CFS 调度器

--wxc200

大家好哈，

兄弟最近在学习调度器，有点儿心得，更多得是迷惑，写出心得来与大家分享，

贴出迷惑来请大家解答。呵呵

linux 自 2.6.23 后引入了一种新的调度器，叫'Completely Fair Scheduler'（wiki）.是由 Ingo Molnar 在很短的时间内写的。他与的 cfs 代码是对之前另一种调度器 "O(1) scheduler" 的替换.

先扯一下 o(1)调度.

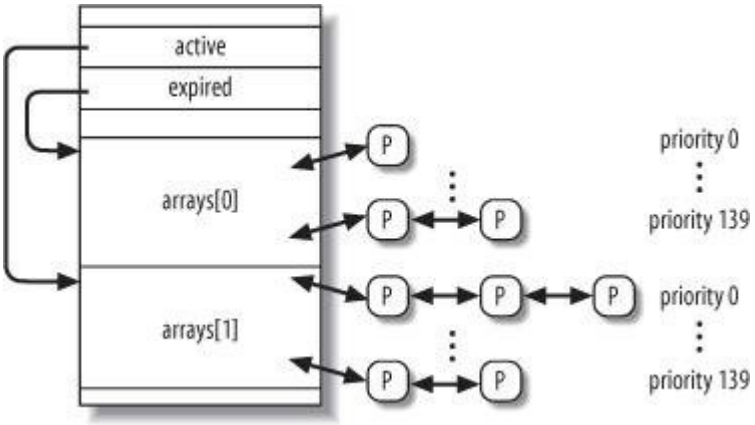
先看 wiki 的解释:

"An O(1) scheduler is a kernel scheduling design that can schedule processes within a constant amount of time, regardless of how many processes are running on the operating system (OS)."

Understanding Linux Kernel(3th) 这本书 chapter 7 讲的进程调度,7.2 节提到"Scheduling Algorithm",说 2.6 对进程的选择是 constant time,兼顾了批处理和交互式进程.进程分类,实时进程(又分为 SCHED_FIFL AND SCHED_RR)和普通进程.

最主要的问题是这些进程怎么组织的,简单看下结构:

没办法传图片,请参照附件"数组".



它有两个数组,active 里面是 has time-slice ,expired 数组是 out-of-slice。简单的说，一个普通的进程被调度运行后，放在 active 里，当其时间片用光后，可能就要移到 expired 数组里了。说“可能”是因为有的进程就不移走。比如交互式进程。

说白了,就是把所有的 process 按照优先级挂在链表上,从里面按照优先级高低选择第一个不为空的进程运行.

普通进程的优先级是 100~139,实时进程要更低,这符合调度的算法.

我有点等不及了,咱们直接奔 cfs 吧~

另外有点就是,引入 hrtimer 之后,进程调度还是在 tick 中断完成的.每个 tick 都会检查进程是否应该调度,当然,主动让 cpu(即调用 scheduler().)的就不算了吧...

hrtimer 那部分东东等咱们聊完了 cfs 再说，那个主要是在原有的时间管理 layer 上新添加了“时间事件”，把时间中断以事件的方式注册。精确度由之前的 hz 提升到了 ns（需要硬件支持）。。。

cfs

Documentation/scheduler/sched-design-CFS.txt 介绍了 cfs 相关东西，也可以在线看.

我按照我的理解来“添油加醋”地翻译下。

1 概括

“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.”

""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.
"

模 拟了个理想的，多任务 cpu.假如有 n 个进程，那么每个进程 的执行速度是 1/n,,即所有的任务都是并行执行的。我认为就算是有并行执行的 cpu，速度也不应该完全平均，按照优先级再分比较划算。比如 2 个进 程，a,b，a 的优先级是 b 的两倍，那么就照着速度比 a:b = 2:1 呗~

“On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime." The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks. ”

当然没有这种 cpu,故引进了个新概念"virtual runtime",这个概念真是折磨人好久。我曾经在 clf 上发帖子问过，有个兄弟回复还是不错的。后面看过代码后，我的理解是：1） 红黑树结点排列的值 2） 每次 task run time 转换的一个值 .

先看上文颜色部分，我真是很难翻译它~ 照字面理解，是实际运行时间与任务数

量的一个比值？ 我举个例子来说明它是怎么计算 的吧。。

在 __update_curr() 这个函数里，会更新当前任务的运行时间信息。假如一个任务 a 自上次调度到这次调度时间间隔为 delta,那么它的 vrumtime 的增量是按照这个公式：delta * NICE_0_LOAD / a->se.load。假如把 NICE_0_LOAD / a->se.load 作为一个比值 p 的话，我们可以这么描述 p 的变化：优先级越高，p 越小。这个 load 是与 nice 值 相关的，有一个静态的数组，我后面在代码里会介绍。
所以，一堆进行都运行了一段时间 delta，那么它们的 vrumtime 就遵循上面的公式。很明显，优先级最大的进程 vrumtime 增量最小。。。

2 操作细节

cfs 就是通过追踪这个 vruntime 来进行任务调度的。它总是选 vruntime 最小的进程来运行。

3 红黑树。

红黑树这个数据结构在内核里用得还不少嘛，不过俺不太懂。哪位兄弟给扫扫盲？ hrtimer 里也用到了 red-black-tree。这里把进程的 vruntime 用 rb-tree 来存储。

4 一些 feature

cfs has no time-slice concept.o(1)有的，cfs 没有“明显”得用，它偷偷摸摸地用。呵呵 翻译完这几段咱再说这个。文档里面那个接口估计是用来调整“最小粒度”的。用于“桌面”和“服务器“两 种不同的调度。后者可能不太希望频繁的调度，浪费时间。前者则希望面面俱到--”不患寡妇而患不均也“

5 调度 policy

SCHED_FIFO/_RR are implemented in sched_rt.c and are as specified by POSIX. 实时进程调度

实时调度和普通进程调度。

6 调度的类。

调度哭可以模块化注册，所以你可以选择实时调度或者是 cfs 调度。不错！

sched_fair.c 文件实现了 cfs 调度的所以 classes

7 组调度。

不光是普通进程，组调度也实现了。。这个是后来一个 patch 加的。

××××××××××××××××××××××××××××××××××××

上面 是对着内核的一篇文档，简要地说了几部分 。。。在正式进行我们的 hack 之前，先唠叨几句，算是个总结和感性的认识吧~吼吼

关于实时进程的调度，这一次先不分析，它和 o1 差不多，还保持着优先级数组的

做法，那是”上流社会“玩儿的游戏，后面再折腾它。”普通大众“们玩儿的就是 cfs 了。

cfs 调度，我写两 部分，一部分是普通进程的调度，没有组的概念。一部分是组的调度。我个人觉得组得调度比较难理解~ 这几天一直在思考。。。今天下午好像豁然开朗了。。。画了个草图，到后面我做出这张图来，大家看看对不对 :D

咱们先说普通进程的调度

关于普通进程的组织，应该有这么一种印象。

有一个队列，叫 cfs_rq,里面维护了个红黑树。每一个 task_struct has a se,称为调度实体。它就代表了一个被调度的进程，里面有此进程的一些信息，比如前面提到的 vruntime.

一个进程创建的时候，就会被放置在这个红黑树里。它自己的位置是由它的 vruntime 值 来决定的。在每个 tick 中断的时候，会更新进程的信息，并检查这个红黑树，判断某些进程是不是要被替换掉。
现在我们来想象下面一个例程。
进 程 a 被创建了，sched_fork()会做一些新创建进程调度方面的初始化。然后应该尝试把此进程放到队列里啊，让它被调度。 set_task_cpu()做了这部分工作。然后这个进程如果不是实时进程，就让它跟 cfs 的类绑定在一块儿，这样下面用到调度的一些方法，就会到 cfs 相关的类去找喽~ 这时候如果抢占打开了，会去调度一次，以让新创建的进程有机会被调度一次。或者在下一个 tick 的时候，会检查已经红黑树，以确认这个进程是不是调度。 （注：上面红色这句有点胡扯的嫌疑，请明白人指点）

比如在每个 tick 中断的时候，会去红黑树里面找 vruntime 最小的那个（红黑树最左边的叶子）去调度。那么这个调度过程，所有用到的方法，就是上面提到的 cfs 的类给实现的。

最后，大家再对 rb-tree 里面的任务结点有个感性的认识吧：

优先级高的进程，总是靠左，以有最大调度机会。比如说有 n 个进程，那么在一段时间内，应该把 n 个进程都运行一遍。这儿有两三个问题，一段时间是多久？每个进程有多少时间去运行呢？每个进程分到的运行时间跟 vruntime 有什么关系呢？

一段时间，与运行着的进程数目有关。进程越多，时间越长。每个进程并不是平均分配这些时间的。按照我的理解，分到这个时间越多的进程，后面 vruntime 增长得越慢。

上面 这几句话，我可是晕了近一个星期才明白的。不知道跟大家的理解一致么？

本 来我错误得理解是这样的，完全公平调度么，当然所有的进程有同样的运行时间。如果是这样，那么 rb-tree 就没用了。因为每个结点都一样的值。所以，请 大家有这样一种概念：两 个进程，a 优先级特别低，b 特高。假如现在有 n 个正被调

度的进程，它们都被调度一遍的时间是 delta，那么 b 会占很高的时间，a 很低的时间。运行完成后，b 还是在很靠左的地方呆着，而 a 一定往最靠右的地方。。。。哎，表述不清啊，，最好画个图。。。这样就为了让 b 得到更常的运行时 间。。。而其 vruntime 仍然很小（可以参照上封邮件里面那个计算 vruntime 的公式）

好了

我们开始真正的代码旅行吧。

1 几个重要的数据结构

1）task_struct : 大家都知道这个。

```
struct task_struct {
 ...

 int prio, static_prio, normal_prio; 进程的优先级
 unsigned int rt_priority;实时进程的优先级
 const struct sched_class *sched_class; 调度类，一堆函数指针，实现调度
 struct sched_entity se;调度实体 一个进程对应一个调度实体，，
 struct sched_rt_entity rt;
....
}
```

2) sched_class 调度相关的函数指针。

```
struct sched_class {
...
 void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup); 入列
 void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);出列
 void (*yield_task) (struct rq *rq);

 void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int sync);检查当前进程可否被新进程抢占

 struct task_struct * (*pick_next_task) (struct rq *rq); 选择下一个进程运行
 void (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
 int (*select_task_rq)(struct task_struct *p, int sync);
```

....
#ifdef CONFIG_FAIR_GROUP_SCHED 请格外留意此宏。。。。跟组调度相关的，，暂时不管它，后面跟组相关的调度再回来看它。
 void (*moved_group) (struct task_struct *p);
#endif
};

3） 调度实体
struct sched_entity {
 struct load_weight load; /* for load-balancing */ nice 对应的 load 值
 struct rb_node run_node; 红黑树结点
 struct list_head group_node;
 unsigned int on_rq; 这个是什么呢？不知道

 u64 exec_start; 上次开始调度时的运行时间
 u64 sum_exec_runtime; 总运行时间
 u64 vruntime;呵呵 都知道了
 u64 prev_sum_exec_runtime; 上次调度总运行时间
...
#ifdef CONFIG_FAIR_GROUP_SCHED 如果是组调度的话，就多了些部分。
 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

}

4)cfs 运行队列
/* CFS-related fields in a runqueue */
struct cfs_rq {
 struct load_weight load;
 unsigned long nr_running;

 u64 exec_clock;
 u64 min_vruntime;

5)大 boss，运行队列
struct rq {

```
,...
 unsigned long nr_running;
 #define CPU_LOAD_IDX_MAX 5
 unsigned long cpu_load[CPU_LOAD_IDX_MAX];
...

 struct cfs_rq cfs;
..
 struct task_struct *curr, *idle;
}
```

6）调度相关类

```
/*
 * All the scheduling class methods:
 */
static const struct sched_class fair_sched_class = {
    .next           = &idle_sched_class,
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task       = yield_task_fair,

    .check_preempt_curr    = check_preempt_wakeup,

    .pick_next_task       = pick_next_task_fair,
    .put_prev_task       = put_prev_task_fair,

#ifdef CONFIG_SMP
    .select_task_rq       = select_task_rq_fair,

    .load_balance       = load_balance_fair,
    .move_one_task       = move_one_task_fair,
#endif

    .set_curr_task          = set_curr_task_fair,
    .task_tick       = task_tick_fair,
    .task_new       = task_new_fair,

    .prio_changed       = prio_changed_fair,
    .switched_to       = switched_to_fair,
```

```
#ifdef CONFIG_FAIR_GROUP_SCHED
    .moved_group        = moved_group_fair,
#endif
};
```

2 吃代码

上面几个结构体的关系还是很好理解的，它们的关系我本来想画个图表示下的，觉得有点麻烦，何况也不难，我就不画了。。直接进代码了哈~

在进行之前呢，先介绍两 篇文章，是一个网友写的。之前和他聊 cfs，后来我迷糊了，他没迷糊，就写了这篇文章。。。还不错。

Linux 进程管理之 CFS 调度器分析

Linux 进程管理之 CFS 组调度分析

大家的理解差不多，他写得很条理，为了防止雷同，雷着大家了，我不会引用它里面的文字的。我就在边读代码的过程中边把自己的体验和疑惑贴出来，大家一同讨论吧~ 错误，一定要指出我的错误啊~呵呵

咱们先从新创建的一个进程说起吧。

1) kernel/fork.c 里,fork routine,do_fork() will create a new process,if its state is running,it will

call wake_up_new_task() to wake up the newly created process FOR THE FIRST TIME.

```
do_fork(){
...
    if (unlikely(clone_flags & CLONE_STOPPED)) {
            ...
        } else {
            wake_up_new_task(p, clone_flags);
        }
...
}
```

2) 这个函数做什么呢？

```
void wake_up_new_task(struct task_struct *p, unsigned long clone_flags)
{
    unsigned long flags;
    struct rq *rq;
```

```
    rq = task_rq_lock(p, &flags);顺序操作运行队列
    BUG_ON(p->state != TASK_RUNNING);
    update_rq_clock(rq);

    p->prio = effective_prio(p);   计算 priority,,普通进程的 priority 就是 static priority

    if (!p->sched_class->task_new || !current->se.on_rq) {如果条件不满足，直接入
列，但请注意 active_task()的最后参数 0，不唤醒
        activate_task(rq, p, 0);
    } else {
        /*
         * Let the scheduling class do new task startup
         * management (if any):
         */
        p->sched_class->task_new(rq, p);调用完全公平类里面的 task_new 做些初始化
的操作。
        inc_nr_running(rq);增加运行队列的运行进程数目
    }
    trace_sched_wakeup_new(rq, p);
    check_preempt_curr(rq, p, 0); 可不可以抢占当前进程？
#ifdef CONFIG_SMP
    if (p->sched_class->task_wake_up)
        p->sched_class->task_wake_up(rq, p);
#endif
    task_rq_unlock(rq, &flags);
}
```

3) 先看 task_new()吧，它会掉到 fair_sched_class 类里的 task_new_fair.

```
static void task_new_fair(struct rq *rq, struct task_struct *p)
{
    struct cfs_rq *cfs_rq = task_cfs_rq(p);
    struct sched_entity *se = &p->se, *curr = cfs_rq->curr;
    int this_cpu = smp_processor_id();

    sched_info_queued(p);

    update_curr(cfs_rq); 更新 cfs 的一些信息
    place_entity(cfs_rq, se, 1);初始化 se 在 cfs 里的信息，包括 vruntime
```

```
    /* 'curr' will be NULL if the child belongs to a different group */
    if (sysctl_sched_child_runs_first && this_cpu == task_cpu(p) &&
            curr && curr->vruntime < se->vruntime) {
        /*
         * 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_task(rq->curr);
    }

    enqueue_task_fair(rq, p, 0);  放进队列里面
}
```

4) 我们先看 update_curr()吧

```
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);计算自上次调度此进程到这次又调度，经过的时间
```

这个时间比较鬼异，是多久呢？假如在运行队列里的 n 个进程之一的 a,再一次被调度到时，这个 delta_exec 是多大呢？ 如果中途有进程退出或睡眠，那么运行队列会动态更新的，所以这个 delta_exec 的变化曲线是什么样子的难说。

```
    __update_curr(cfs_rq, curr, delta_exec);  把刚计算出来的运行时间作为参数传进去，它做什么事情呢？
    curr->exec_start = now;呵呵，更新完了立刻打个时间戳。
```

if (entity_is_task(curr)) { 这个条件比较有趣，我喜欢。跟过去发现是这样定义的：
"/* An entity is a task if it doesn't "own" a runqueue */
#define entity_is_task(se) (!se->my_q) " 如果是组调度，调度实体可能代表一个组，不单单是一个 task。就是看看有没有 my_q 这个指针了，
即有没有它 control 的 cfs_rq
如果是 task,条件满足，

```
        struct task_struct *curtask = task_of(curr);

        cpuacct_charge(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec); 这两 个不管了，看不懂
    }
}
```

6）静悄悄地过来，看看这个经常调用的函数，到底做了啥捏？

```
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
          unsigned long delta_exec) 传进来的执行时间
{
    unsigned long delta_exec_weighted;

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

    curr->sum_exec_runtime += delta_exec; 直接加到总运行时间变量里去，
    schedstat_add(cfs_rq, exec_clock, delta_exec);
    delta_exec_weighted = calc_delta_fair(delta_exec, curr); 这个很重要。。。这个函数名字叫”公平计算 delta"，咋公平捏？第七步分析会看到
    curr->vruntime += delta_exec_weighted; 把上步计算出来的值加到了 vruntime 里
    update_min_vruntime(cfs_rq);
}
```

7) 看看怎么计算这个 delta_exec_weighted 的？

```
/*
 * delta /= w
 */
static inline unsigned long
calc_delta_fair(unsigned long delta, struct sched_entity *se)
```

```
{
   if (unlikely(se->load.weight != NICE_0_LOAD))  如果进程实体有默认的 load 值，直接返回 delta
      delta = calc_delta_mine(delta, NICE_0_LOAD, &se->load);计算应该修正的这个值

   return delta;
}
```

在往下走之前，先弄明白 nice 和 se->load 之间的关系吧。

nice 都知道，se->load 是指调度实体的负载。同理一个 cfs_rq 也有负载。。一般是所有 task 的 load 之和。这个 load 是通过 nice 与 一个静态数组转换来的。一般普通进程的 nice 值在-20~19 之间，其对应 load.weight 值是递减的，具体可参见那个静态数组。而 NICE_0_LOAD 就是 nice=0 的对应的 load 值。

好了，这相当于是做一个修正喽。假如现在传进来的运行时间是 10,那么，这个调度实体应该返回的 delta 是多少呢？看这个函数的 comment 就明白了，delta * NICE_0_LOAD/se->load，是一个比重。。。之前这个地方说过了，呵呵。

那么，调度实体的优先级越高，其得出来的值越小。

反回到 6)中，可以知道 vruntime 是怎么 update 的了。那么，它是如何初始化的呢？后面有，再说,是通过 vslice()计算的。

8）我们再返回到 task_new_fair,看 update_curr()后，接下来做的事情　place_entity.

这个函数也比较好玩，做得是一个”奖励“工作：

```
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);  稍微加一些，呵呵
```

```
    if (!initial) {    如果是睡了，唤醒的，应该有些补偿的   具体怎么补，多了怎么办，少了怎么办？
        /* sleeps upto a single latency don't count. */
        if (sched_feat(NEW_FAIR_SLEEPERS)) {
            unsigned long thresh = sysctl_sched_latency;

            /*
             * convert the sleeper threshold into virtual time
             */
            if (sched_feat(NORMALIZED_SLEEPER))
                thresh = calc_delta_fair(thresh, se);

            vruntime -= thresh;
        }

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

    se->vruntime = vruntime;
}
```

9）最后，task_new_fair 做的事情就是让 task 入列。enqueue_task_fair()

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

    for_each_sched_entity(se) {遍历所有的 se 及它的父亲。。。这个在非组调度时，就是 se,组调度时，会将其 dad 一同入列
        if (se->on_rq)如果已经在运行队列，就停止
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, wakeup);真正的入队函数，，，把 se 插入到 rb-tree.
        wakeup = 1;
    }

    hrtick_update(rq);   hr 部分的，后面再分析。
```

}

10)enqueue_entity() 会更新时间，最终调———enqueue_entity()，把 schedule_entity 往红黑树中存放。每个结点的值是通过 entity_key()来实现的，比较奇怪，是 se->vruntime-cfs->min_vruntime....不知道这么做。

入列操作有一些跟唤醒判断有关的，后面再查资料看看
11）再回到 wake_up_new_task()来。
会调用 check_preempt_curr()，判断当前进程是否被新进程抢占。这个函数调用 sched_fair_class 里的 check_preempt_wakeup,这个函数也很好玩儿，它会调其中一个函数，wakeup_preempt_entity()，来判断当前的进 程和新进程之前满足什么样的关系才可以抢占。

这时新加了个区间：新进程的 vruntime 比 current 小，但要满足一定条件，才能完成抢占。
三种情况返回值为-1/0/1,代表不同的意思。

最后我们又返回到了 wake_up_new_task.一个新进程创建后被调度的过程大体就是上面的流程。

另一个比较有意思的就是 tick 了，明天看看 tick 中断做的事情吧。

一些有意思的关于 cfs 调度的资料：
http://video.linuxfoundation.org/video/1029 视频 cfs 作者的演讲
Some kernel hackers...
Thomas Gleixner
http://www.google.com/search?hl=en&q=Thomas+Gleixner&btnG=Google+Search
http://kerneltrap.org/Thomas_Gleixner
http://de.wikipedia.org/wiki/Thomas_Gleixner

Schedulers: the plot thickens
http://lwn.net/Articles/230574/

[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]
http://lwn.net/Articles/230501/

http://www.ibm.com/developerworks/cn/linux/l-cfs/index.html 完全公平调度程序

鼠眼看 Linux 调度器
http://blog.chinaunix.net/u1/42957/showart.php?id=337604

http://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/index.html
Linux 调度器发展简述

至今不敢写一篇关于 cfs 的文章收藏
http://blog.csdn.net/dog250/archive/2009/01/15/3793000.aspx

http://www.ibm.com/developerworks/cn/linux/l-cfs/index.html
使用完全公平调度程序（CFS）进行多任务处理

3 关于 tick 中断
为了保证调度，在每个 tick 都会对进程时间信息等进行更新。首先找一个从 hrtimer 上层到进程调度之间的点，暂时往进程调度的方向说，后面谈到 hrtimer 时，再往 hrtimer 追根溯源吧。
这个点就是，update_process_times,它是在 timer interrupt 中被调 的，它会调一个跟 process 直接相关的函数,scheduler_tick().

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 *
 * It also gets called by the fork code, when changing the parent's
 * timeslices.
 */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    ...
    update_cpu_load(rq);
    curr->sched_class->task_tick(rq, curr, 0);   task_tick 即公平调度类里的
    spin_unlock(&rq->lock);

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

看看 task_tick：

```
/*
 * 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) {   老朋友了，组调度的时候会遍历父亲们，否则就是它自己
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);  传进来的 queued=0
    }
}
```

继续看 entity_tick;

```
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);   更新当前进程 current 的信息，这个昨天已经看过了

#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);
```

这个要与 check_preempt_curr 对比着看，我当初就在这个地方晕了。。。注释是这样的：“ Preempt the current task with a newly woken task if needed:”

```
}
```

把两 个函数都贴出来：

```
1)  static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int sync)
{
    struct task_struct *curr = rq->curr;
    struct sched_entity *se = &curr->se, *pse = &p->se;
。。。
    if (wakeup_preempt_entity(se, pse) == 1) {
```

这个是根据当前进程 curr 和要调度的 p 之前的 vruntime 比较，有一个区间限度。

```
。。。
}
2) static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;

    ideal_runtime = sched_slice(cfs_rq, curr);
```

算一下当前进程理想的运行时间是多少？

```
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
```

算算它已经运行了多久？

```
    if (delta_exec > ideal_runtime)
        resched_task(rq_of(cfs_rq)->curr);
}
```

有趣的是，上面两 个函数的 comment 是一样的 ：-)

第二个函数折磨了我好久~ 既然是完全公平调度，每个进程的运行时间应该完全公平呀？ 实际上 ideal_runtime 是与进程的优先级有关系的。
首先看 ideal_runtime 是怎么计算出来的。

3)

```
/*
 * We calculate the wall-time slice from the period by taking a part
 * proportional to the weight.
 *
 * s = p*P[w/rw]这是通过一个比例算出的
 */
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    unsigned long nr_running = cfs_rq->nr_running;运行的进程数目

    if (unlikely(!se->on_rq))
        nr_running++;

    return calc_delta_weight(__sched_period(nr_running), se);   这个 weight 是跟进程数目有关系的
}
```

4)看看 cale_delata_weight 第一个参数是怎么得到的。

```
/*
 * The idea is to set a period in which each task runs once.   设置一个 period,让每个进程都跑一遍。
 *
 * When there are too many tasks (sysctl_sched_nr_latency) we have to stretch
 * this period because otherwise the slices get too small.  当进程数目大于默认值（5）时，要拉伸一下这个 period
 *
 * 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;  最小粒度    4ms
          period *= nr_running;  4ms * 进程数目
    }
```

```
    return period;
}
```

这个比较好理解，小于 5 个进程时，这个 period 周期是 20ms,大于 5 个 running number 时，就让 4*nr;后面就是线性的，

回到 3,计算出来的这个 period 作为 calc_delata_weight（）的第一个参数往下传，继而算出当前 se 的比重来。

5）

```
/*
 * delta *= P[w / rw]
 */
static inline unsigned long
calc_delta_weight(unsigned long delta, struct sched_entity *se)
{
    for_each_sched_entity(se) {
        delta = calc_delta_mine(delta,
                se->load.weight, &cfs_rq_of(se)->load);
```

这个函数不陌生，在计算 vruntime 的时候也有它，不过那个函数是 calc_delta_fair,重点是中间的参数，一个是 nice_0_load,一个是当前进程的 load

```
    }

    return delta;
}
```

这个很简单： 假如 n 个进程的 period 是 4n,其中进程 i 的 load.weight 是 m,所有进程的 load 是 M，那么进程 i 应该在这个 period 里面分到的蛋糕是： 4n * m/M

再返回 check_preempt_tick，通过比较实际计算出来的值与理想值 比较，确定当前进程是否被抢占。。

这个地方有 个疑问： check_preempt_tick 是在 tick 中断里调的，check_preempt_curr()昨晚分析过在进程创建的时候有过调用，后都传进 去一个需要调度的进程作为参数，与当前进程比较，满足一定的条件时，才会把当前进程切换掉。前者是在 tick 中断里掉，不知道下一个调的进程是谁，这个应 该由调度器后面去选一个。没关系，它的工作就是看看当前进程把自己的时间片用完了没？用完了的话，我就回去告诉调度器，把你调度走，，，具体怎么调度呢？下面我们接着看吧。。。先猜想一下：1） 把当前进程出列，调整其在红黑树中的位置。2） 从红黑树中选择下最左结点运行

4 调度函数

resched_task（）里设置某个进程需要调度。其实就是设定一个 flag 而已。

```
static inline void set_tsk_need_resched(struct task_struct *tsk)
{
   set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
}
```

来看看大 boss,

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
...

   prev->sched_class->put_prev_task(rq, prev);
   next = pick_next_task(rq, prev);

....
}
```

中间这两 个函数，是我 们关心的。

1)

```
/*
 * Account for a descheduled task:
 */
static void put_prev_task_fair(struct rq *rq, struct task_struct *prev)
{
   struct sched_entity *se = &prev->se;
   struct cfs_rq *cfs_rq;

   for_each_sched_entity(se) {
      cfs_rq = cfs_rq_of(se);
      put_prev_entity(cfs_rq, se);
   }
}
```

这是对一个即将被调度出去的进程的处理。

2)　它会掉 pub_prev_entity 这个例程看得有点儿迷糊。 如果进程还在 runqueue

上，就把它重新放回红黑树，只是位置变了,并把当前的 curr 指针清空。

```
static void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
{
    /*
     * If still on the runqueue then deactivate_task()
     * was not called and update_curr() has to be done:
     */
    if (prev->on_rq)  这个条件还得研究研究，什么情况下用与不用
        update_curr(cfs_rq);

    check_spread(cfs_rq, prev);
    if (prev->on_rq) {
        update_stats_wait_start(cfs_rq, prev);
        /* Put 'current' back into the tree. */
        __enqueue_entity(cfs_rq, prev);   怎么又放回去呢？
    }
    cfs_rq->curr = NULL;
}
```

3) 回到 scheduler 函数，

```
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
    struct task_struct *p;
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;

    if (unlikely(!cfs_rq->nr_running))
        return NULL;

    do {
        se = pick_next_entity(cfs_rq); 挑出下一个可以运行的
        set_next_entity(cfs_rq, se);  设置它为 cfs_rq 的 curr
        cfs_rq = group_cfs_rq(se);是 group 的 se 么？
    } while (cfs_rq);   这个循环其实很有意思，它为 group 调度提供支持。如果 cfs_rq 这层里面的进程被选择完毕，它会接着选择其父 task_group 里的去运行。

    p = task_of(se);
    hrtick_start_fair(rq, p);

    return p;
```

}

4) pick_next_entity 会调用__pick_next_entity(cfs_rq)去选择红黑树最左侧的结点，然后有两 个条件判断，作为返回 se 的最终人选。
"
```
    if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, se) < 1)
        return cfs_rq->next;

    if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, se) < 1)
        return cfs_rq->last;
```
"
wakeup_preempt_entity（）这个我在前面分析过了，这是最终决定两 个进程能否进行切换的一个判断。。。next 和 last 估计是 cfs 里定时更新的亲戚，好事当然先轮着自己喽。。所以这个地方调度会优先选择下，，，具体再讨论吧。

5) 选出来了以后，会通过 set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)设置一些信息，把它贼给 cfs_rq->curr,更新总运行时间等。
6）这儿还有一个奇怪的函数：
hrtick_start_fair(rq, p);
我认为选出了可以运行的进程，下一步应该真正让它运行吧，，这步操作在哪呢？
上面这个函数只是做了些判断，是否当前进程真得需要运行？然后更新了些与 hrtick 相关的 rq 信息。
也有可能是 scheduler() 函数下面做的吧，具体我还没有打到。

这儿有个小猜想，希望得到验证：
1) a task call schedule(),it's se->on_rq = 1,so it need denqueue .call deactivate_task()

2) a task picked by schedule() need to get the cpu to run,it's se->on_rq = 0,so it need enqueue.
3) a task's exec time is over,need change to rb-tree's right it's se->on_rq = 1,just change the tree

这个再想想。。。。

组调度

先说一下提纲

1 关于 group schedule 的文档
2 关于组调度的认识
3 一些与组调度相关的数组结构
4 组调度结构图
5 具体的操作函数
6 后记

1　按照文档的解释，完全公平调度并不仅仅针对单一进程，也应该对组调度。例如有两 个用户 wxc1,wxc2，以用户 userid 来分，调度器分配给两 个组的时间是公平的,wxc1 and wxc2 各有 50%的 CPU，，组内也实现公平调度。

几个宏：
CONFIG_GROUP_SCHED ：打开组调度

CONFIG_RT_GROUP_SCHED 实时进程组调度

CONFIG_FAIR_GROUP_SCHED 普通进程。
组的划分：
1）Based on user id (CONFIG_USER_SCHED)
2）Based on "cgroup" pseudo filesystem (CONFIG_CGROUP_SCHED)
后面这一种还不太熟悉，参照 eric xiao 网友的 blog,在对 cgroup 文件系统进程操作时，会去调用一个接口，里面的函数完成对组调度的操作。
cgroup 这部分后面再学习，暂时还不了解。

2 关于组调度的认识
1）组调度是分层次的。
2）一个组里面有数个进程，这些进程的调度作为一层。
3）所有的组都是连在一起的，他们之前有父子兄弟关系。为首的是 root-task-group
4）另一层是组调度，即把一个组作为一个调度实 体，仍然用 schedule_entity 这个结构体来表示。
5）一个 task_group 在每个 cpu 上都关联着一个 cfs_rq,一个 schedule_entity，注意，这两 个东东是与之前单个进程调度里提到的 cfs_rq & schedule_entity 有分别的。。它们代表得是组。跟之前单个进程不一样。
6）每个 cpu 的 run queue 里有个链表，把所有在这个 cpu 上属于某些组的 cfs_rq 链在了一起。
7） 组与组之间的父子关系，同时代表着组在某个 cpu 上的调度实体之间的父子关系。假如说 wxc1 是 wxc2 组的 parent,那个 wxc1 在 cpu0 上的调度实体 schedu_entity 是 wxc2 在 cpu0 上调度实体的父亲，这个一会在结构图中可以看

到。
8） 每个调度实体都会指向一个它它所属的 cfs_rq,同时还有一个它所在的 cfs_rq.对于代表组的调度实体来说，它的 cfs_rq 是其 parent 所在组的 cfs_rq,而它自己的 cfs_rq,用 my_q 指向，是它所在的组的 cfs_rq

3 一些与组调度相关的数组结构
1) 结构体组：

```
/* task group related information */
struct task_group {
#ifdef CONFIG_CGROUP_SCHED
    struct cgroup_subsys_state css;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    /* schedulable entities of this group on each cpu */
    struct sched_entity **se;每个 cpu 上都有一个调度实体，代表这个组，不是单个 task 的调度实体
    /* runqueue "owned" by this group on each cpu */
    struct cfs_rq **cfs_rq;每个 cpu 上都有一个 cfs_rq,属于这个组的
    unsigned long shares;
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;

    struct rt_bandwidth rt_bandwidth;
#endif

    struct rcu_head rcu;
    struct list_head list;

    struct task_group *parent;  这三个代表着组与组之间的亲戚
    struct list_head siblings;
    struct list_head children;
};
```

2） struct rq{
...

struct list_head leaf_cfs_rq_list; 用来链接在此 cpu 上所有组的 cfs_rq
...
}
3)
struct cfs_rq {

...
#ifdef CONFIG_FAIR_GROUP_SCHED
//Note here: cfs has rq pointer when group schedule
struct rq *rq; /* cpu runqueue to which this cfs_rq is attached */ 指向一个 run queue

/*
* 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.
*/
struct list_head leaf_cfs_rq_list; 链表，把一个 runqueue 上的所有 cfs_rq 链接在一起
struct task_group *tg; /* group that "owns" this runqueue */ 哪个 group 来的？
...
}
4） struct sched_entity {
。。。
#ifdef CONFIG_FAIR_GROUP_SCHED
struct sched_entity *parent; 也有父亲了~以前是野孩子 其父亲一般是其所在 group 的父亲的在同样 cpu 上的调度实体
/* rq on which this entity is (to be) queued: */
struct cfs_rq *cfs_rq; 这两 个 cfs_rq 挺绕的：cfs_rq 这个指针指向的是 parent->my_q
/* rq "owned" by this entity/group: */
struct cfs_rq *my_q;自己的 cfs_rq
#endif
};

4 组调度结构图
请参照附件，手工画得，因为不会用做图软件 汗~

右上角部分：两 个组之前是父子关系 每个组旁边的三角形是代表组的调度实体

5 具体的操作函数

先看两 个组：

1）/* Default task group.
 * Every task in system belong to this group at bootup.
 */
struct task_group init_task_group;

2) /*
 * Root task group.
 * Every UID task group (including init_task_group aka UID-0) will
 * be a child to this group.
 */

init_task_group 是系统初始化时进程的所在组。

root_task_group 是所有 group 的祖宗。

在 sched_init()里面，有对它的初始化过程。

```
sched_init(){
.....
#ifdef CONFIG_FAIR_GROUP_SCHED
        init_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
```

请留意这儿， ptr 加一个偏移，这段空间内存放着在所有 cpu 上的代表组的调度实体的指针。

```
        init_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

#ifdef CONFIG_USER_SCHED
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
#endif /* CONFIG_USER_SCHED */
...
}
```

由于 page_alloc 还没有建立，所以用 alloc_bootmem()，上面的过程建立了一个 task-group 里面的**se,**cfs_rq 部分。。。即两 个指针数组。

留意这句：

```
   init_task_group.parent = &root_task_group;
```

可见 init_task_group 的父亲也是 root_task_group.

接着往下，有一个对每个 cpu 的注册过程，这个过程和一个组创建并注册的过程类似，简单分析下。
这个过程只简我感兴趣的分析：

```
for_each_possible_cpu(i) {
        struct rq *rq;

        rq = cpu_rq(i);
        spin_lock_init(&rq->lock);
        rq->nr_running = 0;
        init_cfs_rq(&rq->cfs, rq);初始化运行列队自己的 cfs_rq，不是组的哦~
        init_rt_rq(&rq->rt, rq);
#ifdef CONFIG_FAIR_GROUP_SCHED
        init_task_group.shares = init_task_group_load;
        INIT_LIST_HEAD(&rq->leaf_cfs_rq_list);
#ifdef CONFIG_CGROUP_SCHED
。。。

        init_tg_cfs_entry(&init_task_group, &rq->cfs, NULL, i, 1, NULL);  这个函数很重要。。。把 initi_task_group,和 运行列队的 cfs_rq 绑定。
#elif defined CONFIG_USER_SCHED
        root_task_group.shares = NICE_0_LOAD;
        init_tg_cfs_entry(&root_task_group, &rq->cfs, NULL, i, 0, NULL);同理。
。。。


   init_tg_cfs_entry(&init_task_group,
                &per_cpu(init_cfs_rq, i),
                &per_cpu(init_sched_entity, i), i, 1,
                root_task_group.se[i]);
。。

}
```

init_tg_cfs_entry()这个函数做了许多好事，提出表扬。
急不可奈地先看看这厮到底做了啥捏：

```
#ifdef CONFIG_FAIR_GROUP_SCHED
static void init_tg_cfs_entry(struct task_group *tg, struct cfs_rq *cfs_rq,
                struct sched_entity *se, int cpu, int add,
```

struct sched_entity *parent) 这几个参数分别是：要初始化的组 + 要初始化的 cfs_rq + 要初始化的 se + CPU + 是否把 cfs 挂在 rq 上 + 代表组的父亲的调度实体

```
{
   struct rq *rq = cpu_rq(cpu);
   tg->cfs_rq[cpu] = cfs_rq;先把指针真满
   init_cfs_rq(cfs_rq, rq);把它和运行队列关链一下。 下面我们会分析这个初始化做什么事情
   cfs_rq->tg = tg;从这两 句我感觉到，指针真是个好东西啊：随意抽像出来一个玩意儿，就可以做为一个接口，把两 个东西绑在一起，让他们有关系。。。这里它就绑定了组和 rq
   if (add)
      list_add(&cfs_rq->leaf_cfs_rq_list, &rq->leaf_cfs_rq_list);
   tg->se[cpu] = se;也把 se 放进指针数组
   /* se could be NULL for init_task_group */
   if (!se)
      return;

   if (!parent) 这儿有意思：没有父亲的话，se 指向的 cfs_rq 就是当前运行队列的 cfs_rq,这个条件在初始化“根组”和初”始化组“时成立。
      se->cfs_rq = &rq->cfs;
   else
      se->cfs_rq = parent->my_q; 否则呢，就指向父亲所拥有的 cfs_rq

   se->my_q = cfs_rq;然后把自己所在组的 cfs_rq 作为自己的队列
   se->load.weight = tg->shares; 共享组的 load 值。
   se->load.inv_weight = 0;
   se->parent = parent; 亲吻下自己的父母
}
#endif
```

看看上面那个如何把 cfs_rq 和运行队列关联？

```
static void init_cfs_rq(struct cfs_rq *cfs_rq, struct rq *rq)
{
   cfs_rq->tasks_timeline = RB_ROOT; 初始化下自己的红黑结点
   INIT_LIST_HEAD(&cfs_rq->tasks);
#ifdef CONFIG_FAIR_GROUP_SCHED
   cfs_rq->rq = rq; 我能找到你，你也能找到我，这样才是稳固的关系。。。。
#endif
   cfs_rq->min_vruntime = (u64)(-(1LL << 20)); 默认一个 cfs_rq 的最左结点的值
```

```
}
```

上面这些分析大体代表了“根组”和“初始化组”在系统初始化的过程中相关流程。

来看一个组怎么被创建的吧。

```
/* allocate runqueue etc for a new task group */
struct task_group *sched_create_group(struct task_group *parent)  parent 这个参数一般在创建的时候会指定，比如是上一个进程组，没有的话可能是前面提到的那两 个组
{
    struct task_group *tg;
    unsigned long flags;
    int i;

    tg = kzalloc(sizeof(*tg), GFP_KERNEL);  申请组结构体的内存空间
    if (!tg)
        return ERR_PTR(-ENOMEM);

    if (!alloc_fair_sched_group(tg, parent))  恩，这个比较好玩儿，申请那个指针数组，一会儿看，
        goto err;

    if (!alloc_rt_sched_group(tg, parent))
        goto err;

    spin_lock_irqsave(&task_group_lock, flags);
    for_each_possible_cpu(i) {  把申请到的组向每一个 cpu 注册
        register_fair_sched_group(tg, i);
        register_rt_sched_group(tg, i);
    }
    list_add_rcu(&tg->list, &task_groups);

    WARN_ON(!parent); /* root should already exist */

    tg->parent = parent;  再亲吻一下自己的父母
    INIT_LIST_HEAD(&tg->children);
    list_add_rcu(&tg->siblings, &parent->children);
    spin_unlock_irqrestore(&task_group_lock, flags);

    return tg;

err:
    free_sched_group(tg);
    return ERR_PTR(-ENOMEM);
}
```

这个流程不审很清晰的。。。。申请空间-->申请指针数组-->向 cpu 的运行队列注册

看看这两 个函数做什么。

```
static
int alloc_fair_sched_group(struct task_group *tg, struct task_group *parent)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se, *parent_se;
    struct rq *rq;
    int i;

    tg->cfs_rq = kzalloc(sizeof(cfs_rq) * nr_cpu_ids, GFP_KERNEL);
    if (!tg->cfs_rq)
        goto err;
    tg->se = kzalloc(sizeof(se) * nr_cpu_ids, GFP_KERNEL);
    if (!tg->se)
        goto err;
```

上面两 步不难，申请指向每个 cpu 的 cfs_rq 和 se 数组

```
    tg->shares = NICE_0_LOAD;

    for_each_possible_cpu(i) {  向每一个 cpu 注册下申请的 cfs_rq 和 se 结构体。
        rq = cpu_rq(i);

        cfs_rq = kmalloc_node(sizeof(struct cfs_rq),
                GFP_KERNEL|__GFP_ZERO, cpu_to_node(i));
        if (!cfs_rq)
            goto err;

        se = kmalloc_node(sizeof(struct sched_entity),
                GFP_KERNEL|__GFP_ZERO, cpu_to_node(i));
        if (!se)
            goto err;

        parent_se = parent ? parent->se[i] : NULL;  判断当前组有父亲么？有的话其父亲在此 cpu 上的调度实体即为它的调度实体的父亲。。真拗口
        init_tg_cfs_entry(tg, cfs_rq, se, i, 0, parent_se);这个是之前我们分析过的，注意其参数
    }

    return 1;

 err:
    return 0;
}
```

上面这个申请函数过程还是蛮顺利的。
下面看看这个注册更简单，它就是把申请到的 cfs_rq 挂到运行队列的链表上。。。

这样一个组的创建过程分析完了。

最后再分析一个进程在不同组之间移动的情况：

```
/* change task's runqueue when it moves between groups.
 *    The caller of this function should have put the task in its new group
 *    by now. This function just updates tsk->se.cfs_rq and tsk->se.parent to
 *    reflect its new group.
 */
void sched_move_task(struct task_struct *tsk)
{
   int on_rq, running;
   unsigned long flags;
   struct rq *rq;

   rq = task_rq_lock(tsk, &flags);  如果是单进程调度，返回的是运行队列的 cfs_rq 结构体，否则返回调度实体指向的 cfs_rq

   update_rq_clock(rq);

   running = task_current(rq, tsk);
   on_rq = tsk->se.on_rq;

   if (on_rq)
      dequeue_task(rq, tsk, 0);  如果还在列上，出列吧，要移走了嘛
   if (unlikely(running))不希望它还在运行，如果是的话，就让它停止？ 或者重新入列？
      tsk->sched_class->put_prev_task(rq, ts)

   set_task_rq(tsk, task_cpu(tsk));  设置新的运行队列  下面有分析

#ifdef CONFIG_FAIR_GROUP_SCHED
   if (tsk->sched_class->moved_group)
      tsk->sched_class->moved_group(tsk); 下面有分析这个函数
#endif

   if (unlikely(running))
      tsk->sched_class->set_curr_task(rq);  试着让它运行 。。。这个函数的逻辑也没看懂
   if (on_rq)
      enqueue_task(rq, tsk, 0);  正式入列    下面分析它
```

```
    task_rq_unlock(rq, &flags);
}
#endif
```

这个流程的逻辑我是比较晕的，我搞不清楚为什么会这么做？
当一个进程从一个组移到另一个组的时候，这些操作具体的逻辑我不太懂，，需要再看下组调度理论性的东西。

set_task_rq 将重新设置一个 task 的运行列队：

```
/* Change a task's cfs_rq and parent entity if it moves across CPUs/groups */
static inline void set_task_rq(struct task_struct *p, unsigned int cpu)
```

这儿设置的可是一个进程哦，一个进程的调度实体。。。与代表组的调度实体要分清楚

```
{
#ifdef CONFIG_FAIR_GROUP_SCHED
    p->se.cfs_rq = task_group(p)->cfs_rq[cpu];
    p->se.parent = task_group(p)->se[cpu];
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    p->rt.rt_rq  = task_group(p)->rt_rq[cpu];
    p->rt.parent = task_group(p)->rt_se[cpu];
#endif
}
```

这样我们大体有个概念了。。。在组高度中，一个组内的进程的父亲都是代表这个组的调度实体。它们指向的 cfs_rq 就是当前组所在这个 cpu 上拥有的 cfs_rq

下面这个函数守成了进程移到新的组的珠一些信息的更新

```
#ifdef CONFIG_FAIR_GROUP_SCHED
static void moved_group_fair(struct task_struct *p)
{
    struct cfs_rq *cfs_rq = task_cfs_rq(p);

    update_curr(cfs_rq);
    place_entity(cfs_rq, &p->se, 1);
}
#endif
```

最后我们看这个入列的函数吧：

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

    for_each_sched_entity(se) {
```

这个 for 还是很有趣

```
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, wakeup);
        wakeup = 1;
    }

    hrtick_update(rq);
}
```

取得其 cfs_rq 非组调度时，返回当前运行队列的 cfs_rq,组调度时，返回的是 se 将要调度到的 cfs_rq

入列了 可能要初始化时间啊之类的

看看 for 循环：

```
#define for_each_sched_entity(se) \
        for (; se; se = se->parent)
```

这个在入列出列时经常遇到。如果打开了组调度，就是上面这样。

举例，组 A 里的一个进程 a 要加入到组 B 在运行列队 cpu0 上的 cfs_rq。这时进程的 se->cfs_rq 应该指向的时在 cpu0 上的组 B 的 cfs_rq。

首先执行一些出队入队操作。当这个环节完了后，还在对组 A 这个调度实体进行更新，这样往上遍历，把所有组 A 的父亲们遍历一次，都执行了入列操作。

为什么这么做呢？ 考虑中~

后记

上面这些是我看组调度中想到的一部分。。。。后面看完 cgroup 调度，应该会更有意思的。

(end)