

# Completely Fair Scheduler代码分析

阅读 [OpenEuler kernel-4.19](#) 中与完全公平调度算法 (Completely Fair Scheduler) 相关的代码 [kernel/sched/fair.c](#), 分析CFS的具体实现过程, 思考CFS相比  $O(1)$  Scheduler 的优势。

(也可以阅读 [Linux kernel-5.12](#) 中与CFS相关的代码 [kernel/sched/fair.c](#))

## 一、CFS简介

Completely Fair Scheduler在Linux kernel 2.6.23之后成为系统预设的调度器。

CFS的思想是模拟一个理想的多任务处理器, 来最大化总体CPU利用率, 同时最大化交互性能。

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:

$$ideal\_runtime = sched\_period \times \frac{weight}{total\_weight} \quad (1)$$

其中,  $sched\_period$  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。

此外, CFS引入了  $vruntime$  的概念, 即程序运行的虚拟时间:

$$vruntime = vruntime + exec\_time \times \frac{NICE\_0\_LOAD}{Weight} \quad (2)$$

每次调用  $vruntime$  最小的进程, 以实现“公平”。

CFS的实现基于红黑树, 插入和删除的时间复杂度为  $O(\log N)$ , 寻找最小值的时间复杂度为  $O(1)$ 。

## 二、Linux进程调度数据结构

### 2.1 Linux 调度器

Linux内核默认提供了5种调度器, 使用 `struct sched_class` 对调度器进行抽象:

1. `stop_sched_class`: 优先级最高的调度器类。
2. `dl_sched_class`: 截止时间调度器类。
3. `rt_sched_class`: 实时调度器类。
4. `cfs_sched_class`: 完全公平调度器类, 此次报告的分析对象。
5. `idle_sched_class`: 空闲调度器类。

Linux内核还为用户程序提供了一些调度策略, 使他们能够选择调度器:

1. `SCHED_NORMAL`: 公平的分时调度策略, 由 CFS 来调度运行。
2. `SCHED_BATCH`: 针对不会与用户交互的批处理任务, 由 CFS 来调度运行。
3. `SCHED_IDLE`: 针对优先级最低的后台任务, 由 CFS 来调度运行。
4. `SCHED_RR`: 执行一定时间片后不再执行, 由 RT 来调度运行。

5. `SCHED_FIFO`：执行直至结束或被更高优先级任务抢占，由 `RT` 来调度运行。
6. `SCHED_DEADLINE`：类似EDF (Earliest Deadline First) 的调度策略，由 `DL` 来调度运行。

## 2.2 runqueue 运行队列

Linux内核使用 `struct rq` 来描述运行队列，具体定义可见 [kernel/sched/sched.h#Line794](#)，其中部分字段如下所示：

```
/*
 * 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 {

    /* ... */

    /* 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;

    /* ... */

};
```

1. 每个CPU都有一个对应的 `rq`。
2. 每个 `rq` 中包含三个不同的调度队列，与本次分析有关的是 `cfs_rq`。
3. 每个分配给CPU的进程或线程，都会被加入相应的运行队列 `xxx_rq` 中。

## 2.3 cfs\_rq

Linux使用 `struct cfs_rq` 描述CFS调度的运行队列，具体定义可见 [kernel/sched/sched.h#L491](#)，部分字段如下所示：

```

/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight  load;
    /*
     * 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;
#ifdef CONFIG_64BIT
    u64                 min_vruntime_copy;
#endif

    /* root of rb_tree */
    struct rb_root_cached  tasks_timeline;

    /*
     * '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;
    struct sched_entity *next;
    struct sched_entity *last;
    struct sched_entity *skip;

    /* ... */
};

```

## 2.4 task\_struct

Linux内核中，**进程和线程**都使用 `struct task_struct` 进行描述，具体定义可见 [include/linux/sched.h#Line598](#)，其中部分字段如下所示：

```

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* ... */

    /* 进程状态 */
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long        state;

    /*
     * prio: 动态优先级
     * static_prio: 静态优先级
     * normal_prio: 动态优先级
     * rt_priority: 实时优先级
     */

```

```

int          prio;
int          static_prio; /* nice value: -20 ~ 19 */
int          normal_prio;
unsigned int  rt_priority;

/* 调度器类, 调度实体, 任务组相关等 */
const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
struct task_group *sched_task_group;
#endif
struct sched_dl_entity dl;

/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */

/* Real parent process: */
struct task_struct __rcu *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;

/* ... */
};

```

## 2.5 sched\_entity

调度实体, 也是CFS调度管理的对象。Linux使用 `struct sched_entity` 进行描述, 具体定义可见 [include/linux/sched.h#Line446](#), 部分字段如下所示:

```

struct sched_entity {
    /* For load-balancing: */
    struct load_weight load;
    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;

#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;
#endif

    /* ... */

};

```

## 2.6 task group

一般来说，调度器对单个任务进行操作。

但有时可能需要对任务进行分组，并为每个任务组提供公平的CPU时间。

Linux引入任务分组的机制，设置任务组对CPU的利用率，从而避免影响其他任务的执行效率。

使用 `struct task_group` 来描述任务组，具体定义可见 [kernel/sched/sched.h#Line363](#)。

## 2.7 调度程序

调度程序依靠多个函数完成调度工作，如下为几个关键函数，具体可见 [kernel/sched/core.c](#)：

1. 主动调度——`schedule()`
2. 周期调度——`scheduler_tick()`
3. 进程唤醒时调度——`wake_up_process()`
4. 进程创建时调度——`sched_fork()`
5. 高精度时钟调度——`hrtick()`

每个函数会调用调度器相应的函数实现自己的功能。

## 三、CFS具体实现分析

Linux内核的调度类需要实现以下函数，`fair_sched_class` 中的定义可见 [kernel/sched/fair.c#Line10608](#)：

```

/*
 * All the scheduling class methods:
 */
const struct sched_class fair_sched_class = {
    .next                = &idle_sched_class,

    // 一个task变为可运行状态，将之添加到运行队列中

```

```

.enqueue_task      = enqueue_task_fair,
// 执行enqueue_task的逆操作
.dequeue_task      = dequeue_task_fair,

// 一个task自愿放弃CPU, 但仍为可运行状态
.yield_task        = yield_task_fair,
.yield_to_task     = yield_to_task_fair,

// 检查一个新成为可运行态的task是否应该抢占当前运行的进程
.check_preempt_curr = check_preempt_wakeup,

// 挑选下一个占据CPU的进程, 由schedule()调用
.pick_next_task     = pick_next_task_fair,
.put_prev_task      = put_prev_task_fair,

#ifdef CONFIG_SMP
.select_task_rq     = select_task_rq_fair,
.migrate_task_rq    = migrate_task_rq_fair,

.rq_online          = rq_online_fair,
.rq_offline         = rq_offline_fair,

.task_dead          = task_dead_fair,
.set_cpus_allowed   = set_cpus_allowed_common,
#endif

// task改变调度策略
.set_curr_task      = set_curr_task_fair,
// 每次时钟中断时, 由scheduler_tick()调用
.task_tick          = task_tick_fair,
// 一个新的task被创建
.task_fork          = task_fork_fair,

/* ... */

};

```

可以通过分析这些函数，了解CFS调度的具体流程。

## \* 扩展部分

fair.c中部分代码与Linux提供的与SMP、NUMA、*group\_sched*、*bandwidth*相关的机制有关，感兴趣的同学可以进一步了解分析。

## 四、参考资料

1. 作业中的视频和slide是清华大学计算机系操作系统课有关CFS的内容。（BTW，你也可以在[学堂在线](#)上观看他们的其他内容，这是操作系统课程的[课程主页](#)和[实验指导书](#)。）
2. Linux Kernel Organization文档中有关[CFS的设计](#)。
3. IBM的Tutorial中对于[CFS的介绍](#)。
4. 《New Scheduling Approaches for Linux OS》1.2节 *Linux process scheduler*。

5. 《Linux Kernel Development (3rd Edition)》第四章 *Process Scheduling*。
6. 《现代操作系统：原理与实现》
7. 如果想要进一步了解如 $O(1)$ 调度算法，可以阅读《Understanding the Linux Kernel (3rd Edition)》第七章 *Process\_Scheduling*（基于Linux kernel 2.6.11）。

## 五、提交要求

**6月6日23:59**之前将报告提交至CANVAS。