CFS源码阅读报告

一、CFS简介

CFS(Completely Fair Schedulre)是Linux的调度算法之一，在Linux kernel 2.6.23之后成为Linux系统预设的调度器。为了更好地理解CFS，我们可以设想一个“完全理想的多任务处理器”，每个进程都能够同时获得处理器的执行时间。在这种情况下，处理器的利用率可以达到最大。而CFS的主要思想是在真实的硬件上来模拟一个理想的多任务处理器。

CFS为了实现“公平”，引入以下几个概念：

- 动态时间片：CFS摒弃了之前调度器中使用的静态时间片，根据进程的优先级计算相应的动态时间片，计算方法如下：

$$idea\_runtime = sched\_period \times \frac{weight}{total\_weight}$$

其中，$idea\_runtime$ 是运行队列中所有任务各执行一个时间片所需要的总时间。$weight$是任务的权重，$total_weight$是所有任务的权重和。

- 虚拟运行时间：CFS每次调用虚拟运行时间最小的进程，实现基于红黑树，插入和删除的时间复杂度为$O(logN)$，寻找最小值的时间复杂度为$O(1)$，虚拟运行时间计算方法如下：

$$vruntime = vruntime + exec\_time \times \frac{NICE\_0\_LOAD}{Weight}$$

从计算方法可以看出，已执行时间$exec_time$越大，虚拟运行时间越大，而权重（优先级）$Weight$越大，虚拟运行时间越小，这样就可以保证优先级高的、运行时间少的进程优先抢占处理器，以实现“公平”。

二、CFS中重要的数据结构

2.1 rq

Linux kernel使用`struct rq`来描述运行队列，CFS为每一个CPU都维护了一个`rq`，每个`rq`中包含三个不同的调度队列，CFS调度对应的调度队列为`cfs_rq`，每个分配给CPU的进程或线程，都会被加入相应的运行队列中。

```
struct rq {
/* ... */
/* 三个调度队列：CFS调度，RT调度，DL调度 */
    struct cfs_rq cfs;
    struct rt_rq rt;
    struct dl_rq dl;

    /*
    * curr 当前占据CPU的进程
    * idle 空闲进程，CPU空闲时调用
    */
    struct task_struct *curr;
    struct task_struct *idle;
    struct task_struct *stop;

    /* CPU of this runqueue: */
```

```
    int cpu;
    int online;
};
```

2.2 cfs_rq

Linux kernel使用`strcut cfs_rq`描述CFS调度的运行队列，由下面的部分代码展示，我们可以看出，结构体内存储了红黑树的根节点`tasks_timeline`。还有一个需要注意的是数据成员`min_vruntime`，即队列中最小的虚拟运行时间。我们考虑当有一个新的实体进入运行队列时，该如何设定其虚拟运行时间是一个重要的问题。如果设置为0，则在相当一段长的时间内，新实体的虚拟运行时间都会比较小，这样以来会造成其他实体的"饥饿"。因此，我们维护数据成员`min_vruntime`，并以其为基础经过简单的计算作为新实体虚拟运行时间的初值，以此来避免"饥饿"的产生，此外，`min_vruntime`数据成员在其他地方也有用到，之后会进行阐述。

```
struct cfs_rq {
    struct load_weight load;    //总权重
    /*
    * nr_running: how many entity would take part in the sharing
    * the cpu power of that cfs_rq
    * h_nr_running: how many tasks in current cfs runqueue
    */
    unsigned int nr_running;
    unsigned int h_nr_running; /* SCHED_{NORMAL,BATCH,IDLE} */
    unsigned int idle_h_nr_running; /* SCHED_IDLE */
    u64 exec_clock;
    u64 min_vruntime;   //最小的虚拟运行时间
    #ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
    #endif
    /* root of rb_tree */
    struct rb_root_cached tasks_timeline;   //红黑树的根节点
    /*
    * '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;  //当前运行的实体
    struct sched_entity *next;  //下一个实体
    struct sched_entity *last;
    struct sched_entity *skip;
};
```

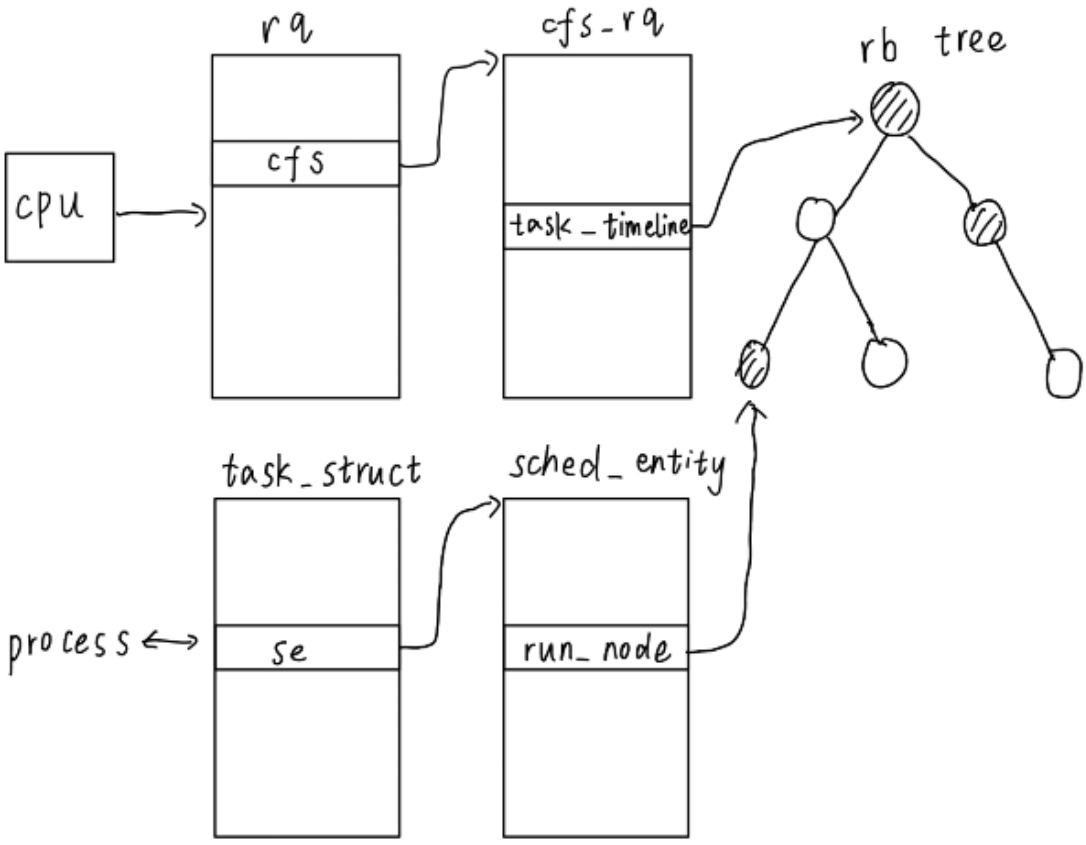
2.3 sched_entity

Linux kernel使用`struct sched_entity`描述调度实体，是CFS调度管理的对象。由以下的代码可以看出，结构体里存储了实体的权重、各种执行时间、虚拟运行时间等。

```
struct sched_entity {

    struct load_weight load;    //权重
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;
    u64 exec_start;     //开始执行的时间
    u64 sum_exec_runtime;   //所需的总时间
    u64 vruntime;   //虚拟运行时间
```

```
    u64 prev_sum_exec_runtime;    //已经执行的时间

    struct sched_statistics statistics;
    #ifdef CONFIG_FAIR_GROUP_SCHED
    int depth;
    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;
    /* cached value of my_q->h_nr_running */
    unsigned long runnable_weight;

};
```

各个结构体之间的关系如下图所示：（下图仅仅大概表述了各个结构体的关系，并不准确，事实上它们的联系更加紧密）



三、CFS中重要的函数

3.1 添加任务到运行队列中

添加任务到运行队列中主要由`enqueue_task_fair`和`enqueue_entity`函数实现。函数的主干如下所示，为了增强代码的可读性，这里只展示主干部分（之后的源码展示亦是如此）。总体的流程为：

- 由进程标识符`task_struct`获得进程对应的实体
- 判断当前实体是否已经属于某个执行队列，如果是则不再执行入队操作
- 由`rq`获得`cfs_rq`

- 更新当前运行调度实体的信息，例如虚拟运行时间
- 初始化要入队的实体的虚拟时间信息
- 更新执行队列的信息，例如总权重
- 修改新进入队列的虚拟运行时间
- 把实体加入红黑树，并把实体的on_rq标志位置为1
- 更新`rq`的信息

```
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;    //获得进程所对应的调度实体

    //......

    for_each_sched_entity(se) {
        if (se->on_rq)  //判断节点是否属于rq
            break;
        cfs_rq = cfs_rq_of(se);    //获得cfs_rq
        enqueue_entity(cfs_rq, se, flags);  //调用enqueue_entity函数
    }

    if (!se) {
        add_nr_running(rq, 1);
    }

    //......

    hrtick_update(rq);  //更新rq
}


static void
·(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 (renorm && curr)
        se->vruntime += cfs_rq->min_vruntime;

    update_curr(cfs_rq);     //更新当前运行调度实体的信息，例如虚拟运行时间

    if (renorm && !curr)
        se->vruntime += cfs_rq->min_vruntime;    //初始化要入队的实体的虚拟时间信息

    //......

    account_entity_enqueue(cfs_rq, se);      //更新执行队列的信息，例如总权重，链表，实体
个数

    //......

    if (flags & ENQUEUE_WAKEUP)
        place_entity(cfs_rq, se, 0);     //修改虚拟运行时间

    if (!curr)
```

```
        __enqueue_entity(cfs_rq, se);    //把实体加入红黑树

    se->on_rq = 1;  //实体成功入队，则把on_rq标志位置为1

    //......
}
```

3.2 把任务移除运行队列

把任务移除运行队列主要由`dequeue_task_fair`和`deueue_entity`函数实现，实际上是**3.1**的逆操作，代码逻辑基本相同。

```
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;   //获得进程所对应的调度实体

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);     //获得cfs_rq
        dequeue_entity(cfs_rq, se, flags);      //调用enqueue_entity函数

    //......

    if (!se) {
        sub_nr_running(rq, 1);
    }

    //......

    hrtick_update(rq);      //更新rq
}

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

    update_curr(cfs_rq);    //更新当前运行实体的信息，例如虚拟运行时间

    //......

    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);   //把实体节点从红黑树上删除

    se->on_rq = 0;      //实体已经从红黑树上删除，因此置on_rq数据成员为0
    account_entity_dequeue(cfs_rq, se);     //更新cfs_rq

    if (!(flags & DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->min_vruntime;   //如果进程不是进入睡眠状态，更新虚拟运行时间

    /* return excess runtime on last dequeue */
    return_cfs_rq_runtime(cfs_rq);

    update_cfs_group(se);

    if ((flags & (DEQUEUE_SAVE | DEQUEUE_MOVE)) != DEQUEUE_SAVE)
        update_min_vruntime(cfs_rq);    //更新最小虚拟运行时间
```

```
}
```

3.3 更新当前运行实体的信息

更新当前运行实体的信息由`update_curr`函数实现，除了在上述实体出队、入队的过程，还会在很多函数中被调用，所以在这里进行单独的说明。总体的流程为：

- 获得当前运行的实体
- 更新实体内的各种时间
- 计算实体虚拟运行时间
- 更新最小运行时间

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;   //当前运行的实体
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start;    //当前进程的运行时间
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;     //更新当前进程的开始时间

    //......

    curr->sum_exec_runtime += delta_exec;   //更新当前进程运行的总时间

    //......

    curr->vruntime += calc_delta_fair(delta_exec, curr);    //更新当前进程的虚拟运行时间
    update_min_vruntime(cfs_rq);    //更新最小虚拟运行时间

    //......

}
```

由上述更新虚拟运行时间的过程可以初步证明第一部分虚拟运行时间的计算方法的正确性。下面我们具体分析`update_min_vruntime`函数。我们在之前的说明可以知道，每一次都会调度虚拟运行时间最小的实体。当前运行的实体虚拟运行时间更新后，最小虚拟运行时间可能有两种：

- 当前运行的实体的虚拟运行时间仍然最小
- 红黑树中最左边节点对应的实体虚拟运行时间最小

我们可以选择以上实体中较小的虚拟运行时间进行更新。

```
static void update_min_vruntime(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;   //获得当前运行的实体
    struct rb_node *leftmost = rb_first_cached(&cfs_rq->tasks_timeline);    //获得红黑树的最左节点
```

```
    u64 vruntime = cfs_rq->min_vruntime;    //运行队列中的最小虚拟运行时间

    //分条件来选择正确的最小虚拟运行时间
    if (curr) {
        if (curr->on_rq)
            vruntime = curr->vruntime;
        else
            curr = NULL;
    }

    if (leftmost) { /* non-empty tree */
        struct sched_entity *se;
        se = rb_entry(leftmost, struct sched_entity, run_node);

        if (!curr)
            vruntime = se->vruntime;
        else
            vruntime = min_vruntime(vruntime, se->vruntime);
    }

    /* ensure we never gain time by being placed backwards. */
    cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime);
#ifndef CONFIG_64BIT
    smp_wmb();
    cfs_rq->min_vruntime_copy = cfs_rq->min_vruntime;
#endif
}
```

3.4 选择下一个占据CPU的进程

选择下一个占据CPU的进程主要有`pick_next_task_fair`函数实现，大致流程为：

- 处理上一个进程
- 选择合适的调度实体
- 更新cfs_rq的信息

```
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;
    unsigned long time;

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

    //......

    put_prev_task(rq, prev);                    //处理上一个进程

    do {
        se = pick_next_entity(cfs_rq, NULL);    //选择合适的调度实体
        set_next_entity(cfs_rq, se);            //更新cfs_rq的信息
```

```
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);

    //......

    return NULL;
}
```

处理上一个进程调用的是`put_prev_task`函数，在这个函数中主要调用了`put_prev_entity`函数，在这里我们仅仅展示后者。在函数中，如果实体仍然处于队列中，这说明更新进程仅仅失去了CPU，但是并没有进入睡眠状态，我们首先更新`cfs_rq`，然后重新把实体加入红黑树，最后把`cfs_rq`的`curr`数据成员置为空。

```
static void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
{

    if (prev->on_rq)
        update_curr(cfs_rq);    //更新cfs_rq

    //.......

    if (prev->on_rq) {
        update_stats_wait_start(cfs_rq, prev);
        __enqueue_entity(cfs_rq, prev);     //让实体重新加入红黑树
        update_load_avg(cfs_rq, prev, 0);
    }
    cfs_rq->curr = NULL;        //实体已经不再执行，cfs_rq的curr数据成员置为空
}
```

选择合适的调度实体由`pick_next_entity`函数实现，这部分即返回一个虚拟运行时间最小的实体，主要是为了处理`buddy`相关的情况，在此不再过多说明。

```
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    struct sched_entity *left = __pick_first_entity(cfs_rq);
    struct sched_entity *se;

    if (!left || (curr && entity_before(curr, left)))
        left = curr;

    se = left;

    //执行和buddy相关的逻辑

    return se;
}
```

更新`cfs_rq`的信息主要由`set_next_entity`函数实现。函数首先把实体从红黑树中删除，然后更改`cfs_rq`中的`curr`数据成员，使其指向要运行的实体，最后更新实体之前的运行时间。

这里需要注意，正在执行的实体并不会保存在红黑树中，当进程失去CPU时，如果进入睡眠状态，则不会进入红黑树，如果没有睡眠，会重新添加到红黑树中，这里的操作在上面的`put_prev_entity`里面执行。

```
static void
set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{

    if (se->on_rq) {
        update_stats_wait_end(cfs_rq, se);
        __dequeue_entity(cfs_rq, se);        //把实体从红黑树中移除
        update_load_avg(cfs_rq, se, UPDATE_TG);
    }

    update_stats_curr_start(cfs_rq, se);
    cfs_rq->curr = se;      //cfs_rq中的curr数据成员指向要运行的实体

    //......

    se->prev_sum_exec_runtime = se->sum_exec_runtime;    //更新实体之前运行的时间
}
```

3.5 创建新的进程

创建新的进程后的操作主要由`task_fork_fair`函数来实现，首先获得当前执行的实体，接着更新`cfs_rq`，之后调用`place_entity`函数修改虚拟运行时间，然后判断父进程的虚拟运行时间是否大于子进程，如果是，则交换虚拟运行时间，最后将子进程的虚拟运行时间减去最小虚拟运行时间，得到一个相对的值。

`place_entity`函数在进程创建时被调用，目的是调整新创建的子进程的虚拟运行时间，对其进行适当的增加（称为“惩罚”），此外，进程苏醒对的时候也会调用，会对虚拟运行时间进行适当的减少（称为“补偿”）。

新进程被创建完成后，会调用`wake_up_new_task`函数唤醒进程，中间会调用**3.1**提到的`enqueue_task_fair`函数,同时还会判断新进程是否能抢占当前进程。

```
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;    //获得当前执行的实体
    if (curr) {
        update_curr(cfs_rq);    //更新cfs_rq
        se->vruntime = curr->vruntime;
    }
    place_entity(cfs_rq, se, 1);    //修改虚拟运行时间

    if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
        swap(curr->vruntime, se->vruntime);     //交换父子进程的虚拟运行时间
        resched_curr(rq);
    }

    //......
```

```
    se->vruntime -= cfs_rq->min_vruntime;    //虚拟运行时间减去最小虚拟运行时间(可以理解为一个相对的值)
    rq_unlock(rq, &rf);
}
```

3.6 检查新成为可运行态的task是否能抢占当前运行的进程

检查新成为可运行态的task是否能抢占当前运行的进程主要由`check_preempt_wakeup`函数实现，一般而言，在创建进程、进程苏醒后调用。我们先来说明`wakeup_preempt_entity`函数，函数逻辑十分简单，主要作用为比较大小：

- 如果当前实体虚拟运行时间小于新实体的虚拟运行时间，返回-1
- 如果当前实体虚拟运行时间大于新实体的虚拟运行时间，且差距大于gran（粒度），返回1
- 如果当前实体虚拟运行时间大于新实体的虚拟运行时间，且差距小于等于gran，返回0

`check_preempt_wakeup`函数主要流程如下：

- 判断想要抢占的进程是否为当前正在执行的进程，如果是，则直接退出
- 更新`cfs_rq`
- 调用`wakeup_preempt_entity`函数，如果返回值为1，则抢占成功，否则抢占不成功

这里设置了粒度这一个概念，要求差距至少要大于粒度才能够执行，是为了避免抢占过于频繁，影响性能。

```
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;     //分别获得新实体和正在运行的实体
    struct cfs_rq *cfs_rq = task_cfs_rq(curr);
    int scale = cfs_rq->nr_running >= sched_nr_latency;
    int next_buddy_marked = 0;

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

    // ......

    update_curr(cfs_rq_of(se));

    // ......

    if (wakeup_preempt_entity(se, pse) == 1) {//调用wakeup_preempt_entity函数
        if (!next_buddy_marked)
            set_next_buddy(pse);
        goto preempt;
    }

    return;

preempt:
    resched_curr(rq);

    //......
    //执行和buddy相关的操作
```

```
}

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

    if (vdiff <= 0)      //如果当前实体虚拟运行时间小于新实体的虚拟运行时间
        return -1;

    gran = wakeup_gran(se);
    if (vdiff > gran)    //如果当前实体虚拟运行时间大于新实体的虚拟运行时间，且差距大于gran
        return 1;

    return 0;        //如果当前实体虚拟运行时间大于新实体的虚拟运行时间，且差距小于等于gran
}
```

3.7 中断触发函数

Linux定时周期性地产生中断，检查当前任务是否耗尽当前进程的时间片和是否应该抢占当前进程，中断产生后主要调用`task_tick_fair`函数，函数执行的过程中会调用`entity_tick`函数。如果运行队列中仅仅有一个进程，则会继续运行该程序，如果不是，则执行`check_preempt_tick`函数执行抢占。

`check_preempt_tick`函数首先计算理想的时间片和已经执行时间，若已执行时间大于理想时间片，则调用`resched_curr`函数修改运行队列的标志位，表示应该进行调度。之后调度器若检测到运行队列需要调度，则会选择虚拟运行时间最小的进程占据CPU。（**3.4**中提到的`pick_next_task_fair`函数）。

```
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);    //调用entity_tick函数
    }

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

static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{

    update_curr(cfs_rq);    //更新cfs_rq

    //......

    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);   //进程数量大于一执行check_preempt_tick函数
}


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);      //计算应该分配的时间片
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;      //计
算已经执行的时间
    if (delta_exec > ideal_runtime) {
        resched_curr(rq_of(cfs_rq));    //修改标志位，表示需要被调度
        clear_buddies(cfs_rq, curr);
        return;
    }

    //......

}
```

3.8 vruntime的溢出问题

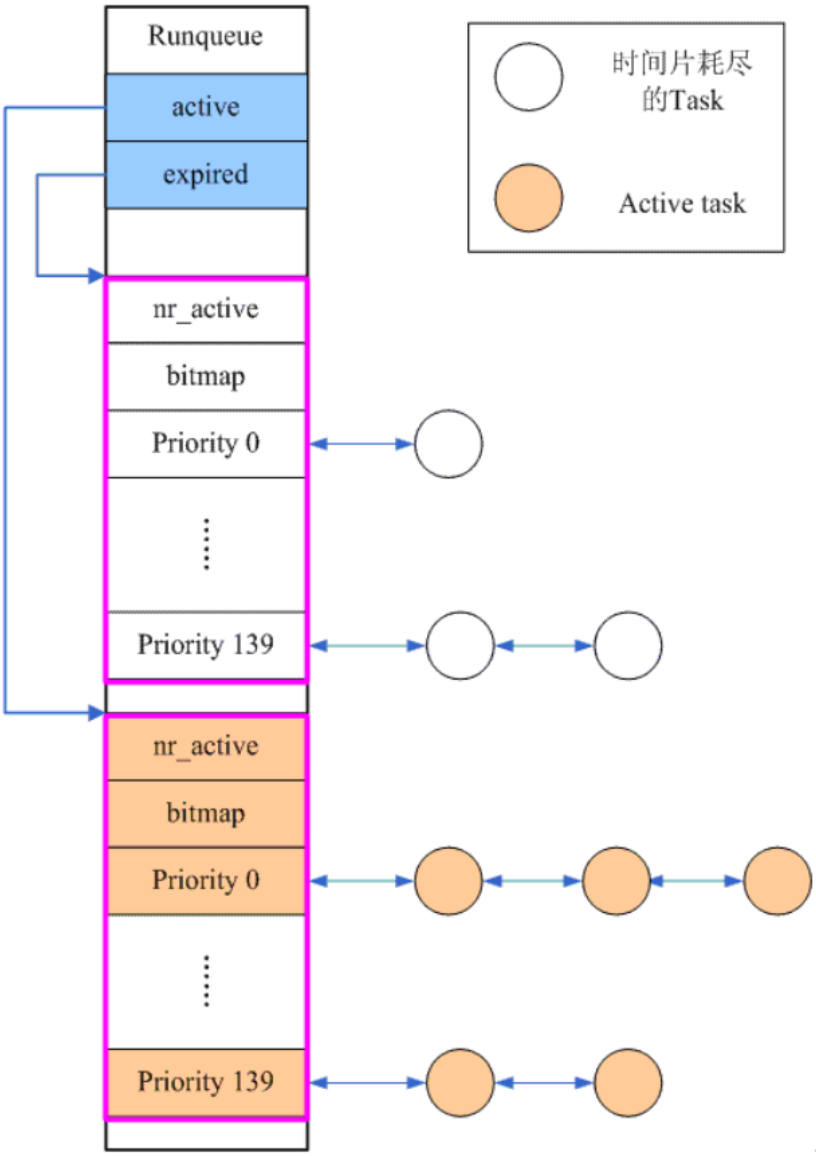
为了解决`vruntime`溢出的问题，在红黑树存储的事实上是$key = vruntime - min_vruntime$，是一个有符号数字，这样并不会因为溢出混乱相对的大小关系。而至于实体中的`vruntime`，它的存在仅仅为了确定相对大小关系，所以不需要对其进行溢出处理。（这要保证溢出后的值比溢出的值小）。我们可以用一个较为直观的例子进行理解：

```
unsigned char a = 251, b = 254;              //a、b均为八位无符号数

b += 5;                                      //b溢出，为00000011，即3

signed char c = a - 250,d = b - 250;         //这里假设250为最小运行时间，c = 1, d = 9

//我们可以看出，尽管b溢出，但是c、d的相对大小关系没有变化
```

四、CFS和$O(1)$调度算法的比较

4.1 $O(1)$调度算法简介

$O(1)$调度算法实际上是一个优先级调度算法。$O(1)$调度算法和CFS类似，为每一个CPU都维护了一个`runqueue`，$O(1)$调度算法的`runqueue`的结构如下所示：



$O(1)$调度算法中共支持140种优先级（0~139，其中0~99为实时进程的优先级，100~139为普通进程的优先级），共维护了两个优先级队列，分别来管理时间片用尽和未用尽的进程。同时，`runquene`还维护了两个`bitmap`（优先级位图），优先级位图用1bit代表一个优先级，某个优先级的进程处于可运行状态时，该优先级所对应的位就被置1。

随着进程调度的持续进行，`active task queue`的队列所有进程的时间片全都用尽，变为空，则交换`active task queue`和`expired task queue`，进行新一轮的调度。

通过上面的描述，我们可以得到，O(1)调度算法流程大致如下：

- 在`active bitmap`里，寻找最左端值为一的bit，确定优先级
- 找到优先级对应的`active task queue`，执行出队操作，如果此时队列为空，则置`active bitmap`相应的比特为0
- 进程时间片用完后，重新计算优先级
- 找到对应的`expired task queue`，执行入队操作，如果之前队列为空，则置`expired bitmap`对应的比特为1

上述所有操作，时间复杂度都为$O(1)$

需要注意的是，对于某些实时交互的进程是不能让用户感受到延迟的，因此$O(1)$调度算法需要判断进程是否为实时的（通过平均睡眠时间来判断），进而选择要加入的队列。

4.2 两种调度算法的对比

- 对于实时的进程，$O(1)$进行了特殊的处理，但是会带来相应的时延，并且如果实时进程较多，可能造成许多进程在一直等待下一轮调度，造成饥饿现象。而CFS算法则主要用于非实时的进程，实时进程由RT调度类处理，效果较$O(1)$算法好。
- $O(1)$调度算法的时间片是静态的，时间片由优先级直接确定，而CFS算法则是根据权重计算得来的，是动态的。相比于$O(1)$调度算法，CFS对于时间片的确定更加灵活。
- $O(1)$调度算法由优先级决定下一个要调用的进程，而CFS引入了虚拟运行时间的概念，综合已执行时间和权重共同决定，显得更加公平。
- CFS的性能略低于$O(1)$调度算法

五、总结

通过这次试验，我对Linux kernel关于多处理器调度算法有了更加深刻的了解，尤其是$O(1)$调度算法和CFS。一开始看到源码中那么长的函数是有点无从下手的，但是通过不断地抽丝剥茧，得到一些主干的内容，CFS实现的逻辑也清晰了起来。大量的源码阅读也让我了解到了Linux的优秀编程思想，对之后的学习有很大的帮助。

最后要感谢刘老师课堂上的讲解和助教提供的材料，才能让我顺利地完成本次课程作业！

六、参考资料

清华大学公开课程

Linux kernel源码

现代操作系统（原理与实现）