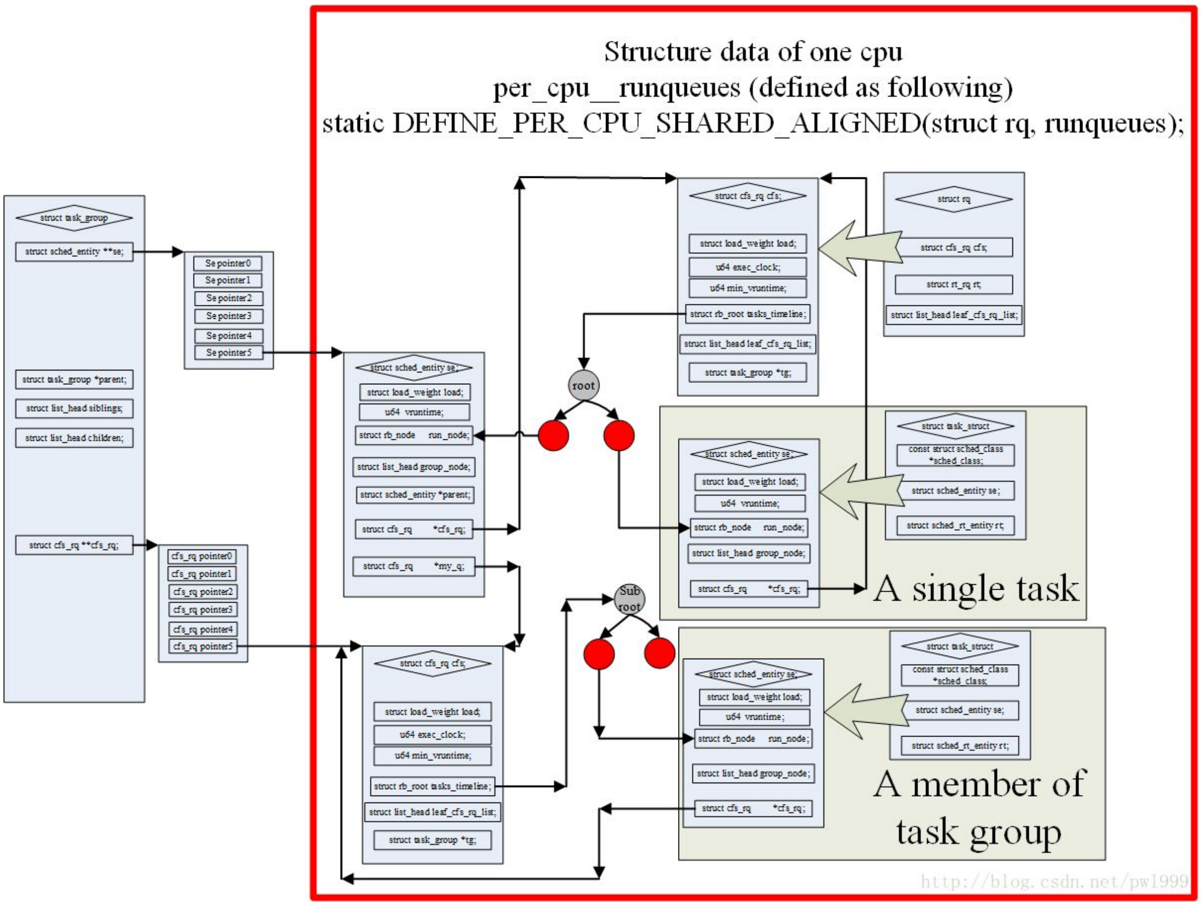
```
        * Maintain a cache of leftmost tree entries (it is frequently
        * used):
        */
       if (leftmost)
           cfs_rq->rb_leftmost = &se->run_node;

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

根据enqueue task的调度时机决策哪个task插入rb tree中：

- 新创建的进程
- idle进程被wakeup
- task的balance(迁移)
- 改变task的cpu亲和性
- 改变task的优先级
- 将task从一个task group设置到新的task group中，如果改变task的rq的话

一般改变task的行为，都会触发重新计算task的vruntime的数值，也就会导致重新插入到对应cpu上的rq rb tree

5 sched_entity和task group的关系



因为新的内核加入了task_group的概念，所以现在不是使用task_struct结构直接参与到schedule计算当中，而是使用sched_entity结构。一个sched_entity结构可能是一个task也可能是一个task_group->se[cpu]。上图非常好的描述了这些结构之间的关系。

其中主要的层次关系如下：

- 一个cpu只对应一个rq;
- 一个rq有一个cfs_rq；
- cfs_rq使用红黑树组织多个同一层级的sched_entity；
- 如果sched_entity对应的是一个task_struct，那sched_entity和task是一对一的关系；
- 如果sched_entity对应的是task_group，那么他是一个task_group多个sched_entity中的一个。task_group有一个数组se[cpu]，在每个cpu上都有一个sched_entity。这种

类型的sched_entity有自己的cfs_rq，一个sched_entity对应一个cfs_rq(se->my_q)，cfs_rq再继续使用红黑树组织多个同一层级的sched_entity；3-5的层次关系可以继续递归下去。

6 几个特殊时刻vruntime的变化

处了常规的scheduler_tick根据tickless更新vruntime，更加进程的状态，调度时机，都会改变vruntime数值。

6.1 新创建的进程vruntime是多少？

假如新进程的vruntime初值为0的话，比老进程的值小很多，那么它在相当长的时间内都会保持抢占CPU的优势，老进程就要饿死了，这显然是不公平的。

CFS的做法是：取父进程vruntime(curr->vruntime) 和 (cfs_rq->min_vruntime + 假设se运行过一轮的值)之间的最大值，赋给新创建进程。把新进程对现有进程的调度影响降到最小。

```
_do_fork() -> copy_process() -> sched_fork() -> task_fork_fair()-
>place_entity:
/*
 * called on fork with the child task as argument from the parent's context
 *  - child not yet on the tasklist
 *  - preemption disabled
 */
static void task_fork_fair(struct task_struct *p)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se, *curr;
    struct rq *rq = this_rq();

    raw_spin_lock(&rq->lock);
    update_rq_clock(rq);

    cfs_rq = task_cfs_rq(current);
    curr = cfs_rq->curr;
    /*如果curr存在，则新的task的vruntime为curr的vruntime，即继承父亲的vruntime*/
    if (curr) {
        update_curr(cfs_rq);
        se->vruntime = curr->vruntime;
    }
```

```
        /*更新新的task 调度实体se的vruntime, 下面讲解*/
        place_entity(cfs_rq, se, 1);
         /* 如果sysctl_sched_child_runs_first标志被设置,
            确保fork子进程比父进程先执行*/
        if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
            /*
             * Upon rescheduling, sched_class::put_prev_task() will place
             * 'current' within the tree based on its new key value.
             */
            swap(curr->vruntime, se->vruntime);
            resched_curr(rq);
        }
          /* 防止新进程运行时是在其他cpu上运行的,
            这样在加入另一个cfs_rq时再加上另一个cfs_rq队列的min_vruntime值即可
            (具体可以看enqueue_entity函数)
         */
        se->vruntime -= cfs_rq->min_vruntime;
        raw_spin_unlock(&rq->lock);
    }

    /*根据进程的状态, 重新计算进程的vruntime时间*/
    static void
    place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
    {
        u64 vruntime = cfs_rq->min_vruntime;
    //     printk("[samarxie] min_vruntime = %llu\n",vruntime);
        /*
         * The 'current' period is already promised to the current tasks,
         * however the extra weight of the new task will slow them down a
         * little, place the new task so that it fits in the slot that
         * stays open at the end.
         */
        if (initial && sched_feat(START_DEBIT))
            vruntime += sched_vslice(cfs_rq, se);
    //     printk("[samarxie] vruntime =
    %llu,load_weight=%lu\n",vruntime,se->load.weight);

        /* sleeps up to a single latency don't count.task flag为ENQUEUE_WAKEUP
        则initial=0 */
        if (!initial) {
            unsigned long thresh = sysctl_sched_latency;

            /*
             * Halve their sleep time's effect, to allow
             * for a gentler effect of sleepers:
             */ /*睡眠时间的影响减半, 目的对休眠者产生温和的影响,相当于衰减*/
            if (sched_feat(GENTLE_FAIR_SLEEPERS))
                thresh >>= 1;

            vruntime -= thresh;
        }
      /*在 (curr->vruntime) 和 (cfs_rq->min_vruntime + 假设se运行过一轮的值),
        之间取最大值 */
        /* ensure we never gain time by being placed backwards. */
        se->vruntime = max_vruntime(se->vruntime, vruntime);
    }
    /*在task_fork_fair创建进程的时候减去了对应cfs_rq的vruntime,但是在入队列的时候,
    重新加上了对应的数值, 避免进程在不同cpu上的min_vruntime不同导致的问题, 真的很巧妙啊*/
    static void
    enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
    {
        /*
         * Update the normalized vruntime before updating min_vruntime
         * through calling update_curr().
```

```
         */
        /* 在enqueue时给se->vruntime重新加上cfs_rq->min_vruntime */
        if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
            se->vruntime += cfs_rq->min_vruntime;

    }
```

6.2 休眠进程的vruntime一直保持不变吗？

如果休眠进程的 vruntime 保持不变，而其他运行进程的 vruntime 一直在推进，那么等到休眠进程终于唤醒的时候，它的vruntime比别人小很多，会使它获得长时间抢占CPU的优势，其他进程就要饿死了。这显然是另一种形式的不公平。

CFS是这样做的：在休眠进程被唤醒时重新设置vruntime值，以min_vruntime值为基础，给予一定的补偿，但不能补偿太多。

```
    static void
    enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
    {

        if (flags & ENQUEUE_WAKEUP) {
            /* (1) 计算进程唤醒后的vruntime */
            place_entity(cfs_rq, se, 0);
            enqueue_sleeper(cfs_rq, se);
        }


    }

    |→

    static void
    place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
    {
        /* (1.1) 初始值是cfs_rq的当前最小值min_vruntime */
        u64 vruntime = cfs_rq->min_vruntime;

        /*
         * The 'current' period is already promised to the current tasks,
         * however the extra weight of the new task will slow them down a
         * little, place the new task so that it fits in the slot that
         * stays open at the end.
         */
        if (initial && sched_feat(START_DEBIT))
            vruntime += sched_vslice(cfs_rq, se);

        /* sleeps up to a single latency don't count. */
        /* 在最小值min_vruntime的基础上给予补偿，
            默认补偿值是3ms(sysctl_sched_latency/2) wakeup initial=0
```

```
         */
    if (!initial) {
        unsigned long thresh = sysctl_sched_latency;

        /*
         * Halve their sleep time's effect, to allow
         * for a gentler effect of sleepers:
         */
        if (sched_feat(GENTLE_FAIR_SLEEPERS))
            thresh >>= 1;

        vruntime -= thresh;
    }

    /* ensure we never gain time by being placed backwards. */
    se->vruntime = max_vruntime(se->vruntime, vruntime);
}
```

6.3 休眠进程在唤醒时会立刻抢占CPU吗？

进程被唤醒默认是会马上检查是否库抢占，因为唤醒的vruntime在cfs_rq的最小值min_vruntime基础上进行了补偿，所以他肯定会抢占当前的进程。

CFS可以通过禁止WAKEUP_PREEMPTION来禁止唤醒抢占，不过这也就失去了抢占特性。

```
wake_up_process(p)->try_to_wake_up() -> ttwu_queue() -> ttwu_do_activate() -> ttwu_do_wakeup() -> check_preempt_curr() -> check_preempt_wakeup()

void check_preempt_curr(struct rq *rq, struct task_struct *p, int flags)
{
    const struct sched_class *class;
    /*根据当前调度类来进行抢占设置*/
    if (p->sched_class == rq->curr->sched_class) {
        rq->curr->sched_class->check_preempt_curr(rq, p, flags);
    } else {   /*否则从最高优先级sched_class开始遍历*/
        for_each_class(class) {
            if (class == rq->curr->sched_class)
                break;
            /*如果找到进程的调度类,则设置重新调度*/
            if (class == p->sched_class) {
                resched_curr(rq);
                break;
            }
        }
```

```
    }

    /*
     * A queue event has occurred, and we're going to schedule.  In
     * this case, we can save a useless back to back clock update.
     */
    if (task_on_rq_queued(rq->curr) && test_tsk_need_resched(rq->curr))
        rq_clock_skip_update(rq, true);
}
/*cfs调度类,定义的抢占函数*/
const struct sched_class fair_sched_class = {
.............
    .check_preempt_curr = check_preempt_wakeup,
.............
}

/*
 * Preempt the current task with a newly woken task if needed:
 */
/*主要是一些条件的判决来决策是否需要抢占.*/
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
    struct task_struct *curr = rq->curr;
    struct sched_entity *se = &curr->se, *pse = &p->se;
    struct cfs_rq *cfs_rq = task_cfs_rq(curr);
    /*nr_running 是否大于8*/
    int scale = cfs_rq->nr_running >= sched_nr_latency;
    int next_buddy_marked = 0;

    if (unlikely(se == pse))
        return;

    /*
     * This is possible from callers such as attach_tasks(), in which we
     * unconditionally check_prempt_curr() after an enqueue (which may have
     * lead to a throttle).  This both saves work and prevents false
     * next-buddy nomination below.
     */
    if (unlikely(throttled_hierarchy(cfs_rq_of(pse))))
        return;
```

```
•
•     if (sched_feat(NEXT_BUDDY) && scale && !(wake_flags & WF_FORK)) {
•         set_next_buddy(pse);
•         next_buddy_marked = 1;
•     }
•
•     /*
•      * We can come here with TIF_NEED_RESCHED already set from new task
•      * wake up path.
•      *
•      * Note: this also catches the edge-case of curr being in a throttled
•      * group (e.g. via set_curr_task), since update_curr() (in the
•      * enqueue of curr) will have resulted in resched being set.  This
•      * prevents us from potentially nominating it as a false LAST_BUDDY
•      * below.
•      */
•     if (test_tsk_need_resched(curr))
•         return;
•
•     /* Idle tasks are by definition preempted by non-idle tasks. */
•     if (unlikely(curr->policy == SCHED_IDLE) &&
•         likely(p->policy != SCHED_IDLE))
•         goto preempt;
•
•     /*
•      * Batch and idle tasks do not preempt non-idle tasks (their preemption
•      * is driven by the tick):
•      */  /*是否设置抢占标志位*/
•     if (unlikely(p->policy != SCHED_NORMAL) || !sched_feat(WAKEUP_PREEMPTION))
•         return;
•
•     find_matching_se(&se, &pse);
•     update_curr(cfs_rq_of(se));
•     BUG_ON(!pse);
•     if (wakeup_preempt_entity(se, pse) == 1) {
•         /*
•          * Bias pick_next to pick the sched entity that is
•          * triggering this preemption.
```

```
            */
            if (!next_buddy_marked)
                set_next_buddy(pse);
            goto preempt;
        }

        return;

preempt:
        resched_curr(rq); /*重新调度*/
        /*
         * Only set the backward buddy when the current task is still
         * on the rq. This can happen when a wakeup gets interleaved
         * with schedule on the ->pre_schedule() or idle_balance()
         * point, either of which can * drop the rq lock.
         *
         * Also, during early boot the idle thread is in the fair class,
         * for obvious reasons its a bad idea to schedule back to it.
         */
        if (unlikely(!se->on_rq || curr == rq->idle))
            return;

        if (sched_feat(LAST_BUDDY) && scale && entity_is_task(se))
            set_last_buddy(se);
}
```

所以一般唤醒进程会立即运行,就算你wakup立马sleep,也会先抢占在休眠

6.4 进程迁移到不同cpu的时候，vruntime会变吗？

不同cpu的负载时不一样的，所以不同cfs_rq里se的vruntime水平是不一样的。如果进程迁移vruntime不变也是非常不公平的。CFS使用了一个很聪明的做法：在退出旧的cfs_rq时减去旧cfs_rq的min_vruntime，在加入新的cfq_rq时重新加上新cfs_rq的min_vruntime。

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    ..................
    /*
     * Normalize the entity after updating the min_vruntime because the
```

```
         * update can refer to the ->curr item and we need to reflect this
         * movement in our normalized position.
         */
        /* 退出旧的cfs_rq时减去旧cfs_rq的min_vruntime */
        if (!(flags & DEQUEUE_SLEEP))
            se->vruntime -= cfs_rq->min_vruntime;
    ……………….
}

static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
      ……………….
        /*
         * Update the normalized vruntime before updating min_vruntime
         * through calling update_curr().
         */
        /*  加入新的cfq_rq时重新加上新cfs_rq的min_vruntime */
        if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
            se->vruntime += cfs_rq->min_vruntime;
        ……………….
}
```

至此基本的调度算法怎么挑选一个task,根据vruntime数值插入rb tree和调度se,这部分简单.后续会陆续讲解调度器如何分配task到其他cpu上,即balance