析内核中cfs调度器相关代码 [参考blog](http://blog.csdn.net/zzsfqiuyigui/article/details/7867251)

CFS允许每个进程运行一段时间、循环轮转、选择运行最少的进程作为下一个运行进程，而不再采用分配给每个进程时间片的做法了，CFS在所有可运行进程总数基础上计算出一个进程应该运行多久，而不是依靠nice值来计算时间片。nice值在CFS中被作为进程获得的处理器运行比的权重：越高的nice值（越低的优先级）进程获得更低的处理器使用权重，这是相对默认nice值进程的进程而言的；相反，更低的nice值（越高的优先级）的进程获得更高的处理器使用权重。

数据结构

运行队列

```
struct cfs_rq {
	struct load_weight load;/*运行负载*/
	unsigned long nr_running;/*运行进程个数*/

	u64 exec_clock;
	u64 min_vruntime;/*保存的最小运行时间*/

	struct rb_root tasks_timeline;/*运行队列树根*/
	struct rb_node *rb_leftmost;/*保存的红黑树最左边的
	节点，这个为最小运行时间的节点，当进程
	选择下一个来运行时，直接选择这个*/

	struct list_head tasks;
	struct list_head *balance_iterator;

	/*
	 * 'curr' points to currently running entity on this cfs_rq.
	 * It is set to NULL otherwise (i.e when none are currently running).
	 */
	struct sched_entity *curr, *next, *last;

	unsigned int nr_spread_over;

#ifdef CONFIG_FAIR_GROUP_SCHED
	struct rq *rq;	/* cpu runqueue to which this cfs_rq is attached */

	/*
	 * 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;
	struct task_group *tg; /* group that "owns" this runqueue */

#ifdef CONFIG_SMP
	/*
	 * the part of load.weight contributed by tasks
	 */
	unsigned long task_weight;

	/*
	 *   h_load = weight * f(tg)
	 *
	 * Where f(tg) is the recursive weight fraction assigned to
	 * this group.
	 */
	unsigned long h_load;

	/*
	 * this cpu's part of tg->shares
	 */
	unsigned long shares;

	/*
	 * load.weight at the time we set shares
	 */
	unsigned long rq_weight;
#endif
#endif
};
```

运行实体结构为sched_entity，所有的调度器都必须对进程运行时间做记账。CFS不再有时间片的概念，但是他也必须维护每个进程运行的时间记账，因为他需要确保每个进程只在公平分配给他的处理器时间内运行。CFS使用调度器实体结构来最终运行记账。

```
/*
 * CFS stats for a schedulable entity (task, task-group etc)
 *
 * Current field usage histogram:
 *
 *     4 se->block_start
 *     4 se->run_node
 *     4 se->sleep_start
 *     6 se->load.weight
 */
struct sched_entity {
	struct load_weight	load;		/* for load-balancing */
	struct rb_node		run_node;
	struct list_head	group_node;
	unsigned int		on_rq;

	u64			exec_start;
	u64			sum_exec_runtime;
	u64			vruntime;/*存放进程的虚拟运行时间,用于调度器的选择*/
	u64			prev_sum_exec_runtime;

	u64			last_wakeup;
	u64			avg_overlap;

	u64			nr_migrations;

	u64			start_runtime;
	u64			avg_wakeup;

	u64			avg_running;

#ifdef CONFIG_SCHEDSTATS
	.....
/*需要定义相关宏*/
};
```

调度器的实体作为一个名为se的成员变量，潜入在进程描述符struct task_struct内。具体的调度类:

```
/*
 * All the scheduling class methods:
 */
static const struct sched_class fair_sched_class = {
	.next			= &idle_sched_class,/*下一个为idle进程*/
	.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,

	.get_rr_interval	= get_rr_interval_fair,

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

对于从运行队列中删除函数dequeue_task_fair

```
/*
 * The dequeue_task method is called before nr_running is
 * decreased. We remove the task from the rbtree and
 * update the fair scheduling stats:
 */
static void dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep)
{
	struct cfs_rq *cfs_rq;
	struct sched_entity *se = &p->se;

	for_each_sched_entity(se) {/*考虑了组调度*/
		cfs_rq = cfs_rq_of(se);/*获得se对应的运行队列*/
		dequeue_entity(cfs_rq, se, sleep);
		/* Don't dequeue parent if it has other entities besides us */
		if (cfs_rq->load.weight)
			break;
		sleep = 1;
	}
	/*更新hrtick*/
	hrtick_update(rq);
}
/*删除动作发生在进程阻塞(变为不可运行状态)
或者终止时(结束运行)*/
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)
{
	/*
	 * Update run-time statistics of the 'current'.
	 */
	update_curr(cfs_rq);

	update_stats_dequeue(cfs_rq, se);
	if (sleep) {
#ifdef CONFIG_SCHEDSTATS
		if (entity_is_task(se)) {
			struct task_struct *tsk = task_of(se);

			if (tsk->state & TASK_INTERRUPTIBLE)
				se->sleep_start = rq_of(cfs_rq)->clock;
			if (tsk->state & TASK_UNINTERRUPTIBLE)
				se->block_start = rq_of(cfs_rq)->clock;
		}
#endif
	}

	clear_buddies(cfs_rq, se);

	if (se != cfs_rq->curr)
		__dequeue_entity(cfs_rq, se);
	account_entity_dequeue(cfs_rq, se);
	update_min_vruntime(cfs_rq);
}
/*实现记账功能,由系统定时器周期调用*/
static void update_curr(struct cfs_rq *cfs_rq)
{
	struct sched_entity *curr = cfs_rq->curr;
	u64 now = rq_of(cfs_rq)->clock;/*now计时器*/
	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):
	 */
	 /*获得从最后一次修改负载后当前任务所占用的运行总时间*/
	/*即计算当前进程的执行时间*/
	delta_exec = (unsigned long)(now - curr->exec_start);
	if (!delta_exec)/*如果本次没有执行过，不用重新更新了*/
		return;
	/*根据当前可运行进程总数对运行时间进行加权计算*/
	__update_curr(cfs_rq, curr, delta_exec);
	curr->exec_start = now;/*将exec_start属性置为now*/

	if (entity_is_task(curr)) {/*下面为关于组调度的，暂时不分析了*/
		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);
	}
}
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
	      unsigned long delta_exec)
{
	unsigned long delta_exec_weighted;

	schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));
	/*总运行时间更新*/
	curr->sum_exec_runtime += delta_exec;
	/*更新cfs_rq的exec_clock*/
	schedstat_add(cfs_rq, exec_clock, delta_exec);
	/*用优先级和delta_exec来计算weighted以用于更细vruntime*/
	delta_exec_weighted = calc_delta_fair(delta_exec, curr);
	/*vruntime可以准确地测量给定进程的运行时间
	而且可知道谁应该是下一个被运行的进程*/

	/*更新进程的vruntime*/
	curr->vruntime += delta_exec_weighted;
	update_min_vruntime(cfs_rq);
}
static inline unsigned long
calc_delta_fair(unsigned long delta, struct sched_entity *se)
{
	/*NICE_0_LOAD: 优先级0 的weight*/
	 /* 如果不是优先级0,就要调用calc_delta_mine计算delta的weight值*/
	if (unlikely(se->load.weight != NICE_0_LOAD))
		delta = calc_delta_mine(delta, NICE_0_LOAD, &se->load);

	return delta;
```

```
}
 /*在这里不打算详细分析calc_delta_mine (delta_exec,weight,lw),它的执行过程约为delta *= weight /
lw.
从这个函数中可以看到,如果进程的优先级为0,那么就是返回delta.
如果不为0,就会调用calc_delta_mine()对delta值进行修正.对上面对calc_delta_mine()的说明来看
,有如下关系: Delta = delta* NICE_0_LOAD/ se->load
Se->load值是怎么来的呢? 可以跟踪sys_nice(),就可以发现se->load
其它就是表示nice对应的load值,nice越低,值越大.
据此,就可以得到一个结论.在执行相同时间的条件下(delta相同),
高优先的进程计算出来的delta值会比低优先级的进程计算出来
的低.应此,高优先的进程就会位于rb_tree的左边,在下次调度的
时候就会优先调度.
*/
static unsigned long
calc_delta_mine(unsigned long delta_exec, unsigned long weight,
		struct load_weight *lw)
{
	u64 tmp;

	if (!lw->inv_weight) {
		if (BITS_PER_LONG > 32 && unlikely(lw->weight >= WMULT_CONST))
			lw->inv_weight = 1;
		else
			lw->inv_weight = 1 + (WMULT_CONST-lw->weight/2)
				/ (lw->weight+1);
	}

	tmp = (u64)delta_exec * weight;
	/*
	 * Check whether we'd overflow the 64-bit multiplication:
	 */
	if (unlikely(tmp > WMULT_CONST))
		tmp = SRR(SRR(tmp, WMULT_SHIFT/2) * lw->inv_weight,
			WMULT_SHIFT/2);
	else
		tmp = SRR(tmp * lw->inv_weight, WMULT_SHIFT);

	return (unsigned long)min(tmp, (u64)(unsigned long)LONG_MAX);
}

static void
account_entity_dequeue(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
	/*cfs_rq->load更新*/
	update_load_sub(&cfs_rq->load, se->load.weight);
	if (!parent_entity(se))
		dec_cpu_load(rq_of(cfs_rq), se->load.weight);
	if (entity_is_task(se)) {/*组调度相关*/
		add_cfs_task_weight(cfs_rq, -se->load.weight);
		list_del_init(&se->group_node);
	}
	/*运行个数减一*/
	cfs_rq->nr_running--;
```

```
        se->on_rq = 0;/*表示不再运行队列中*/
}
```

删除函数最终将由下面函数从红黑树里面删除

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        if (cfs_rq->rb_leftmost == &se->run_node) {
                struct rb_node *next_node;

                next_node = rb_next(&se->run_node);
                cfs_rq->rb_leftmost = next_node;
        }

        rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

向运行队列中添加项的函数为enqueue_task_fair完成

```
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;
        /*对于主调度，会对一个组中的所有进程进行操作*/
        for_each_sched_entity(se) {
                if (se->on_rq)
                        break;
                cfs_rq = cfs_rq_of(se);
                enqueue_entity(cfs_rq, se, wakeup);
                wakeup = 1;
        }

        hrtick_update(rq);
}
```

该函数更新相关调度信息后最终会调用下面函数插入运行进程的红黑树

```
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
        struct rb_node *parent = NULL;
        struct sched_entity *entry;
        s64 key = entity_key(cfs_rq, se);
        int leftmost = 1;

        /*
         * Find the right place in the rbtree:
         */
        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.
                */ /*key为被插入进程的vruntime*/
                if (key < entity_key(cfs_rq, entry)) {
                        link = &parent->rb_left;
                } else {
                        link = &parent->rb_right;
                        leftmost = 0;
                }
        }

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

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

可见CFS的运行队列布局是放在红黑树里面的，而这颗红黑树的排序方式是按照运行实体的vruntime来的。vruntime的计算方式在上面已经做了分析。

进程选择

CFS调度算法的核心是选择具有最小vruntine的任务。运行队列采用红黑树方式存放，其中节点的键值便是可运行进程的虚拟运行时间。CFS调度器选取待运行的下一个进程，是所有进程中vruntime最小的那个，他对应的便是在树中最左侧的叶子节点。实现选择的函数为

pick_next_task_fair

```
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 {/*此循环为了考虑组调度*/
                se = pick_next_entity(cfs_rq);
                set_next_entity(cfs_rq, se);/*设置为当前运行进程*/
                cfs_rq = group_cfs_rq(se);
        } while (cfs_rq);

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

        return p;
}
```

该函数最终调用__pick_next_entity完成实质工作

```
/*函数本身并不会遍历数找到最左叶子节点(是
所有进程中vruntime最小的那个),因为该值已经缓存
在rb_leftmost字段中*/
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
	/*rb_leftmost为保存的红黑树的最左边的节点*/
	struct rb_node *left = cfs_rq->rb_leftmost;

	if (!left)
		return NULL;

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

总结

CFS调度运行队列采用红黑树方式组织，红黑树种的key值以vruntime排序。每次选择下一个进程运行时即是选择最左边的一个进程运行。而对于入队和处队都会更新调度队列、调度实体的相关信息。