操作系统实验2：CFS代码分析

费扬 519021910917
2022.05.01
2022春 操作系统 lab-2

一、实验题目

根据提供的源代码分析CFS的代码实现原理，比如包括哪些函数？设置了哪些关键参数？执行流程是怎样的？结合教材ppt撰写代码分析报告。

二、实验目的

1. 加深对CFS (Completely Fair Scheduling) 调度的理解。
2. 对Linux和OpenEuler的底层实现进行分析并掌握。

三、背景知识

基本介绍

- cfs定义了一种新的模型，它给cfs_rq（cfs的run queue）中的每一个进程安排一个虚拟时钟: virtual runtime (vruntime)。
- 如果一个进程得以执行，随着时间的增长（也就是一个个tick的到来），其vruntime将不断增大。没有得到执行的进程vruntime不变。
- 而调度器总是选择vruntime跑得最慢的那个进程来执行。这就是所谓的“完全公平”。
- 为了区别不同优先级的进程，优先级高的进程vruntime增长得慢，以至于它可能得到更多的运行机会。

主要思想

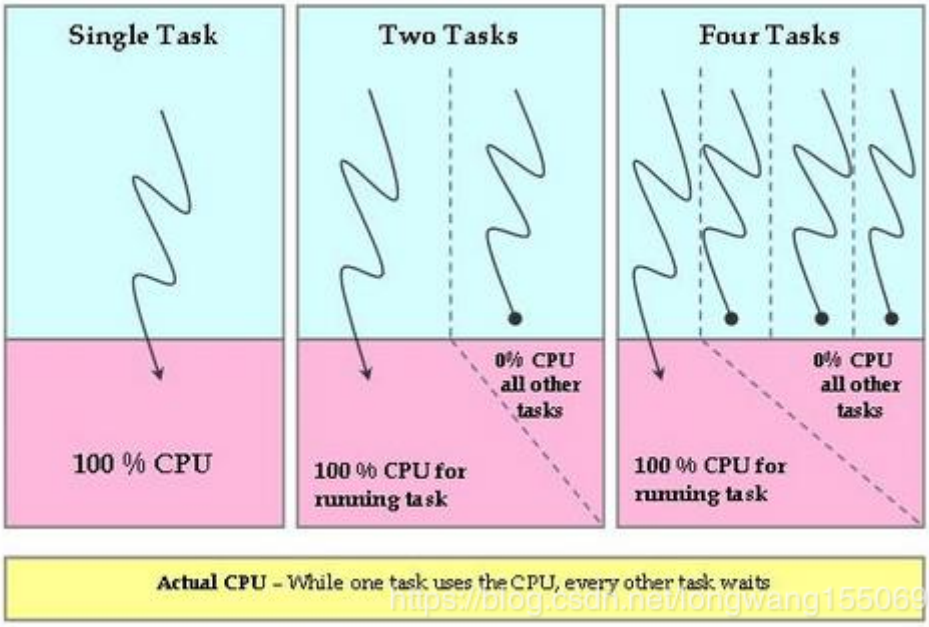
完全公平调度器(CFS)最早是在2017年merged进Linux2.6.23版本中的，一直到现在都是系统中默认的调度器。内核文章中的sched-design-CFS.txt文档对CFS调度器有一个简单的介绍。

> 80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

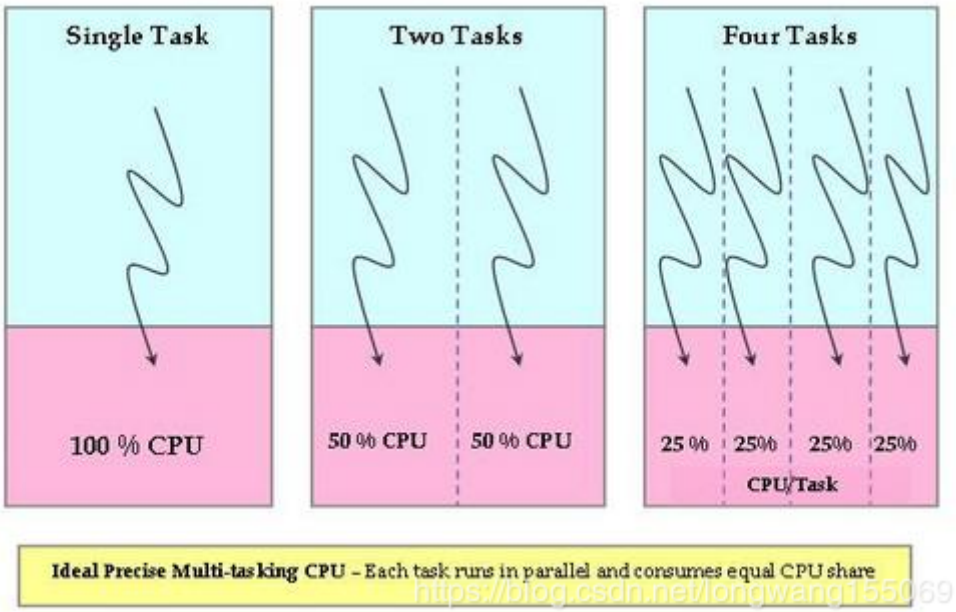
这句话的意思是CFS的80%的设计总结起来就一句话“在一个真实的硬件上，实现公平，精确的多任务CPU”。

> "Ideal multi-tasking CPU" is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 tasks running, then it runs each at 50% physical power --- i.e., actually in parallel.

内核文档是这样说的。"理想的，多任务CPU"是在同一时刻每个任务以1/nr_running_speed来运行，也就是同一时刻每个进程瓜分CPU的时间是相同的。例如如果有两个进程运行的话，每个进程占有50%的CPU时间。
举一个例子:两个批处理进程，总共只能运行10ms。
实际情况：每个进程运行5ms，占有100%的CPU利用率：



理想情况：每个进程运行10ms，占有50%的CPU利用率。



而所谓的理想情况就是CFS提到的"Ideal multi-tasking CPU"。

上述的例子在一个单核CPU，只有一个处理核心的CPU上是完全不可能的，因为同一时刻只能运行一个进程，另外一个进程必须等待。而CFS就是为了的达到完全公平调度，它应该怎么做呢?

实现完全公平调度

在O(n)调度器和O(1)调度器中，我们知道都是通过优先级来分配对应的timeslice,也就是时间片。而这些时间片都是固定的。比如在O(n)调度器中nice0对应的时间片是60ms。而在CFS调度器中，不再有时间片的概念。而是根据当前系统中可运行进程的总数来计算出进程的可运行时间的。

在O(n)调度器和O(1)调度器中，普通进程都是通过nice值来获取对应时间片，nice值越大获取的时间片就越多，运行机会就越多。而在CFS调度器中引入了权重weight的概念，通过nice值转化为对应的权重，优先级越高的进程对应的权重就越大，意味着就可以获得更多的CPU时间。

> 进程占用CPU的时间 = 进程的weight / 总的可运行进程weight

CFS是让进程通过运行一段时间来达到公平，进程占用的时间就是进程的weight占总的可运行进程的总权重的比值。

举例：总共10ms的时间，单核cpu

- **进程的优先级相同：**

如果两个进程的优先级相同，则对应的权重相同，则每个进程占用5ms的CPU时间；如果有5个进程，每个进程占用2ms的CPU时间；如果共10个进程，每个进程占用1ms的CPU时间。

- **进程的优先级不同：**

如果两个进程的优先级不同，比如A进程nice是0，B的nice值1，则A进程的优先级就高，weight就越大，对应B的优先级小，weight也小于A。假设A的权重是6，B的权重是4。则A占2/3的CPU时间，B占1/3的CPU时间。

这样一来就达到了公平性，每个进程在各子拥有的权重比例下，占用不同份额的CPU时间。

设计思路

CFS思路很简单，就是根据各个进程的权重分配运行时间：

进程的运行时间计算公式为:

> 分配给进程的运行时间 = 调度周期 进程权重 / 所有进程权重之和

调度周期很好理解，就是将所有处于TASK_RUNNING态进程都调度一遍的时间,差不多相当于O(1)调度算法中运行队列和过期队列切换一次的时间。

从实际运行时间到vruntime的换算公式：

> vruntime = 实际运行时间 * 1024 (NICE_0_LOAD) / 进程权重

这里直接写为1024，实际上它等于nice为0的进程的权重，代码中是NICE_0_LOAD。也就是说，所有进程都以nice为0的进程的权重1024作为基准，计算自己的vruntime增加速度。

举个例子，比如只有两个进程A, B，权重分别为1和2，调度周期设为30ms，那么分配给A的CPU时间为:30ms * (1/(1+2)) = 10ms；而B的CPU时间为：30ms * (2/(1+2)) = 20ms。那么在这30ms中A将运行10ms，B将运行20ms。而由于B的权重是A的2倍，那么B的vruntime增加速度只有A的一半。
综合上述两个公式可以知道：

> vruntime = (调度周期 * 进程权重 / 所有进程总权重) * 1024 / 进程权重 = 调度周期 * 1024 / 所有进程总权重

四、数据结构

本小节重点在分析CFS调度器中涉及到的一些常见的数据结构，对这些数据结构做一个简单的概括。
以下代码主要摘自于fair.c和sched.h两个文件。

- **linux/fair.c at v5.12 · torvalds/linux (github.com)**
- **linux/sched.h at v5.12 · torvalds/linux (github.com)**

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 {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    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 (*yield_task)   (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

    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_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
    int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags
*rf);
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

    void (*task_woken)(struct rq *this_rq, struct task_struct *task);

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask,
                 u32 flags);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);

    struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
```

```
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serliazed by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    void (*switched_from)(struct rq *this_rq, struct task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);

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

    void (*update_curr)(struct rq *rq);

#define TASK_SET_GROUP      0
#define TASK_MOVE_GROUP     1

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_change_group)(struct task_struct *p, int type);
#endif
};



    /*
     * 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);
```

由于sched_class成员比较多，我们这里先介绍几个常用的。可以看到调度类中基本都是一些函数指针，这些函数指针分别代表的不同的含义。

- 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内核提供的调度类

```
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里进程。

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL		0
#define SCHED_FIFO		1
#define SCHED_RR		2
#define SCHED_BATCH		3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE		5
#define SCHED_DEADLINE		6
```

同时也提供了六种调度策略。下表中表示了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：

```
#ifdef CONFIG_SMP
#define sched_class_highest (&stop_sched_class)
#else
#define sched_class_highest (&dl_sched_class)
#endif
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)
```

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

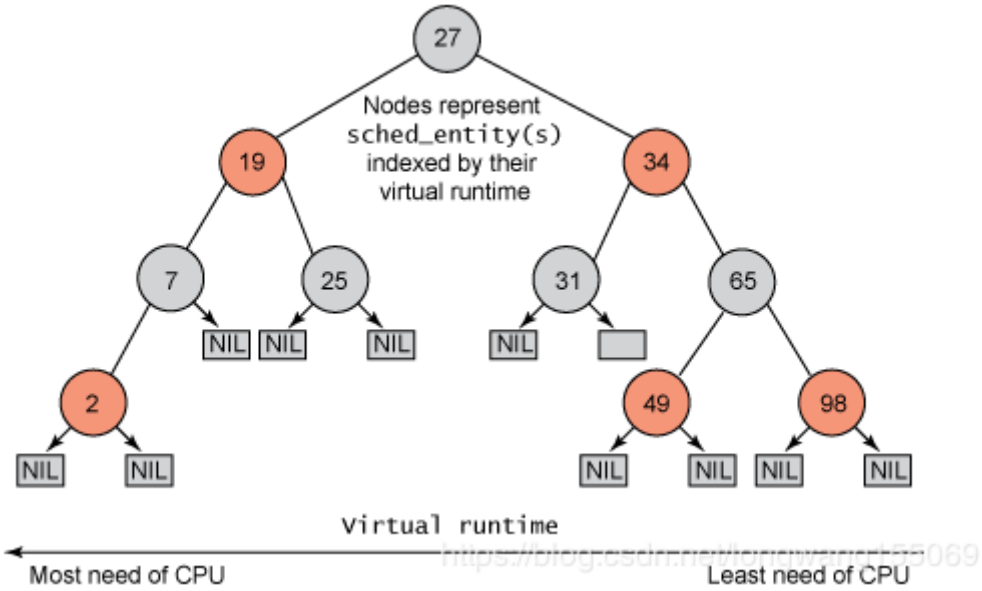
红黑树

红黑树中，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维护了一课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。



即PPT上的这个图示

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;
```

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

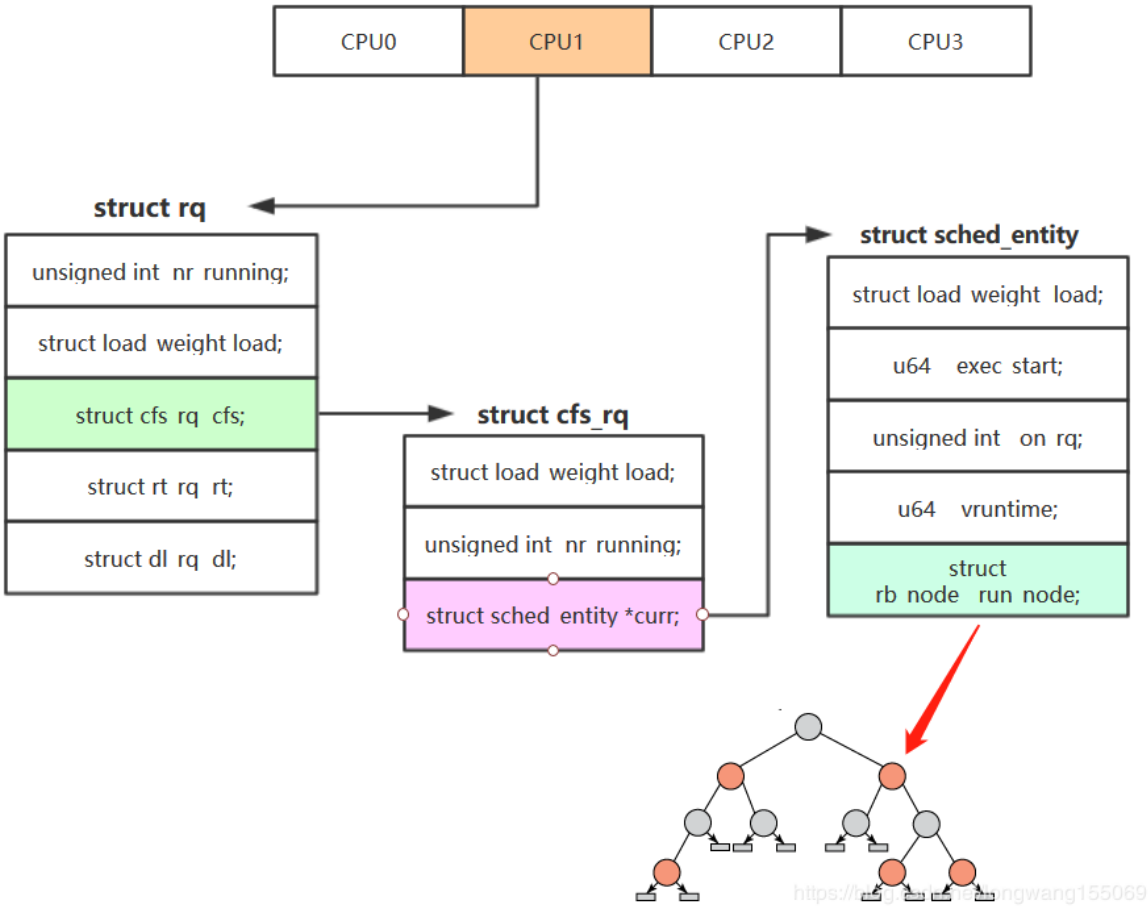
可以得出的一个结论是，一个struct rq中包括了各种类型的进程，有DL的，有实时的，有普通的。通过将不同进程的挂到不同的运行队列中管理。

```
/* 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。

五、主要代码分析

函数分析

在分析代码之前，有一些函数需要先分析，这些函数很重要：

calc_delta_fair

calc_delta_fair函数是用来计算进程的vruntime的函数。在之前CFS原理篇了解了一个进程的vruntime的计算公式。一个进程的vruntime 等于进程实际运行时间 乘以 NICE0进程对应权重 除以 当前进程的权重。为了保证除法计算中不涉及浮点运算，linux内核通过先左移32位，然后右移32位来避免浮点运算。修改后的公式为：

$$vruntime = \frac{delta\_time * NICE\_0\_LOAD * 2^{32}}{weight} >> 32$$

再经过转化成如下的公式：

$$vruntime = (delta\_time * NICE\_0\_LOAD * inv\_weight) >> 32$$

$$inv\_weight = \frac{2^{32}}{weight}$$

其中inv_weight的值内核代码中已经计算好了，在使用的时候只需要通过查表就可以的到Inv_weigth的值：

```
/*
 * Inverse (2^32/x) values of the sched_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:
 */
const u32 sched_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,
};
```

通过上面的计算方式可以轻松的获取一个进程的vruntime。

```
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;
}
```

解释：如果当前调度实体的权重值等于NICE_0_LOAD,则直接返回进程的实际运行时间。因为nice0进程的虚拟时间等于物理时间。否则调用__calc_delta函数计算进程的vruntime。

```
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);
}
```

这段代码可以计算出一个进程的虚拟运行时间vruntime

sched_slice

此函数是用来计算一个调度周期内，一个调度实体可以分配多少运行时间：

```
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;
}
```

__sched_period函数是计算调度周期的函数，此函数当进程个数小于8时，调度周期等于调度延迟等于6ms。否则调度周期等于进程的个数乘以0.75ms，表示一个进程最少可以运行0.75ms，防止进程过快发生上下文切换。

接着就是遍历当前的调度实体，如果调度实体没有调度组的关系，则只运行一次。获取当前CFS运行队列cfs_rq，获取运行队列的权重cfs_rq->rq代表的是这个运行队列的权重。最后通过__calc_delta计算出此进程的实际运行时间。

__calc_delta此函数之前计算虚拟函数的时候介绍过，它不仅仅可以计算一个进程的虚拟时间，它在这里是计算一个进程在总的调度周期中可以获取的运行时间，计算公式见此文内链接公式1。

place_entity

此函数用来惩罚一个调度实体，本质是修改它的vruntime的值：

```
static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
    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. */
    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);
}
```

- 获取当前CFS运行队列的min_vruntime的值；
- 当参数initial等于true时，代表是新创建的进程，新创建的进程则给它的vruntime增加值，代表惩罚它。这是对新创建进程的一种惩罚，因为新创建进程的vruntime过小，防止一直占在CPU；
- 如果inital不为true,则代表的是唤醒的进程，对于唤醒的进程则需要照顾，最大的照顾是调度延时的一半；
- 确保调度实体的vruntime不得倒退，通过max_vruntime获取最大的vruntime。

update_curr

update_curr函数用来更新当前进程的运行时间信息：

```
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;

    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;

    schedstat_set(curr->statistics.exec_max,
              max(delta_exec, curr->statistics.exec_max));

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

    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);


    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

- delta_exec = now - curr->exec_start；计算出当前CFS运行队列的进程，距离上次更新虚拟时间的差值；

- curr->exec_start = now; 更新exec_start的值；
- curr->sum_exec_runtime += delta_exec; 更新当前进程总共执行的时间；
- 通过calc_delta_fair计算当前进程虚拟时间；
- 通过update_min_vruntime函数来更新CFS运行队列中最小的vruntime的值。

新进程创建

通过新进程创建流程，来分析下CFS调度器是如何设置新创建的进程的。在fork创建一个新进程的时候我们涉及到sched模块时是简单的，这里重点分析。

```
int sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    unsigned long flags;

    __sched_fork(clone_flags, p);
    /*
     * We mark the process as NEW here. This guarantees that
     * nobody will actually run it, and a signal or other external
     * event cannot wake it up and insert it on the runqueue either.
     */
    p->state = TASK_NEW;

    /*
     * Make sure we do not leak PI boosting priority to the child.
     */
    p->prio = current->normal_prio;


    if (dl_prio(p->prio))
        return -EAGAIN;
    else if (rt_prio(p->prio))
        p->sched_class = &rt_sched_class;
    else
        p->sched_class = &fair_sched_class;

    init_entity_runnable_average(&p->se);

    raw_spin_lock_irqsave(&p->pi_lock, flags);
    /*
     * We're setting the CPU for the first time, we don't migrate,
     * so use __set_task_cpu().
     */
    __set_task_cpu(p, smp_processor_id());
    if (p->sched_class->task_fork)
        p->sched_class->task_fork(p);
    raw_spin_unlock_irqrestore(&p->pi_lock, flags);

    init_task_preempt_count(p);
    return 0;
}
```

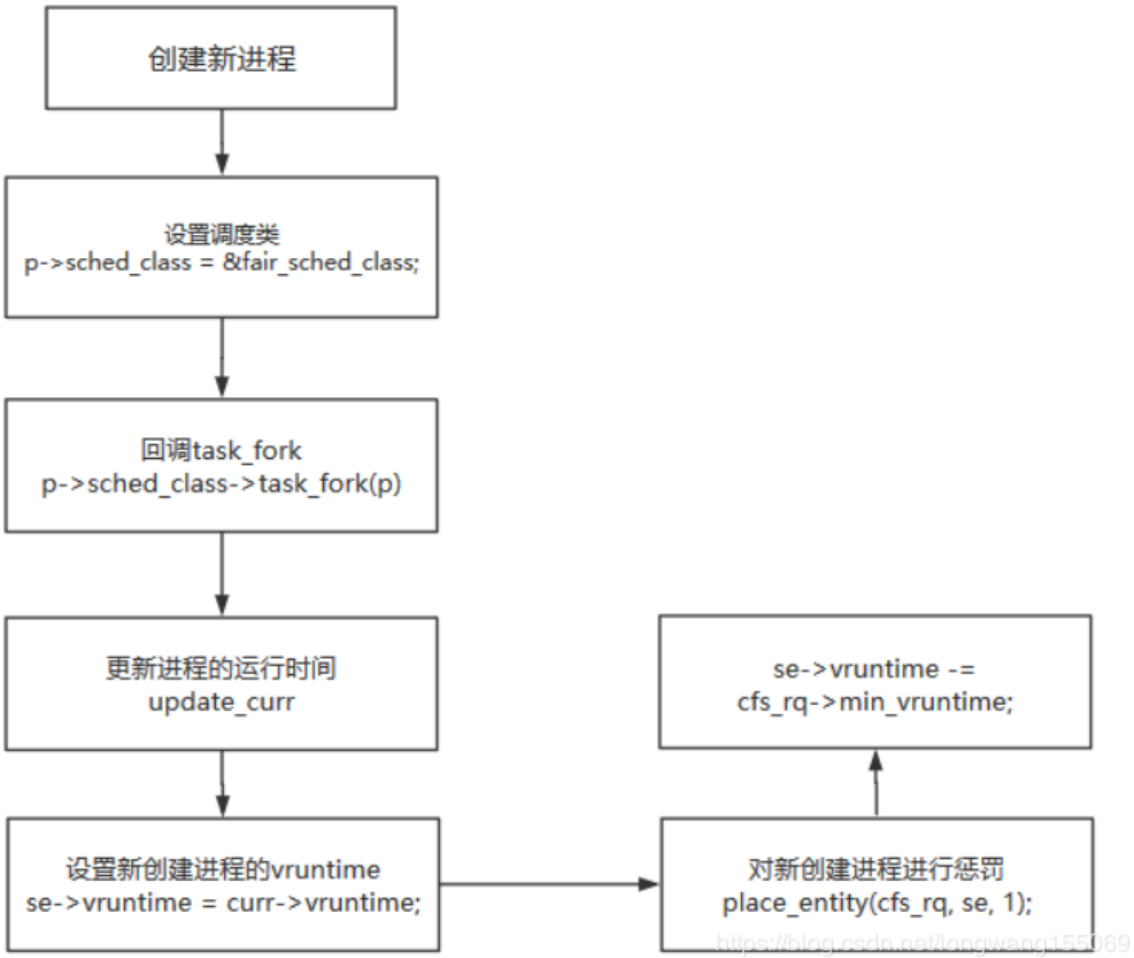
- __sched_fork 主要是初始化调度实体的，此处不用继承父进程，因为子进程会重新运行的，对这些值会进程重新的复制；
- 设置进程的状态为TASK_NEW，代表这是个新进程；
- 将当前current进程的优先级设置给新创建的进程，新创建进程动态优先级p->prio = current->normal_prio；
- 根据进程的优先级设置进程的调度类，如果是RT进程设置调度类为rt_sched_class， 如果是普通进程设置调度类为fair_sched_class；
- 设置当前进程在哪个CPU运行，此处只是简单的设置。在加入调度器的运行队列时还会设置一次；

- 最后调用调度类中的task_fork函数指针，最后会调用到fair_sched_class中的task_fork函数指针。

```
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();
    struct rq_flags rf;

    rq_lock(rq, &rf);
    update_rq_clock(rq);

    cfs_rq = task_cfs_rq(current);
    curr = cfs_rq->curr;
    if (curr) {
        update_curr(cfs_rq);
        se->vruntime = curr->vruntime;
    }
    place_entity(cfs_rq, se, 1);

    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);
    }

    se->vruntime -= cfs_rq->min_vruntime;
    rq_unlock(rq, &rf);
}
```

- 通过current获取当前CFS运行队列，通过运行队列的curr指针获取当前的调度实体，然后通过update_curr更新当前调度实体的运行时间，同时将当前调度实体的虚拟vruntime的值赋值给新创建进程的vruntime；
- 通过place_entity函数对当前新创建的进程进行惩罚，以为第三个参数是1，则会对新创建进程惩罚；
- se->vruntime -= cfs_rq->min_vruntime; 将当前调度实体虚拟运行时间减去min_vruntime，这句可以这样理解，因为在此调度实体添加到运行队列中还有一段时间，在这段时间内min_vruntime的值会改变的。当在添加到运行队列的时候在加上，就显得公平了。

用如下的流程图总结新创建一个进程流程的过程：



将新进程加入到就绪队列中

在fork完一个进程后，会通过wake_up_new_task函数来唤醒一个进程，将新创建的进程加入到就绪队列中。

```
void wake_up_new_task(struct task_struct *p)
{
    struct rq_flags rf;
    struct rq *rq;

    raw_spin_lock_irqsave(&p->pi_lock, rf.flags);
    p->state = TASK_RUNNING;
#ifdef CONFIG_SMP
    /*
     * Fork balancing, do it here and not earlier because:
     *  - cpus_allowed can change in the fork path
     *  - any previously selected CPU might disappear through hotplug
     *
     * Use __set_task_cpu() to avoid calling sched_class::migrate_task_rq,
     * as we're not fully set-up yet.
     */
    p->recent_used_cpu = task_cpu(p);
    __set_task_cpu(p, select_task_rq(p, task_cpu(p), SD_BALANCE_FORK, 0));
#endif
    rq = __task_rq_lock(p, &rf);
    update_rq_clock(rq);
    post_init_entity_util_avg(&p->se);

    activate_task(rq, p, ENQUEUE_NOCLOCK);
    p->on_rq = TASK_ON_RQ_QUEUED;
    trace_sched_wakeup_new(p);
```

```
    check_preempt_curr(rq, p, WF_FORK);
    task_rq_unlock(rq, p, &rf);
}
```

- 设置进程的状态为TASK_RUNNING，代表进程已经处于就绪状态了；
- 如果打开SMP，则会通过__set_task_cpu重新设置一个最优的CPU的，让新进程在其上面运行；
- 最后通过activate_task(rq, p, ENQUEUE_NOCLOCK)；函数将新创建的进程加入到就绪队列。

```
void activate_task(struct rq *rq, struct task_struct *p, int flags)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;

    enqueue_task(rq, p, flags);
}

static inline void enqueue_task(struct rq *rq, struct task_struct *p, int
flags)
{
    if (!(flags & ENQUEUE_NOCLOCK))
        update_rq_clock(rq);

    if (!(flags & ENQUEUE_RESTORE)) {
        sched_info_queued(rq, p);
        psi_enqueue(p, flags & ENQUEUE_WAKEUP);
    }

    p->sched_class->enqueue_task(rq, p, flags);
}
```

最终会调用到CFS调度类中的enqueue_task函数指针。

```
static void
    enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;

    for_each_sched_entity(se) {
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, flags);

        /*
         * end evaluation on encountering a throttled cfs_rq
         *
         * note: in the case of encountering a throttled cfs_rq we will
         * post the final h_nr_running increment below.
         */
        if (cfs_rq_throttled(cfs_rq))
            break;
        cfs_rq->h_nr_running++;

        flags = ENQUEUE_WAKEUP;
    }

}
```

- 如果调度实体的on_rq已经设置，则代表在就绪队列中，直接跳出

- 通过enqueue_entity函数将调度实体入队
- 增加CFS运行队列的可运行个数h_nr_running

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /*
     * If we're the current task, we must renormalise before calling
     * update_curr().
     */
    if (renorm && curr)
        se->vruntime += cfs_rq->min_vruntime;

    update_curr(cfs_rq);

    /*
     * Otherwise, renormalise after, such that we're placed at the current
     * moment in time, instead of some random moment in the past. Being
     * placed in the past could significantly boost this task to the
     * fairness detriment of existing tasks.
     */
    if (renorm && !curr)
        se->vruntime += cfs_rq->min_vruntime;

    /*
     * When enqueuing a sched_entity, we must:
     *   - Update loads to have both entity and cfs_rq synced with now.
     *   - Add its load to cfs_rq->runnable_avg
     *   - For group_entity, update its weight to reflect the new share of
     *     its group cfs_rq
     *   - Add its new weight to cfs_rq->load.weight
     */
    update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
    update_cfs_group(se);
    enqueue_runnable_load_avg(cfs_rq, se);
    account_entity_enqueue(cfs_rq, se);

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

    check_schedstat_required();
    update_stats_enqueue(cfs_rq, se, flags);
    check_spread(cfs_rq, se);
    if (!curr)
        __enqueue_entity(cfs_rq, se);
    se->on_rq = 1;

    if (cfs_rq->nr_running == 1) {
        list_add_leaf_cfs_rq(cfs_rq);
        check_enqueue_throttle(cfs_rq);
    }
}
```

- se->vruntime += cfs_rq->min_vruntime; 将调度实体的虚拟时间添加回去，之前在fork的时候减去了min_vruntime,现在需要加回去，现在的min_vruntime比较准确；
- update_curr(cfs_rq); 来更新当前调度实体的运行时间以及CFS运行队列的min_vruntime；
- 通过注释看当一个调度实体添加到就绪队列中去时，需要更新运行队列的负载以及调度实体的负载；

- 如果设置了ENQUEUE_WAKEUP，则代表当前进程是唤醒进程，则需要进行一定的补偿；
- __enqueue_entity将调度实体添加到CFS红黑树中；
- se->on_rq = 1;设置on_rq为1，代表已经添加到运行队列中去了。

选择下一个运行的进程

当通过fork创建一个进程，然后将其添加到CFS运行队列的红黑树中，接下来就需要选择其运行了，我们直接看schedule函数。为了方便阅读主干，将代码做下精简：

```
static void __sched notrace __schedule(bool preempt)
{

    cpu = smp_processor_id();             //获取当前CPU
    rq = cpu_rq(cpu);                     //获取当前的struct rq, PER_CPU变量
    prev = rq->curr;                      //通过curr指针获取当前运行进程


       next = pick_next_task(rq, prev, &rf);   //通过pick_next回调选择进程

    if (likely(prev != next)) {

        rq = context_switch(rq, prev, next, &rf);  //如果当前进程和下一个进程不同,
则发生切换
       }
```

- 通过pick_next获取下一个运行进程
- 通过context_switch发生上下文切换

```
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    if (likely((prev->sched_class == &idle_sched_class ||
            prev->sched_class == &fair_sched_class) &&
           rq->nr_running == rq->cfs.h_nr_running)) {

        p = fair_sched_class.pick_next_task(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto again;

        /* Assumes fair_sched_class->next == idle_sched_class */
        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev, rf);

        return p;
    }

again:
    for_each_class(class) {
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
```

```
}
```

pick_next主要的两个步骤

- 因为系统中普通进程占多达数，这里通过判断当前进程的调度类以及运行队列的可运行进程个数是否等于CFS运行队列中可运行进程的个数，来做一个优化。如果相同则说明剩余的进程都是普通进程，则直接调用fair_sched_class中的pick_next回调；
- 否则跳到again处，老老实实的按照调度类的优先级从高往低依次遍历调用pick_next_task函数指针了。

```
static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags
*rf)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

again:
    if (!cfs_rq->nr_running)
        goto idle;

    put_prev_task(rq, prev);

    do {
        se = pick_next_entity(cfs_rq, NULL);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);
```

- 如果CFS运行队列中没有进程了，则会返回idle进程；
- pick_next_entry会从CFS红黑树最左边节点获取一个调度实体；
- set_next_entry则设置next调度实体到CFS运行队列的curr指针上；
- 后面再context_switch则会发生切换。

Schedule_tick(周期性调度)

接下来的部分围绕一个进程的生命周期，分析一个进程是如何被抢占？ 如果睡眠？ 如何被调度出去的？

周期性调度就是Linux内核会在每一个tick的时候会去更新当前进程的运行时间，已经判断当前进程是否需要被调度出去等。

在时钟中断的处理函数中会调用update_process_times，最终调用到调度器相关的scheduler_tick函数中。

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

    sched_clock_tick();

    rq_lock(rq, &rf);

    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    psi_task_tick(rq);

    rq_unlock(rq, &rf);
```

```
    perf_event_task_tick();

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

获取当前CPU上的运行队列rq, 在根据调度类sched_class去调用该进程调度类中的task_tick函数，此处我们只描述CFS调度类。

```
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);
    }

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

    update_misfit_status(curr, rq);
    update_overutilized_status(task_rq(curr));
}
```

通过当前的task_struct，获取调度实体se，然后根据调度实体se获取CFS运行队列，通过entity_tick函数做进一步操作

```
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);

    /*
     * Ensure that runnable average is periodically updated.
     */
    update_load_avg(cfs_rq, curr, UPDATE_TG);
    update_cfs_group(curr);


    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
}
```

- update_curr在之前有分析过，此函数主要是更新当前current进程的执行时间，vruntime以及CFS运行队列的min_vruntime
- update_load_avg 主要是用来更新调度实体的负载以及CFS运行队列的负载，在负载章节详细描述
- 如果当前CFS运行队列的个数大于1，则需要坚持下是否需要抢占当前进程的。

```
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;

    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    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.
     */
    if (delta_exec < sysctl_sched_min_granularity)
        return;

    se = __pick_first_entity(cfs_rq);
    delta = curr->vruntime - se->vruntime;

    if (delta < 0)
        return;

    if (delta > ideal_runtime)
        resched_curr(rq_of(cfs_rq));
}
```

- sched_slice用来获取当前进程在一个调度周期中理想的运行时间
- sum_exec_runtime代表的是在这一次调度中，总共执行的时间，是在update_curr中每次更新的
- prev_sum_exec_runtime是代表上次调度出去的时间，是在pick_next函数中设置的。
- delta_exec的时间则代表的是本次调度周期中实际运行的时间。
- 如果时间运行的时间大于理性的调度时间，则表示本次调度时间已经超出预期，需要调度出去，则需要设置need_resched标志
- 如果时间的运行时间小于sysctl_sched_min_granularity，则不需要调度。sysctl_sched_min_granularity此值保证在一个调度周期中最少运行的时间
- 从CFS红黑树找出最左边的调度实体se。将当前进程的vruntime和se的vruntime做比值
- 如果delta小于0，则说明当前进程vruntime比最新的vruntime还小，则不调度，继续运行
- 如果大于ideal_runtime， 如果大于理想时间，则表示运行时间已经超过太多，则需要调度。

进程睡眠

当一个进程由于要等待资源，而不得不去放弃CPU，则会选择将自己调度出去。比如串口在等待有数据发送过来，则不得不让出CPU，让别的进程来占用CPU，最大资源的使用CPU。通常需要睡眠的进程会使用schedule函数来让出CPU。

```
asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;

    sched_submit_work(tsk);
    do {
        preempt_disable();
```

```
        __schedule(false);
        sched_preempt_enable_no_resched();
    } while (need_resched());
}

static void __sched notrace __schedule(bool preempt)
{
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    if (!preempt && prev->state) {
        if (signal_pending_state(prev->state, prev)) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
            prev->on_rq = 0;

       ........
}
```

当一个进程调度schedule的函数时，传递的参数是flase。false的意思是当前不是发生抢占。之前在进程的基本概念中描述了进程的状态，进程的状态是running的时候等于0，其余是非0的。则就通过deactivate_task函数，将当前进程从rq中移除掉。

```
void deactivate_task(struct rq *rq, struct task_struct *p, int flags)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible++;

    dequeue_task(rq, p, flags);
}

static inline void dequeue_task(struct rq *rq, struct task_struct *p, int
flags)
{

    p->sched_class->dequeue_task(rq, p, flags);
}
```

最终调用到属于该进程的调度类中的dequeue_task函数中，这里还是以CFS调度类为例子：

```
static void dequeue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
    int task_sleep = flags & DEQUEUE_SLEEP;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        dequeue_entity(cfs_rq, se, flags);

                cfs_rq->h_nr_running--;
       }
}
```

获取该进程的调度实体，再获取调度实体属于的CFS运行队列，通过dequeue_entity函数将调度实体从CFS运行队列删除。

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

```
{
    /*
     * Update run-time statistics of the 'current'.
     */
    update_curr(cfs_rq);

    /*
     * When dequeuing a sched_entity, we must:
     *   - Update loads to have both entity and cfs_rq synced with now.
     *   - Subtract its load from the cfs_rq->runnable_avg.
     *   - Subtract its previous weight from cfs_rq->load.weight.
     *   - For group entity, update its weight to reflect the new share
     *     of its group cfs_rq.
     */
    update_load_avg(cfs_rq, se, UPDATE_TG);
    dequeue_runnable_load_avg(cfs_rq, se);

    update_stats_dequeue(cfs_rq, se, flags);

    clear_buddies(cfs_rq, se);

    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
```

当一个调度实体产品你个CFS润兴队列移除时，需要做以下事情

- 更新调度实体和CFS运行队列的负载
- 减去调度实体的负载从CFS_rq->runnable_avg中
- 减去调度实体的权重以及组调度的权重等
- 调用__dequeue_entity函数将需要移除的调度实体从CFS红黑树移除
- 然后更新on_rq的值等于0，代表此调度实体已不在CFS就绪队列中。

唤醒一个进程

之前在fork一个新进程之后，最后会通过wake_up_new_task来唤醒一个进程：

```
void wake_up_new_task(struct task_struct *p)
{
    p->state = TASK_RUNNING;

    activate_task(rq, p, ENQUEUE_NOCLOCK);
    p->on_rq = TASK_ON_RQ_QUEUED;

    check_preempt_curr(rq, p, WF_FORK);
}
```

通过activate_task会将此函数添加到就绪队列中，同时check_preempt_curr函数用来检查唤醒的进程是否可以强制当前进程。因为一个唤醒的进程可能是更高优先级的实时进程，当前进程是个普通进程等，都有可能发生。

```
void check_preempt_curr(struct rq *rq, struct task_struct *p, int flags)
{
    const struct sched_class *class;

    if (p->sched_class == rq->curr->sched_class) {
        rq->curr->sched_class->check_preempt_curr(rq, p, flags);
    } else {
        for_each_class(class) {
            if (class == rq->curr->sched_class)
```

```
                break;
            if (class == p->sched_class) {
                resched_curr(rq);
                break;
            }
        }
    }

}
```

- 如果唤醒的进程的调度类和当前运行进程的调度类是相同的，则调用调度类中的check_preempt_curr回调
- 如果唤醒的进程的调度类和当前正在运行的调度类不一样。如果当前是普通进程，这里唤醒的是实时进程，则直接调用reshced_curr函数，给当前进程设置need_sched的标志位，在下一个调度点调度出去。
- 如果当前进程比唤醒的进程的调度类低，则需要设置调度标志，调度当前进程
- 如果当前进程和唤醒的进程调度类相同，则通过check_preempt_curr函数去检查是否需要调度
- 如果当前进程比唤醒的进程的调度类高，则啥事不做

```
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int
wake_flags)
{
    if (wakeup_preempt_entity(se, pse) == 1)
            resched_curr(rq);

}
```

通过wakeup_preempt_entity来判断是否可以强制当前进程，如果可以则设置need_sched标志位。

```
/*
 * Should 'se' preempt 'curr'.
 *
 *             |s1
 *        |s2
 *   |s3
 *         g
 *      |<--->|c
 *
 *  w(c, s1) = -1
 *  w(c, s2) =  0
 *  w(c, s3) =  1
 *
 */

static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
    s64 gran, vdiff = curr->vruntime - se->vruntime;

    if (vdiff <= 0)
        return -1;

    gran = wakeup_gran(se);
    if (vdiff > gran)
        return 1;

    return 0;
}
```

- 第一个参数是当前进程的调度实体，第二个参数是唤醒的进程的调度实体
- vdiff的是值是当前调度实体的虚拟时间和唤醒进程调度实体的虚拟时间之差

- 如果vdiff小于0，则表示当前进程的虚拟时间小于唤醒的vruntime，则不抢占
- wakeup_gran是用来计算唤醒的调度实体在sysctl_sched_wakeup_granularity时间内的vruntime。
- 大概意思就是当前进程的调度实体小于唤醒进程的调度实体的值大于gran，则才可以选择调度
- 如果当前进程的和唤醒进程的vrumtime的差值没达到gran的，则不选择，通过注释可以清晰的看到

进程抢占调度总结

- 当一个进程通过fork创建之时，在sched_fork函数中会对此进程设置对应的调度类，设置优先级，更新vruntime的值
- 此时需要将进程添加到就绪队列中，对于CFS就绪队列，则需要添加到CFS红黑树中，跟踪进程的vruntime为键值添加。因为就绪队列添加了一个新的进程，则整个就绪队列的负载，权重都会发生变化，则需要重新计算。
- 当添加到就绪队列之后，则就需要通过pick_next回调来选择一个新的进程，选择的策略是选择CFS红黑树vruntime的进程来运行
- 当此进程运行一段时间后，则就会通过schedule_tick函数来判断当前进程是否运行时间超过了理想的时间，如果超过则调度出去
- 或者当此进程需要等待系统资源，则也会通过schedule函数去让出cpu，则就会从CFS就绪队列中移除此进程，移除一个进程同样整个CFS运行队列的权重和负载就会发生变化，则需要重新计算
- 当资源就绪之后，则需要将当前进程唤醒，唤醒的时候还需要检查当前进程是否会被高优先级的进程抢占，如果存在高优先级的调度类则发生抢占，如果是同等调度类的则需要判断vruntime的值是否大于一个范围，如果是则设置调度标志。

六、CFS对比其他调度算法

Linux内核的CFS进程调度算法，无论是从2.6.23将其初引入时的论文，还是各类源码分析，文章，以及Linux内核专门的图书，都给人这样一种感觉，即 ***CFS*调度器是革命性的，它将彻底改变进程调度算法。** 预期中，人们期待它会带来令人惊艳的效果。人们希望CFS速胜，但是却只是 **在某些方面比*O(1)*调度器稍微好一点点。** 甚至在某些方面比不上古老的BSD调度器。

4.3BSD采用了1秒抢占制，每间隔1秒，会对整个系统进程进行优先级排序，然后找到优先级最高的投入运行，非常简单的一个思想，现在看看它是如何计算优先级的。

首先，每一个进程j均拥有一个CPU滴答的度量值C_j，每一个时钟滴答，当前在运行的进程的CPU度量值C会递增：

$$C_j = C_j + 1$$

当一个1秒的时间区间i过去之后，C_j被重置，该进程j的优先级采用下面的公式计算：

$$C_j(i) = \frac{C_j(i-1)}{2}$$

$$Prio_j = Base\_Nice_j + \frac{C_j(i)}{2}$$ 【此处为了简单，忽略$nice$值的影响】

可以计算，在一个足够长的时间段内，两个进程运行的总时间比例，将和它们的$Base_Prio$优先级的比例相等。

4.3BSD的优先级公平调度是CPU滴答驱动的。

4.3BSD调度器

Linux内核在2.6.23就采用了CFS调度器。所以一个原因就是没有比较。Android系统上，CFS没有机会和O(1)做比较。另外，即便回移一个O(1)调度器到Android系统去和CFS做比较，CFS同样不惊艳，原因很简单。Android系统几乎都是交互进程，前台进程却永远只有一个，你几乎感受不到进程的切换卡顿，换句话说，即便CFS对待交互式进程比O(1)好太多，你也感受不到，因为对于手机，平板而言，你切换APP的时间远远大于进程切换的时间粒度。

那么，CFS到底好在哪里？

简单点说，CFS的意义在于，**在一个混杂着大量计算型进程和*IO*交互进程的系统中，*CFS*调度器对待*IO*交互进程要比*O(1)*调度器更加友善和公平**，理解这一点至关重要。

其实，CFS调度器的理念非常古老，就说在业界，CFS的思想早就被应用在了磁盘IO调度，数据包调度等领域，甚至最最古老的SRV3以及4.3BSD UNIX系统的进程调度中早就有了CFS的身影，可以说，Linux只是**使用*CFS*调度器**，而不是设计了CFS调度器。

Linux CFS只是为了解决O(1)中一个**“静态优先级/时间片映射”**问题，那么可想而知，它并不能带来什么惊艳效果。但是”，这个O(1)调度器的问题其实在计算密集型的守护进程看来是好事，毕竟高优先级进程可以**无条件持续运行很久而不切换**。这对于吞吐率的提高，cache利用都是有好处的。

当然，使用调优CFS的时候，难免也要遇到IO睡眠奖惩等剩余的事情去设计一些trick算法，这需要花费精力。

七、Reference

深度解读：为什么Win/iOS很流畅而Linux/Android却很卡顿？怎样才能不卡顿？-面包板社区 (eet-china.com)
CFS调度主要代码分析一 - 云+社区 - 腾讯云 (tencent.com)
为什么Linux CFS调度器没有带来惊艳的碾压效果 - 云+社区 - 腾讯云 (tencent.com)
linux内核分析——CFS（完全公平调度算法） - Summer (luochunhai.github.io)
The Linux Kernel documentation — The Linux Kernel documentation
linux/kernel/sched at v5.12 · torvalds/linux (github.com)
kernel: The openEuler kernel is the core of the openEuler OS, serving as the foundation of system performance and stability and a bridge between processors, devices, and services. - Gitee.com