Shanghai Jiao Tong University

CS353 Linux Kernel

Source Reading: CFS

Chao Gao

5142029014

June 13, 2017

1 Introduction

In this project, we are required to read the linux kernel source and totally understand a part of linux by the code level. There is no doubt that this will greatly help us to dig into the linux kernel and get to know how it works. And I try to know more about the CFS.
The **Completely Fair Scheduler** is the default process scheduler in the linux kernel from 2.6.23. It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance.
The original scheduler is of O(1) time complexty based on run queues, and CFS is baesd on red-black tree. In part 3 I will dig into the source code.

My OS environment is Ubuntu 16.04 LTS, the kernel version is **4.9.13**, all the scheduler part code is in kernel/sched folder, and the main part about CFS is in the file **kernel/sched/fair.c**.

2 Process scheduler

A good process scheduler should be designed as follows.

- Fairness: every process gets resonable CPU time.
- Efficient: always has a process running on the CPU.
- Response time: make the iteractive process quicker to get response.
- Thoughput: handle more processes in a specific time.

There are two kinds of processes, normal processes and real-time processes. CFS is used for normal processes.
The motivation of CFS is getting all tasks executed fairly.

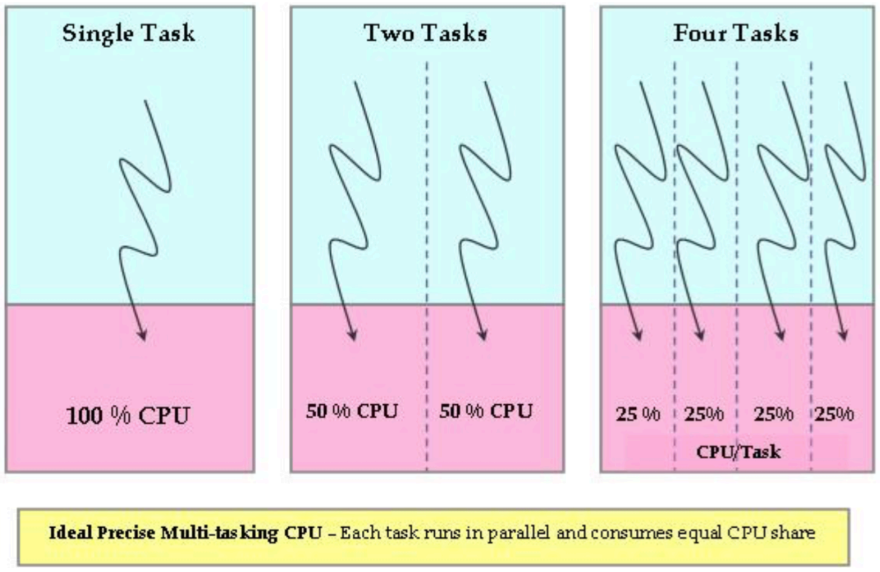
Single Task
100 % CPU
Two Tasks
50 % CPU
50 % CPU
Four Tasks
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

Figure 1: Figure 1.

Simply speaking about CFS algorithm in my own words, put all the tasks as nodes of a **red-black tree**, use **virtual tuntime** to measures the time that the task has used, the task with the least virtual time is the **leftmost node** of the tree, which has been cached. When the picked process is removed, the tree is updated.

3 Code detail

3.1 sched_entity

In process scheduling, process is not the actual entity. Because in linux, there exists not only process scheduling but also group scheduling. The struct of process is defined in linux/sched.h.

```
 struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
     struct thread_info thread_info;
#endif
     volatile long state; //−1 unrunnable,0 runnable,>0 stopped
```

```
    void *stack;
    atomic_t usage;
    unsigned int flags; //per process flags,defined below
    unsigned int ptrace;

    ...

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    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;

    ...
```

Here **struct sched_entity se** is the scheduling entity object. And struct sched_entity is also defined in linux/sched.h.

```
struct sched_entity {
    struct load_weight  load;  /* for load-balancing */
    struct rb_node      run_node;//node on red-black tree
    struct list_head    group_node;
    unsigned int        on_rq; //if on the run queue

    u64         exec_start;
    u64         sum_exec_runtime; //whole running time
    u64         vruntime;    //virtual run time
    u64         prev_sum_exec_runtime;

    u64         nr_migrations;
    struct cfs_rq        *cfs_rq;
    ...
```

There is a **sched_class** in task_struct which consists of scheduling functions. Because in different scheduling algorithms, the operations are different. Therefore, when we are using CFS, just create the corresponding sched_class and invoke corresponding scheduling functions.

```
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq,
                struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq,
                struct task_struct *p, int flags);
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq,
                struct task_struct *p, bool preempt);

    void (*check_preempt_curr) (struct rq *rq,
                struct task_struct *p, int flags);

    struct task_struct * (*pick_next_task) (struct rq *rq,
                        struct task_struct *prev,
                        struct pin_cookie cookie);
    void (*put_prev_task) (struct rq *rq,
                    struct task_struct *p);
    ...
    void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq,
                    struct task_struct *p, int queued);
    ...
};
```

Just take a look at some functions.

- enqueue_task: when a task is about to running, put the sched_entity to the red-black tree.
- dequeue_task: when a task is finished, remove the sched_entity from the red-black tree.
- pick_next_task: choose the next process.
- task_tick: process switch.

3.2 Vruntime

Vruntime is the very important concept in the CFS.
Let's recap the basic idea of CFS, in the ideal state, each process can get the same time slice, and run on the CPU at the same time. However, in fact at one time there is only one process occupying CPU. In other words, when a process is running, other processes have to wait. In order to achieve fairness, CFS must punish the currently running process so that the processes that are waiting are scheduled in the next time.
CFS uses vruntime to record the execution time of the process.
Every period counter generates a tick interrupt. At this time, function **scheduler_tick** would be invoked, which is defined in kernel/sched/core.c.

```
void scheduler_tick(void)
{
    ...
    raw_spin_lock(&rq->lock);
    update_rq_clock(rq);

    //invoke scheduler
    curr->sched_class->task_tick(rq, curr, 0);
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    raw_spin_unlock(&rq->lock);

    perf_event_task_tick();

    ...
```

Since the architecture is made up of modules. So if we use CFS, at this time program will jump to function **task_tick_fair**, which is defined in kernel/sched/-fair.c.

```
static void task_tick_fair(struct rq *rq,
        struct task_struct *curr, int queued)
{
    ...

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }
    ...
}
```

Then take a look at function **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);
    ...

    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
}
```

As the source code comments say, function **update_curr** is used to update the statistics of the run-time(vruntime), and kernel will deside how to schedule the processes.

```
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;
    //get the CPU time after the latest upate of load
    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;

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

    //add the delta_exec to the current execution time
    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    //calculate the vruntime
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);

```

```
        ...

}
```

Note that function **calc_delta_fair** translates realtime to vruntime. The priority is larger, the vruntime grows more slowly.

```
static inline u64 calc_delta_fair(u64 delta,
                struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

    return delta;
}
```

Take a look at function **__calc_delta**.

```
static u64 __calc_delta(u64 delta_exec,
                unsigned long weight, struct load_weight *lw)
{
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;

    __update_inv_weight(lw);

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

    /* hint to use a 32x32->64 mul */
    fact = (u64)(u32)fact * lw->inv_weight;

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

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

We can know **vruntime = real-execution-time * NICE_0_LOAD/lw.weight**.

3.3 CFS schduler class

Let's return back to the struct **sched_entity**.
There is a member called **cfs_rq**, this is the CFS runnning queue, which is defined in kernel/sched/sched.h.

```
struct cfs_rq {
    struct load_weight load;
    unsigned int nr_running, h_nr_running;

    u64 exec_clock;
    u64 min_vruntime;
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif
    struct rb_root tasks_timeline;

    //next scheduler node(leftmost node of red-black tree)
    struct rb_node *rb_leftmost;

    struct sched_entity *curr, *next, *last, *skip;
...


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

};
```

Note that **rb_node *rb_leftmost** is the leftmost node of the red-black tree, which will be the next scheduling node. And sched_entity ***curr** is the current running process, sched_entity ***next** means some urgent processes which have to run on the CPU immediately even with complict to CFS. And one important point is that the root of red-black tree is **rb_node * tasks_timeline**.

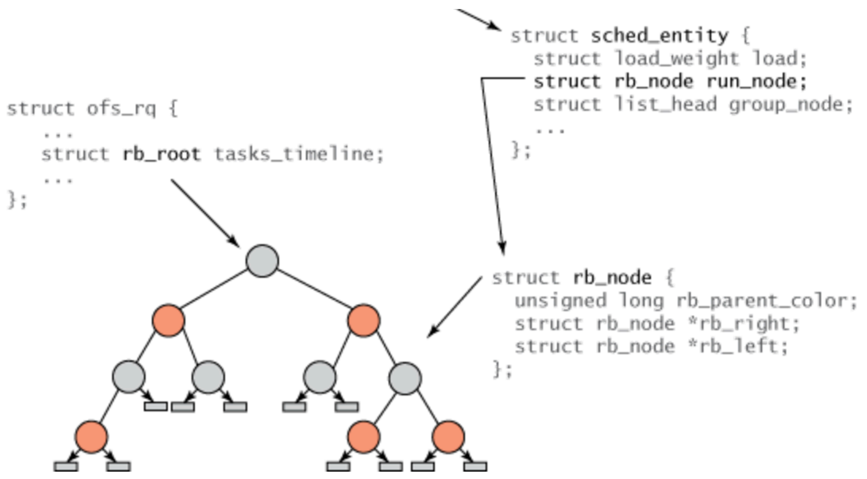

Figure 2: Figure 2.

Now we get to the very important stage, we get to know the CFS schedule class, which consists of all the functions with regard to scheduling algorithms. **fair_sched_class** in defined in kernel/fair.c.

```
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,
    .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_curr_task              = set_curr_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,
    ...
};
```

Give a partial list of the function.

- enqueue_task: called when a task enters a runnable state. It puts the scheduling entity into the red-black tree.
- dequeue_task: when a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree.
- check_preempt_curr: this function checks if a task that entered the runnable

state should preempt the currently running task.

- pick_next_task: this function choose the most appropriate task eligible to run next.
- set_curr_task: this function is called when a task changes its scheduling class or changes its task group.
- task_tick: this function is mostly called from time tick functions, it might lead to process switch. This derives the running preemption.

3.4 Process switch

Since we have known about CFS schduler class and vruntime, now we can turn to the process switch part.
In **pick_next_task** function which is defined in kernel/core.c, scheduler will do the scheduling. And next the CFS scheduler's function will be invoked, which is called **pick_next_task_fair** function. And the task chosen will start executing.

```
static struct task_struct *
pick_next_task_fair(struct rq *rq,
        struct task_struct *prev, struct pin_cookie cookie)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

    ...

    if (!cfs_rq->nr_running)
        goto idle;

    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);

    if (hrtick_enabled(rq))
        hrtick_start_fair(rq, p);

    return p;

    ...
}
```

We can see **pick_next_entity** function to pick the next sched entity, and remove the node in red-black tree and update the red-black tree by using **set_new_entity** function.
Here I dig into pick_next_entity function. And it is implemented by a core function called **__pick_first_entity**.

```
struct sched_entity *__pick_first_entity(
                        struct cfs_rq *cfs_rq)
{
    struct rb_node *left = cfs_rq->rb_leftmost;

    if (!left)
        return NULL;

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

We can see the leftmost node of red-black tree is chosen, which is the same as our knowledge learned in class. Our scheduler successfully works.

4 Conclusion

CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.
Through the code reading, I get to know more about CFS. In this report, I mainly cover the following parts.

- know the sched entity in scheduling.
- understand the vruntime.
- know the CFS scheduler class.
- basic process switch in CFS.

Of course this is not the end, I will get to know more about group scheduler in the future.
In conclusion, in this project, I mainly read the source code about CFS, and since the code is too large, I just get a brief understanding but it is still really helpful for me.

5 Summary

Though kernel source reading, I truly understand more about CFS algorithm, and the tough thing is linux kernel changes so rapidly that many resources are not useful at today's time. And it is a hot potato to "get through" the deep function trees, luckily I get over that.
In linux kernel course this semester, I really learn a lot about linux, with pains. But this is only a beginning, linux will be more frequently used in my next learning career.
Finally, thanks a lot to Prof.Chen for teaching and TAs for answering questions.