Completely Fair Scheduler代码分析

实验要求：

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？并结合教材ppt撰写代码分析报告。
Linux kernel-5.12的源代码地址为：https://github.com/torvalds/linux/tree/v5.12 。

CFS简介：

CFS作为主线调度器之一，也是最典型的O(1)调度器之一，在Linux2.6.23内核版本中引入，它最大的特点就是能保证任务调度的公平性。

以下为CFS的几大特征：

1. 给每一个进程设置一个虚拟时钟`vitural runtime (vruntime)`，如果一个进程得以执行，则随着执行时间增长，其`vruntime`将不断增大，没有得到执行的进程的`vruntime`保持不变，调度器每次选择具有最小的`vruntime`的进程来执行。vruntime计算公式为：

$$vruntime = exectime * \frac{NICE\_0\_LOAD}{current\_weight}$$

   其中，exectime为当前进程执行时间，current_weight为当前进程权重，使得权重较高的进程的vruntime增长较慢，拥有更多运行时间。

2. 基于红黑树实现进程调度，每次选取红黑树最左边的进程执行，搜索的时间复杂度为$O(1)$，插入和删除的时间复杂度为$O(logN)$。

Linux代码分析

数据结构：

本部分介绍Linux实现CFS调度算法相关的数据结构，具体定义在linux/sched.h at v5.12 · torvalds/linux (github.com)文件中。

struct rq

Linux通过`struct rq`结构体来描述运行队列，每个CPU都有自己的`struct rq`结构，其用于描述在此`CPU`上所运行的所有进程。

```
struct rq {
    /* runqueue lock: */
    raw_spinlock_t      lock;       //2.6.33版本新引入的自旋锁，确保struct rq操作的原子性。

    /*...*/

    struct cfs_rq       cfs;        //cfs调度队列
    struct rt_rq        rt;         //实时调度队列
    struct dl_rq        dl;         //截止时间调度队列
```

```
    /*...*/

    struct task_struct __rcu    *curr;  //当前占据CPU的进程
    struct task_struct  *idle;          //空闲进程，CPU空闲时调用
    struct task_struct  *stop;          //暂停进程
    unsigned long       next_balance;   //下次进行负载平衡执行时间

    /*...*/

    int         cpu;                //该队列所属的CPU
    int         online;
    /*...*/
}
```

`struct rq`中有三个队列，分别记录CFS调度，实时调度，截止时间调度的进程队列信息，并且记录着当前CPU的进程信息。

struct cfs_rq

Linux通过`struct cfs_rq`结构体来描述CFS调度的运行队列。

```
struct cfs_rq {
    struct load_weight  load;       //CFS运行队列的负载权重值
    unsigned int        nr_running;         //调度的实体数量
    unsigned int        h_nr_running;
    unsigned int        idle_h_nr_running;

    u64         exec_clock;         //运行时间
    u64         min_vruntime;       //最少的虚拟运行时间，调度实体入队出队时需要进行修改

    /*...*/

    struct rb_root_cached   tasks_timeline;  //红黑树，用于存放调度实体

    /*
     * '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;      //当前运行的调度实体
    struct sched_entity *next;      //下一个运行的调度实体
    struct sched_entity *last;      //CFS运行队列中排最后的调度实体
    struct sched_entity *skip;      //跳过的调度实体

    /*...*/
}
```

`struct cfs_rq`中记录了CFS调度算法执行的相关信息，包括执行时间、执行进程、调度实体数量等信息，以及采用的红黑树数据结构。

struct sched_entity

`sched_entity`是调度实体，即`cfs_rq`中直接调度的实体对象。

```
struct sched_entity {
    struct load_weight      load;           //调度实体的负载权重值
    struct rb_node          run_node;       //用于连接到CFS运行队列的红黑树中的节点
    struct list_head        group_node;     //用于连接到CFS运行队列的cfs_tasks链表中的节点
    unsigned int            on_rq;          //用于表示是否在运行队列中

    u64             exec_start;             //当前调度实体的开始执行时间
    u64             sum_exec_runtime;       //调度实体执行的总时间
    u64             vruntime;               //虚拟运行时间，这个时间用于在CFS运行队列中排队
    u64             prev_sum_exec_runtime;  //上一个调度实体运行的总时间

    u64             nr_migrations;      //负载均衡

    struct sched_statistics     statistics;     //统计信息

    //组调度相关信息
#ifdef CONFIG_FAIR_GROUP_SCHED
    int             depth;                      //任务组的深度
    struct sched_entity     *parent;            //指向调度实体的父对象
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq           *cfs_rq;            //指向调度实体归属的CFS队列
    /* rq "owned" by this entity/group: */
    struct cfs_rq           *my_q;              //指向归属于当前调度实体的CFS队列
    /* cached value of my_q->h_nr_running */
    unsigned long           runnable_weight;
#endif

    /*...*/
}
```

`sched_entity`存储了调度实体自身的权重和运行时间等信息，以及其他数据结构实体信息，如红黑树节点，cfs_task链表节点等，此外，定义了组调度的相关信息，如任务组深度，父对象等。

struct task_struct

`task_struct`是Linux中用于描述进程或线程的结构体，`sched_entity`相当于对调度单元做了抽象处理。`sched_entity`和`cfs_rq`是cfs调度器的特定成员，而`task`是应用于所有调度器的。

```
struct task_struct {
    /*...*/
    volatile long state;        //当前进程所处的状态，0表示TASK_RUNNIng,非0表示休眠或停止等

    /*...*/

    randomized_struct_fields_start

    void                *stack;     //栈信息
    atomic_t            usage;
```

```
    /* Per task flags (PF_*), defined further below: */
    unsigned int            flags;
    unsigned int            ptrace;

    /*...*/

    int             on_rq;          // 当前进程是否处于就绪队列上

    int             prio;
    int             static_prio;    // 进程的静态优先级，该优先级直接决定了非实时进程的 load_weight，从而决定了该进程对应调度实体的 vruntime 增长速度。
    int             normal_prio;
    unsigned int            rt_priority;

    const struct sched_class    *sched_class;       //进程所属的调度器类
    struct sched_entity     se;                     //cfs调度实体
    struct sched_rt_entity      rt;

     /*...*/
}
```

`task_struct`中记录了与进程和线程相关的信息，以及该进程调度的上层抽象实体。

struct task_group

`task_group`为组调度结构体，Linux支持将任务分组来对CPU资源进行分配管理，该结构中为系统中的每个CPU都分配了`struct sched_entity`调度实体和`struct cfs_rq`运行队列，其中`struct sched_entity`用于参与CFS的调度。

```
struct task_group {
     /*...*/

#ifdef CONFIG_FAIR_GROUP_SCHED

    struct sched_entity **se;       //每个CPU的调度实体

    struct cfs_rq       **cfs_rq;   //cfs运行队列
    unsigned long       shares;

    /*...*/

    struct task_group   *parent;    //该组的父节点
    struct list_head    siblings;   //兄弟节点链表
    struct list_head    children;   //子节点链表

    /*...*/
}
```

Linux调度器

Lniux提供了五种调度器，分别为：

- `stop_sched_class`：优先级最高的调度器。
- `dl_sched_class`：截止时间调度器。
- `rt_sched_class`：实时调度器。
- `cfs_sched_class`：完全公平调度器。

- `idle_sched_class`：空闲调度器。

sched_class抽象结构体

Linux内核抽象了一个调度结构体`struct sched_class`，这是一种典型的面向对象的设计思想，将共性的特征抽象出来封装成类，在实例化上面各个调度器的时候，可以根据具体的调度算法来实现，`struct sched_class`定义于linux/sched.h at v5.12 · torvalds/linux (github.com)文件中。

```
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
//进程入队
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
//进程出队
    void (*yield_task)   (struct rq *rq);
//当前进程放弃CPU
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);
//当前进程放弃CPU并交给某一指定进程

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int
flags);//检测是否能抢占当前进程

    struct task_struct *(*pick_next_task)(struct rq *rq);
// 选择下一个进程

    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
//设置上一个进程
    void (*set_next_task)(struct rq *rq, struct task_struct *p, bool
first);//设置下一个进程

    /*...*/

    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);
                                        //用于修改priority或者修改scheduler
class时的hook函数。
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                  int oldprio);         //修改优先级

    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
};
```

该结构体的变量主要为进程调度函数指针，通过将指针指向不同的功能函数，可以实现不同调度方法，达到C++中多态的效果。

CFS调度器的实现

CFS调度器在文件linux/fair.c at v5.12 · torvalds/linux (github.com)定义，创建一个名为`fair_sched_class`的结构体，并为结构体中的指针进行赋值，从而实现了CFS调度器。

```
DEFINE_SCHED_CLASS(fair) = {

    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task     = yield_task_fair,
    .yield_to_task      = yield_to_task_fair,

    .check_preempt_curr = check_preempt_wakeup,

    .pick_next_task     = __pick_next_task_fair,
    .put_prev_task      = put_prev_task_fair,
    .set_next_task          = set_next_task_fair,

    /*...*/

    .task_tick      = task_tick_fair,
    .task_fork      = task_fork_fair,

    .prio_changed       = prio_changed_fair,
    .switched_from      = switched_from_fair,
    .switched_to        = switched_to_fair,

    .get_rr_interval    = get_rr_interval_fair,

    .update_curr        = update_curr_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
    .task_change_group  = task_change_group_fair,
#endif

#ifdef CONFIG_UCLAMP_TASK
    .uclamp_enabled     = 1,
#endif
};
```

CFS调度函数

出队入队函数

`enqueue_task_fair`函数功能为向运行队列中添加进程。

```
if (se->on_rq)       //确认实体是否在队列中
    break;
cfs_rq = cfs_rq_of(se);
enqueue_entity(cfs_rq, se, flags);       //将实体加入队列

cfs_rq->h_nr_running++;
cfs_rq->idle_h_nr_running += idle_h_nr_running;

/* end evaluation on encountering a throttled cfs_rq */
if (cfs_rq_throttled(cfs_rq))
    goto enqueue_throttle;

flags = ENQUEUE_WAKEUP;
```

其中将调度实体加入队列的函数`enqueue_entity`：

```
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /*...*/
    update_curr(cfs_rq);         //更新当前任务运行时的统计信息，如果当前任务时间耗尽则进行调度
    if (renorm && !curr)
        se->vruntime += cfs_rq->min_vruntime;            //基于最小vruntime，设置新进程的vruntime
    update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);      //更新调度实体与CFS调度队列的负载
    se_update_runnable(se);
    update_cfs_group(se);                                //更新组任务权重
    account_entity_enqueue(cfs_rq, se);                  //将调度实体的权重加入CFS队列中

    /*...*/

    if (!curr)
        __enqueue_entity(cfs_rq, se);        //将调度实体加入红黑树中
    /*...*/
}
/*...*/
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_add_cached(&se->run_node, &cfs_rq->tasks_timeline, __entity_less);
}
```

在该函数中，调度器先更新调度实体和运行队列的负载和权重，然后将调度实体对应的红黑树节点加入队列管理的红黑树中。除此之外，`enqueue_entity`通过当前的虚拟执行时间和最小虚拟时间重置虚拟执行时间。

同理，出队函数`dequeue_task_fair`也主要依赖`dequeue_entity`实现，并通过函数`__dequeue_entity`将对应的红黑树节点移出队列，设置虚拟运行时间和修改相应的标志位，更新运行队列的负载和权重信息。

进程主动放弃CPU的处理函数

`yield_task_fair`函数

```
static void yield_task_fair(struct rq *rq)
{
    /*...*/
    if (unlikely(rq->nr_running == 1))      //如果红黑树中只有一个任务，则直接返回
        return;

    clear_buddies(cfs_rq, se);              //清除cfs运行队列的实体se

    if (curr->policy != SCHED_BATCH) {
        update_rq_clock(rq);

        update_curr(cfs_rq);

        rq_clock_skip_update(rq);
    }

    set_skip_buddy(se);
}
```

该函数先检查运行队列中任务数量，如果只有一个任务则直接返回。调用`clear_buddies`清除实体se。

`yield_to_task_fair`基于`yield_task_fair`函数，先检查目标任务状态，然后设置目标任务为下一个执行实体，接着调用`yield_task_fair`函数清除当前实体。

__pick_next_task_fair

`__pick_next_task_fair`函数主要依赖`pick_next_task_fair`函数实现。

```
static struct task_struct *__pick_next_task_fair(struct rq *rq)
{
    return pick_next_task_fair(rq, NULL, NULL);
}
```

`pick_next_task_fair`函数的主要功能如下：

```
if (prev)
    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);

p = task_of(se);
```

它在`cfs_rq`中选出新的调度实体，并将其设置为下一个可执行的调度实体，选取实体的函数`pick_next_entity`主要功能部分：

```
struct sched_entity *left = __pick_first_entity(cfs_rq); // 选出最左节点代表的进程
```

```
if (!left || (curr && entity_before(curr, left)))
    left = curr; // 如果选取进程失败或者进程虚拟运行时间小于当前进程，则选择当前进程

if (cfs_rq->skip && cfs_rq->skip == se) {    //如果选择的进程为跳过进程，则重新选取
    struct sched_entity *second;

    if (se == curr) {
        second = __pick_first_entity(cfs_rq); // 如果选择的进程就是当前进程，那么选择队列第一个进程
    } else {
        second = __pick_next_entity(se);
        if (!second || (curr && entity_before(curr, second)))
            second = curr;
    }

    if (second && wakeup_preempt_entity(second, left) < 1)
        se = second; // 如果选取进程可抢占最左节点代表的进程，则选择选取的进程作为下一个调度的进程
}
```

该函数先选取最左边节点代表的进程，然后检查进程选取状态，如果选择失败或跳过，则重新选取。如果选取的进程的虚拟运行时间小于当前进程，则选取当前进程。

put_prev_task_fair

`put_prev_task_fair`的功能是将当前进程加入到调度队列中，主要功能由`put_prev_entity`实现：

```
if (prev->on_rq) {
    update_stats_wait_start(cfs_rq, prev);
    /* Put 'current' back into the tree. */
    __enqueue_entity(cfs_rq, prev);     ////将调度实体加入红黑树中
    /* in !on_rq case, update occurred at dequeue */
    update_load_avg(cfs_rq, prev, 0);
}
```

同样通过`__enqueue_entity`函数将调度实体加入调度队列（红黑树）中。

set_next_task_fair

`set_next_task_fair`的功能为设置下一个运行的实体，主要功能由`set_next_entity`函数实现：

```
static void set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    /* 'current' is not kept within the tree. */
    //如果目标实体已经在运行队列中，则从运行队列中删去该实体。
    if (se->on_rq) {
        /*
         * Any task has to be enqueued before it get to execute on
         * a CPU. So account for the time it spent waiting on the
         * runqueue.
         */
        update_stats_wait_end(cfs_rq, se);
        __dequeue_entity(cfs_rq, se);
        update_load_avg(cfs_rq, se, UPDATE_TG);
    }

```

```
    update_stats_curr_start(cfs_rq, se);        //更新当前调度实体并开始计时。
    cfs_rq->curr = se;

    /*
     * Track our maximum slice length, if the CPU's load is at
     * least twice that of our own weight (i.e. dont track it
     * when there are only lesser-weight tasks around):
     */
    if (schedstat_enabled() &&
        rq_of(cfs_rq)->cfs.load.weight >= 2*se->load.weight) {
        schedstat_set(se->statistics.slice_max,
            max((u64)schedstat_val(se->statistics.slice_max),
                se->sum_exec_runtime - se->prev_sum_exec_runtime));
    }

    se->prev_sum_exec_runtime = se->sum_exec_runtime;    //更新实体的虚拟运行时间
}
```

task_tick_fair

`task_tick_fair` 函数的主要功能为更新运行时的各类统计信息，比如 `vruntime`， 运行时间、负载值、权重值等，并检查是否需要抢占，主要是比较运行时间是否耗尽，以及 `vruntime` 的差值是否大于运行时间等。`task_tick_fair` 函数主要通过调用 `entity_tick()` 来完成调度工作，`entity_tick()` 函数主要功能部分为：

```
static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
    // 更新当前进程的虚拟运行时间
    update_curr(cfs_rq);
    /*...*/
    if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
        check_preempt_tick(cfs_rq, curr); // 判断是否需要进行进程调度
}
```

`entity_tick()` 函数主要完成以下工作：

- 调用 `update_curr()` 函数更新进程的虚拟运行时间，这个前面已经介绍过。
- 调用 `check_preempt_tick()` 函数判断是否需要进行进程调度。

`check_preempt_tick()` 函数主要功能部分：

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;
    /*...*/
    // 计算当前进程可用的时间片
    ideal_runtime = sched_slice(cfs_rq, curr);

    // 进程运行的实际时间
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;

    // 如果进程运行的实际时间大于其可用时间片，那么进行调度
    if (delta_exec > ideal_runtime)
        resched_task(rq_of(cfs_rq)->curr);
```

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

`check_preempt_tick()` 函数主要完成以下工作：

- 通过调用 `sched_slice()` 计算当前进程可用的时间片。
- 获取当前进程在当前调度周期实际已运行的时间。
- 如果进程实际运行的时间大于其可用时间片，那么调用 `resched_task()` 函数进行进程调度。

update_curr_fair

`update_curr_fair` 函数的主要功能为更新当前实体的状态，通过 `update_curr` 函数实现：

```
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;         //获取当前时间片执行的时间
    if (unlikely((s64)delta_exec <= 0))          //如果执行时间小于等于0则直接返回。
        return;

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

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

    curr->sum_exec_runtime += delta_exec;        //更新当前进程的运行总时间
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    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);                 //更新当前队列的最小虚拟运行时间
    /*...*/
}
```

计算虚拟运行时间的函数 `calc_delta_fair` 主要通过调用 `__calc_delta` 函数实现：

```
static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct load_weight *lw)
{
    //lw为当前实体的权重，weight=NICE_0_LOAD
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;

    __update_inv_weight(lw);

    if (unlikely(fact >> 32)) {
        while (fact >> 32) {
            fact >>= 1;
            shift--;
```

```
        }
    }

    fact = mul_u32_u32(fact, lw->inv_weight);

    while (fact >> 32) {
        fact >>= 1;
        shift--;
    }

    return mul_u64_u32_shr(delta_exec, fact, shift);
}
```

该函数为了保证足够的精度，采用了“fixed-point arithmetic”，运行时间计算公式为：
$vruntime = (delta\_exec \times weight \times lw.inv\_weight) >> 32$

由此可以得到虚拟运行时间的计算公式为：$vruntime = delta\_exec \times \frac{weight}{lw.weight}$

总结

本次作业主要分析了Linux中CFS算法函数以及所依赖的数据结构，其中部分代码是为了实现其他调度算法以及进行安全检查，在分析的过程中予以省略。值得注意的一点是，在创建一个进程时就实现了对CFS调度相关内容和数据结构的初始化，对于一个进程，如果创建时未设定`prio`属性，则会采用CFS调度算法，否则采用实时调度算法。

通过本次作业，我对CFS的实现细节有了更好的了解，同时，分析阅读Linux内核源代码让我了解了许多C语言高级知识，在分析调度器实现时，Linux抽象了一个调度结构体`struct sched_class`，在实例化各个调度器的时候，可以将函数指针指向具体的调度算法来实现，通过C语言实现面向对象的设计，将共性的特征抽象出来封装成类。相比于C++，C的这种实现方法更加复杂，并且需要额外的宏定义，通过分析这段代码，让我对宏定义的理解更加深入。