1. 哲学家就餐问题

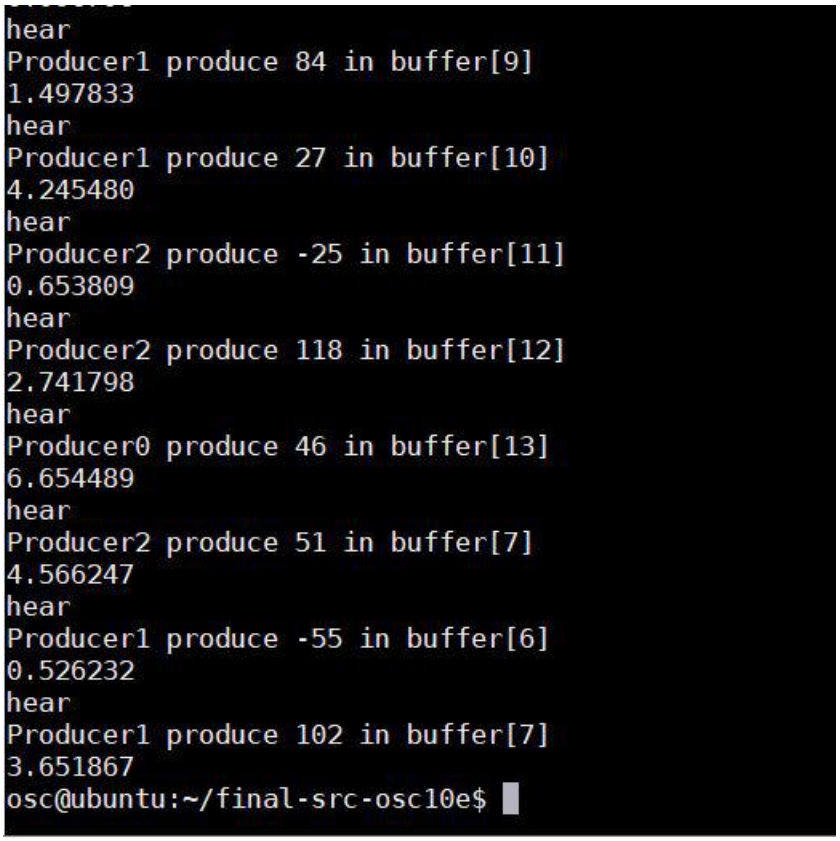
```
osc@ubuntu:~/final-src-osc10e$ gcc pthread-phi.c -o pp -lpthread -lrt
osc@ubuntu:~/final-src-osc10e$ ./pp
state: 0 0 0 0 1
4 is hungry
4 pick up 0
4 pick up 4,star to eat
state: 0 0 0 0 2
4 is eating
4 put down 4
4 put down 0
4 finshi eating
state: 0 0 0 0 0
4 is thinking
state: 0 0 0 0 1
4 is hungry
4 pick up 0
4 pick up 4,star to eat
```

（具体解释详见代码）

2. 生产者消费者问题

生产者

```
osc@ubuntu:~/final-src-osc10e$ gcc test2.c -o t -lrt -lpthread -lm
osc@ubuntu:~/final-src-osc10e$ ./t 3
hear
Producer2 produce 103 in buffer[0]
2.791299
hear
Producer1 produce 105 in buffer[1]
0.675286
hear
Producer0 produce 81 in buffer[2]
4.865270
hear
Producer1 produce 74 in buffer[3]
0.791000
hear
Producer1 produce 41 in buffer[4]
1.771934
hear
Producer1 produce -70 in buffer[5]
1.391488
hear
Producer2 produce -14 in buffer[6]
2.000095
hear
Producer1 produce -29 in buffer[7]
0.262578
hear
Producer1 produce 124 in buffer[8]
0.996796
hear
```


hear
Producer1 produce 84 in buffer[9]
1.497833
hear
Producer1 produce 27 in buffer[10]
4.245480
hear
Producer2 produce -25 in buffer[11]
0.653809
hear
Producer2 produce 118 in buffer[12]
2.741798
hear
Producer0 produce 46 in buffer[13]
6.654489
hear
Producer2 produce 51 in buffer[7]
4.566247
hear
Producer1 produce -55 in buffer[6]
0.526232
hear
Producer1 produce 102 in buffer[7]
3.651867
osc@ubuntu:~/final-src-osc10e$

消费者

```
osc@ubuntu:~/final-src-osc10e$ ./c 2
hear
Consumer2 comsum 46 in buffer[13]
hear
Consumer1 comsum 118 in buffer[12]
hear
Consumer2 comsum -25 in buffer[11]
hear
Consumer0 comsum 27 in buffer[10]
hear
Consumer2 comsum 84 in buffer[9]
hear
Consumer2 comsum 124 in buffer[8]
hear
Consumer0 comsum -29 in buffer[7]
hear
Consumer1 comsum 51 in buffer[7]
hear
Consumer1 comsum -14 in buffer[6]
hear
Consumer0 comsum 102 in buffer[7]
hear
Consumer2 comsum -55 in buffer[6]
hear
Consumer1 comsum -70 in buffer[5]
hear
Consumer1 comsum 41 in buffer[4]
hear
Consumer0 comsum 74 in buffer[3]
hear
```

```
hear
Consumer2 comsum 81 in buffer[2]
hear
Consumer2 comsum 105 in buffer[1]
hear
Consumer2 comsum 103 in buffer[0]
osc@ubuntu:~/final-src-osc10e$
```

从对应的数组中数据可以看出生产者和消费者都在正常运行

（具体解释详见代码）

3.Linux 内核实验

a.CFS 算法

CFS 即完全公平调度算法是针对普通进程的一种调度算法，在理想状态下，CFS 能实现将处理器的资源和运行时间尽可能得平分给每一个进程。

为了达到这个目的，CFS 抛弃了单纯以 nice 值给进程分配时间片的做法，对于低 nice 值得程序，它得到的时间片很小，每隔很小的一段时间就要进行一次进程切换，大大增加了 CPU 在把进程挪进挪出过程中的额外消耗。CFS 就是对不同的 nice 值也能实现分配上的相对公平的同时减少 CPU 的额外开销和进程的周转时间。

具体实现起来先是设定一个调度周期（ sched_latency_ns ），目标是让每个进程在这个周期内至少有机会运行一次，然后根据进程的数量，大家平分这个调度周期内的 CPU 使

用权，由于进程的 nice 值不同，分割调度周期的时候要加权；每个进程的累计运行时间保存在自己的 vruntime 字段里，CFS 根据 vruntime 的值，将所有的进程存入红黑树，每次在红黑树中选择 vruntime 最小的进程运行。在这种情况下，运行的时间都是按照权值来计算，而不是之前的绝对时间。这样就不会出现“时间片过小，放大进程切换所带来的消耗”的问题了。

问题一：优先级、nice 值和权重之间的关系

（1）优先级 PRI 决定了进程被调度的先后顺序，nice 值则表示进程可被执行的优先级的修正数值，PRI 值越小越快被执行，
加入 nice 值后，将会使得 PRI 变为：PRI(new)=PRI(old)+nice。由此看出，PR 是根据 NICE 排序的，规则是 NICE 越小 PR 越前（小，优先权更大），即其优先级会变高，则其越快被执行。实际上 nice 值为 CPU 提供了一个可以根据系统的资源以及具体进程的各类资源消耗情况，主动干预进程的优先级值，算是另一种意义上的优先级。

（2）权重则是基于 nice 值而确定的一个数，用于计算 vruntime。一个进程的权重越大，则说明这个进程更需要运行，因此它的虚拟运行时间就越小，这样被调度的机会就越大。

问题二：CFS 调度器中的 vruntime 的基本思想是什么？是如何计算的？何时得到更新？其中的 min_vruntime 有什么作用？

（1）vruntime 的基本思想就是通过计算进程的实际运行时间和进程的权重确定一个虚拟运行时间，CPU 根据这个时间确定应该调度的进程。

（2）vruntime = 实际运行时间 * NICE_0_LOAD / 进程权重

NICE_0_LOAD = 1024, 表示 nice 值为 0 的进程权重

可以看到，进程的权重越大，运行同样的时间，它的 vruntime 增长的越慢，需要运行的优先级就越高。

而一个进程在一个调度周期内的 vruntime 大小为：

vruntime = 进程在一个调度周期内的实际运行时间 * 1024 / 进程权重

= (调度周期 * 进程权重 / 所有进程总权重) * 1024 / 进程权重

= 调度周期 * 1024 / 所有进程总权重

可以看出在一个调度周期内虽然进程的权重不同，但是它们的 vruntime 增长速度应该是一样的 ，与权重无关。

在一个调度周期内，每个进程将系统分配给自己的运行时间使用完后，它们的 vruntime 的值是一样大的，因此一个进程的 vruntime 值越大，说明它得到的运行时间就越多，即每个周期内的进程的 vruntime 既有平均分配的部分，也有按权分配的部分，从而保证了 CFS 基于 vruntime 的机制既能公平选择进程，又能让高优先级进程获得较多的运行时间。

（3）当有新进程运行时，他的 vruntime 会初始化为 min_vruntime（最小虚拟时间），这也是保证公平的一种实现。如果这个值为 0 的话，则这个新进程会在相当长的一段时间内保存抢占 CPU 的优势，而老线程因为 vruntime 已经足够的多而运行时间变少，可能会饿晕，当不会饿死。这明显是不公平的。

以下是代码阅读部分，因为 Ubuntu 没有内核源码所以从 The Linux Kernel Archives 下了个来看，版本为 Linux4.3，所有的代码定位也是以 Linux4.3 版本做参照。因为想不到什么比较好的排版方式就采用了文字+注释的方式，结合原本就有的英语注释感觉会更好理解一点。加上由于 CFS 算法文件涉及到的代码分布比较散，所以看起来可能会有点跳，有些

代码可能也会漏。

1.首先关注被调度的进程实例，它的状态由 `struct sched_entity` 定义

<include\linux\sched,h>-line1246

```
1.  struct sched_entity {
2.      struct load_weight  load;       /* for load-balancing */ //加载时的优先级
3.      struct rb_node      run_node;//所处的红黑树节点
4.      struct list_head    group_node;//所在的进程分组
5.      unsigned int        on_rq;//
6.
7.      u64         exec_start; //进程开始时间
8.      u64         sum_exec_runtime;//进程在 CPU 上运行的总时间
9.      u64         vruntime;//进程的虚拟时间 ！！！！！！
10.     u64         prev_sum_exec_runtime;//上一次进程从 CPU 被挪出时在 CPU 上运行的总时间
11.
12.     u64         nr_migrations;
13.
14. #ifdef CONFIG_SCHEDSTATS
15.     struct sched_statistics statistics;
16. #endif
17.
18. #ifdef CONFIG_FAIR_GROUP_SCHED
19.     int         depth;
20.     struct sched_entity *parent;
21.     /* rq on which this entity is (to be) queued: */
22.     struct cfs_rq       *cfs_rq;//所在的 CFS 就绪队列
23.     /* rq "owned" by this entity/group: */
24.     struct cfs_rq       *my_q;
25. #endif
26.
27. #ifdef CONFIG_SMP
28.     /* Per entity load average tracking */
29.     struct sched_avg    avg;
30. #endif
31. };
```

其中 `u64 vruntime` 是 CFS 调度算法的关键变量，对于它的更新是通过函数 static void update_curr 实现的

<kernel\sched\fari.c>-line700

```
1.  static void update_curr(struct cfs_rq *cfs_rq)
2.  {
3.      struct sched_entity *curr = cfs_rq->curr;//就绪队列当前正在执行的进程
4.      u64 now = rq_clock_task(rq_of(cfs_rq));
5.      u64 delta_exec;
6.
```

```
7.      if (unlikely(!curr))//如果当前没有正在执行的进程
8.          return;//则不用更新啥事不做
9.      //如果当前有进程执行
10.     delta_exec = now - curr->exec_start;//计算当前进程本次被挪进 cpu 后已运行的时间
11.     if (unlikely((s64)delta_exec <= 0))
12.         return;
13.
14.     curr->exec_start = now;//更新进程的开始时间 ，以备下次使用
15.
16.     schedstat_set(curr->statistics.exec_max,
17.               max(delta_exec, curr->statistics.exec_max));
18.
19.     curr->sum_exec_runtime += delta_exec;//将本次运行的时间差加入进程实际运行的总时间
20.     schedstat_add(cfs_rq, exec_clock, delta_exec);
21.
22.     curr->vruntime += calc_delta_fair(delta_exec, curr);//调用 calc_delta_fair 函数计算 vruntime
23.     update_min_vruntime(cfs_rq);//更新 min_vruntime 的值
24.
25.     if (entity_is_task(curr)) {
26.         struct task_struct *curtask = task_of(curr);
27.
28.         trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
29.         cpuacct_charge(curtask, delta_exec);
30.         account_group_exec_runtime(curtask, delta_exec);
31.     }
32.
33.     account_cfs_rq_runtime(cfs_rq, delta_exec);
34. }
```

具体用来计算 vruntim 的函数 `calc_delta_fair` 定义如下

<kernel\sched\fari.c>-line596

```
1.  static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
2.  {
3.      if (unlikely(se->load.weight != NICE_0_LOAD))
4.          delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
5.
6.      return delta;
7.  }
```

转到函数 `__calc_delta`

<kernel\sched\fari.c>-line214

```
1.  static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct load_weight *lw)
2.  {
3.      u64 fact = scale_load_down(weight);
4.      int shift = WMULT_SHIFT;
5.
```

```
        __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);
    }
```

讲真这一段有点看不懂，所以没写注释，大概意思就是把传进来的`delta,NICE_0_LOAD,&se->load`三个参数分别对应实际运行时间、nice 值为零的权重以及自身权重代进公式中计算 vruntime。

接下来对 min_vruntime 的更新通过函数 `update_min_vruntime` 实现

<kernel\sched\fari.c>-line457

```
    static void update_min_vruntime(struct cfs_rq *cfs_rq)
    {
        u64 vruntime = cfs_rq->min_vruntime;//给当前初始化一个 vruntim

        if (cfs_rq->curr)//如果当前 CFS 队列上有进程
            vruntime = cfs_rq->curr->vruntime;//则当前被选择的 vruntime 即为该进程的 vruntime

        if (cfs_rq->rb_leftmost) {//如果 CFS 队列的红黑树存在最左节点即存在等待被调度的进程
            struct sched_entity *se = rb_entry(cfs_rq->rb_leftmost,
                                struct sched_entity,
                                run_node);//登记最左节点进程信息

            if (!cfs_rq->curr)//如果当前 CFS 队列上没有正在运行的进程
                vruntime = se->vruntime;//则 vruntime 为左节点进程的 vruntime
            else////如果当前 CFS 队列上有进程
                 vruntime = min_vruntime(vruntime, se->vruntime);//则在当前进程和左节点进程中挑选较小的 vruntime
        }
```

```
    /* ensure we never gain time by being placed backwards. */
    /*保证 min_vruntime 单调不减(min_vruntime 下降的话会导致当前进程被重复调用执行),
    所以只有在当前的 vruntime 超出的 cfs_rq->min_vruntime 的时候才更新*/
    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
}
```

1. 前面函数中涉及到了 **CFS** 队列实例 `cfs_rq`
   <kernel\sched\sched.h>-line343

```
struct cfs_rq {
    struct load_weight load;//CFS 队列总进程权重
    unsigned int nr_running, h_nr_running;//队列中进程的个数

    u64 exec_clock;//运行的时钟
    u64 min_vruntime;//CFS 队列的 vruntime 值， 即上一次被选中的队列中 vruntime 最小值
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif

    struct rb_root tasks_timeline;//红黑树的根节点
    struct rb_node *rb_leftmost;//红黑树的最左节点，即 vruntime 值最小的节点

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    /*当前运行的进程、下一个要调度的进程、最后一个要调度的进程、即将抢占的进程 */
    struct sched_entity *curr, *next, *last, *skip;

#ifdef  CONFIG_SCHED_DEBUG
    unsigned int nr_spread_over;
#endif

#ifdef CONFIG_SMP
    /*
     * CFS load tracking
     */
    struct sched_avg avg;
    u64 runnable_load_sum;
    unsigned long runnable_load_avg;
#ifdef CONFIG_FAIR_GROUP_SCHED
```

```
33.     unsigned long tg_load_avg_contrib;
34. #endif
35.     atomic_long_t removed_load_avg, removed_util_avg;
36. #ifndef CONFIG_64BIT
37.     u64 load_last_update_time_copy;
38. #endif
39. 
40. #ifdef CONFIG_FAIR_GROUP_SCHED
41.     /*
42.      *   h_load = weight * f(tg)
43.      *
44.      * Where f(tg) is the recursive weight fraction assigned to
45.      * this group.
46.      */
47.     unsigned long h_load;
48.     u64 last_h_load_update;
49.     struct sched_entity *h_load_next;
50. #endif /* CONFIG_FAIR_GROUP_SCHED */
51. #endif /* CONFIG_SMP */
52. 
53. #ifdef CONFIG_FAIR_GROUP_SCHED
54.     struct rq *rq;  /* cpu runqueue to which this cfs_rq is attached 承载了本 CFS 队列的 CPU 队列*/
55. 
56.     /*
57.      * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
58.      * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
59.      * (like users, containers etc.)
60.      *
61.      * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a cpu. This
62.      * list is used during load balance.
63.      */
64.     int on_list;
65.     struct list_head leaf_cfs_rq_list;
66.     struct task_group *tg;  /* group that "owns" this runqueue */
67. 
68. #ifdef CONFIG_CFS_BANDWIDTH
69.     int runtime_enabled;
70.     u64 runtime_expires;
71.     s64 runtime_remaining;
72. 
73.     u64 throttled_clock, throttled_clock_task;
74.     u64 throttled_clock_task_time;
75.     int throttled, throttle_count;
76.     struct list_head throttled_list;
```

```
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

2. 对该队列插入新进程由函数 `enqueue_task_fair` 实现

<kernel\sched\fari.c>-line4079

```
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;

    for_each_sched_entity(se) {//循环插入节点
        if (se->on_rq)//判断当前节点是否已在队列上
            break;//如果在的话就不用重新插入一遍
        cfs_rq = cfs_rq_of(se);//得到当前 CFS 运行队列
        enqueue_entity(cfs_rq, se, flags);//往当前队列所维护的红黑树中插入当前节点

        /*
         * end evaluation on encountering a throttled cfs_rq
         *
         * note: in the case of encountering a throttled cfs_rq we will
         * post the final h_nr_running increment below.
        */
        if (cfs_rq_throttled(cfs_rq))
            break;
        cfs_rq->h_nr_running++;

        flags = ENQUEUE_WAKEUP;
    }

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        cfs_rq->h_nr_running++;

        if (cfs_rq_throttled(cfs_rq))
            break;

        update_load_avg(se, 1);
        update_cfs_shares(cfs_rq);
    }

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

```

```
    hrtick_update(rq);
}
```

主要操作由函数 `enqueue_entity` 实现

<kernel\sched\fari.c>-line2943

```
static void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /*
     * Update the normalized vruntime before updating min_vruntime
     * through calling update_curr().
     */
    if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
        se->vruntime += cfs_rq->min_vruntime;

    /*
     * Update run-time statistics of the 'current'.
     更新当前进程的统计信息
     */
    update_curr(cfs_rq);
    enqueue_entity_load_avg(cfs_rq, se);
    account_entity_enqueue(cfs_rq, se);
    update_cfs_shares(cfs_rq);

    if (flags & ENQUEUE_WAKEUP) {//如果当前进程是从休眠中被唤醒的
        place_entity(cfs_rq, se, 0);// 调用函数处理其 vruntime
        enqueue_sleeper(cfs_rq, se);
    }

    update_stats_enqueue(cfs_rq, se);
    check_spread(cfs_rq, se);
    if (se != cfs_rq->curr)//如果当前节点不是正在运行的进程，则调用插入的实际操作
        __enqueue_entity(cfs_rq, se);  //具体实现插入操作的函数
    se->on_rq = 1;

    if (cfs_rq->nr_running == 1) {
        list_add_leaf_cfs_rq(cfs_rq);
        check_enqueue_throttle(cfs_rq);//抢占？？？
    }
}
```

这之中涉及到两个函数，一个是处理被唤醒函数 vruntime 值的 `place_entity` 该函数存在的意义主要是防止休眠进程的 vruntime 在休眠时保持不变，而导致它在被唤醒时拥有比其他运行过的进程小得多的 vruntime 从而疯狂占用 CPU 让其他进程饿死

<kernel\sched\fari.c>-line2908

```
static void place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
```

```
        u64 vruntime = cfs_rq->min_vruntime;
        /*
         * The 'current' period is already promised to the current tasks,
         * however the extra weight of the new task will slow them down a
         * little, place the new task so that it fits in the slot that
         * stays open at the end.
         */
        if (initial && sched_feat(START_DEBIT))
            vruntime += sched_vslice(cfs_rq, se);
        /* sleeps up to a single latency don't count. */
        if (!initial) {
            unsigned long thresh = sysctl_sched_latency;
            /*
             * Halve their sleep time's effect, to allow
             * for a gentler effect of sleepers:
             */
            if (sched_feat(GENTLE_FAIR_SLEEPERS))
                thresh >>= 1;

            vruntime -= thresh;
        }
        /* ensure we never gain time by being placed backwards. */
        se->vruntime = max_vruntime(se->vruntime, vruntime);
    }
```

`enqueue_entity`中另一个主要函数就是承担了红黑树插入操作具体实现的 `__enqueue_entity`

<kernel\sched\fari.c>-line488

```
    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;
        int leftmost = 1;
        /*
         * Find the right place in the rbtree:
         *在红黑数中为当前进程找到正确的节点位置并插入‘
         *具体算法涉及到红黑树结构啥的我注释写不清楚好像也不是作业重点就省略注释了
         */
        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.
             */
```

```
            if (entity_before(se, 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);
    }
```

4.把进程从 CFS 队列中移出的操作由函数 `dequeue_task_fair` 实现

<kernel\sched\fari.c>-line4127

（由于代码逻辑跟函数的嵌套和插入进程操作差不多就不贴完整代码占位置了）

```
    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;
        int task_sleep = flags & DEQUEUE_SLEEP;

        for_each_sched_entity(se) {
            cfs_rq = cfs_rq_of(se);
            dequeue_entity(cfs_rq, se, flags);
    ......
    }
```

主要操作由 `dequeue_entit` 完成

<kernel\sched\fari.c>-line4127

```
    static void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
    {
        /*
         * Update run-time statistics of the 'current'.
         */
        update_curr(cfs_rq);
        dequeue_entity_load_avg(cfs_rq, se);

        update_stats_dequeue(cfs_rq, se);
        clear_buddies(cfs_rq, se);
```

```
    ............
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);  //具体实现移出操作的函数
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);

    /*
     * Normalize the entity after updating the min_vruntime because the
     * update can refer to the ->curr item and we need to reflect this
     * movement in our normalized position.
     */
    if (!(flags & DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->min_vruntime;

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

    update_min_vruntime(cfs_rq);  //更新 min_vruntime
    update_cfs_shares(cfs_rq);
}
```

具体实现在 __dequeue_entity

<kernel\sched\fari.c>-line526

```
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);
}
```

b.内核系统调用

首先先写一个内核模块测试一下代码：

（因为只是看一下代码就直接用 Mobaxterm 自带的编辑器看了，方便一点）

```
{
        printk("se.exec_start\t\t:%16llu\n",current->se.exec_start);
    printk("se.vruntime\t\t:%16llu\n",current->se.vruntime);
    printk("se.sum_exec_runtime\t\t:%16llu\n",current->se.sum_exec_runtime);
    printk("se.load.weight\t\t:%16lu\n",current->se.load.weight);
    printk("se.avg.load_sum\t\t:%16llu\n",current->se.avg.load_sum);
    printk("se.avg.util_sum\t\t:%16u\n",current->se.avg.util_sum);
    printk("se.avg.load_avg\t\t:%16lu\n",current->se.avg.load_avg);
    printk("se.avg.util_avg\t\t:%16llu\n",current->se.avg.util_avg);
    printk("se.avg.last_update_time\t\t:%16llu\n",current->se.avg.last_update_time);
    printk("se.nr_migrations\t\t:%16llu\n",current->se.nr_migrations);
    printk("nr_switches\t\t:%16lu\n",current->nvcsw+current->nivcsw);
    printk("nr_voluntary_switches\t\t:%16lu\n",current->nvcsw);
    printk("nr_involuntary_switches\t\t:%16lu\n",current->nivcsw);
    return 0;
}
```

加载到内核中后 dmesgm 查看有：

代码没问题下一步写系统调用。但还是因为 Ubuntu 没有内核代码，添加系统调用需要修改内核代码只能重新装一个，上一题用的 4.3 版本有点太低了，重新下了 4.9 的。

（1）添加系统调用函数

<kernel\sys.c>

```
asmlinkage long sys_listse(void){
    printk("se.exec_start\t\t:%16llu\n",current->se.exec_start);
     printk("se.vruntime\t\t:%16llu\n",current->se.vruntime);
     printk("se.sum_exec_runtime\t\t:%16llu\n",current->se.sum_exec_runtime);
     printk("se.load.weight\t\t:%16lu\n",current->se.load.weight);
     printk("se.avg.load_sum\t\t:%16llu\n",current->se.avg.load_sum);
     printk("se.avg.util_sum\t\t:%16u\n",current->se.avg.util_sum);
     printk("se.avg.load_avg\t\t:%16lu\n",current->se.avg.load_avg);
     printk("se.avg.util_avg\t\t:%16llu\n",current->se.avg.util_avg);
     printk("se.avg.last_update_time\t\t:%16llu\n",current->se.avg.last_update_time);
     printk("se.nr_migrations\t\t:%16llu\n",current->se.nr_migrations);
     printk("nr_switches\t\t:%16lu\n",current->nvcsw+current->nivcsw);
     printk("nr_voluntary_switches\t\t:%16lu\n",current->nvcsw);
     printk("nr_involuntary_switches\t\t:%16lu\n",current->nivcsw);
     return 0;
}
#endif /* CONFIG_COMPAT */
```

（2）添加声明

<arch\x86\include\asm\syscalls.h>

```
/* Common in X86_32 and X86_64 */
/* kernel/ioport.c */
asmlinkage long sys_ioperm(unsigned long, unsigned long, int);
asmlinkage long sys_iopl(unsigned int);
asmlinkage long sys_listse(void);

```

（3）添加系统调动 ID

<arch\x86\entry\syscalls\syscall_64.tbl>

```
331 common  pkey_free   sys_pkey_free
333     64      listse                  sys_listse

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512 x32 rt_sigaction    compat_sys_rt_sigaction
```

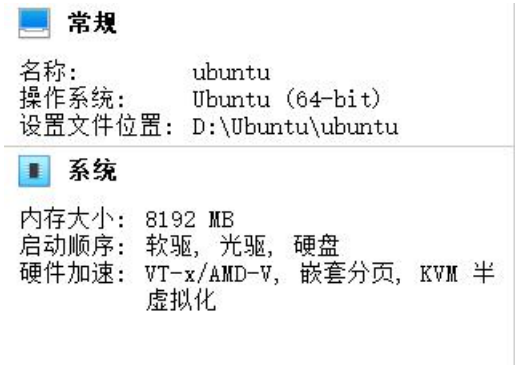
（4）编译并安装新内核

```
  CC      kernel/smpboot.o
  CC      kernel/ucount.o
  CC      mm/shmem.o
  CC      arch/x86/events/msr.o
  CC      kernel/groups.o
  CC      kernel/bpf/core.o
```

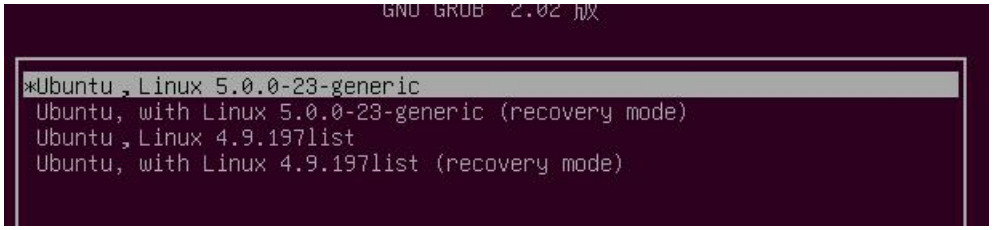
然后就报错了：

```
  LD      drivers/net/built-in.o
  LD      drivers/built-in.o
  LD      vmlinux.o
ld: final link failed: Memory exhausted
Makefile:1004: recipe for target 'vmlinux' failed
make: *** [vmlinux] Error 1
```

这个问题困扰了我挺久的，因为它报错的是内存溢出，网上相似错误给的资料比较少。磁盘修改大小容易内存修改还挺难做的，只能说是虚拟机性能不行支撑不了一个新内核。纠结了半天最后还是选择新装一个性能好一点的虚拟机重新来。新虚拟机直接采用了 Ubuntu 官网给的最新版本，并配置一个大一点的内存：



上述操作全部重新来一遍之后成功编译安装，可以看到系统里面有两个内核：

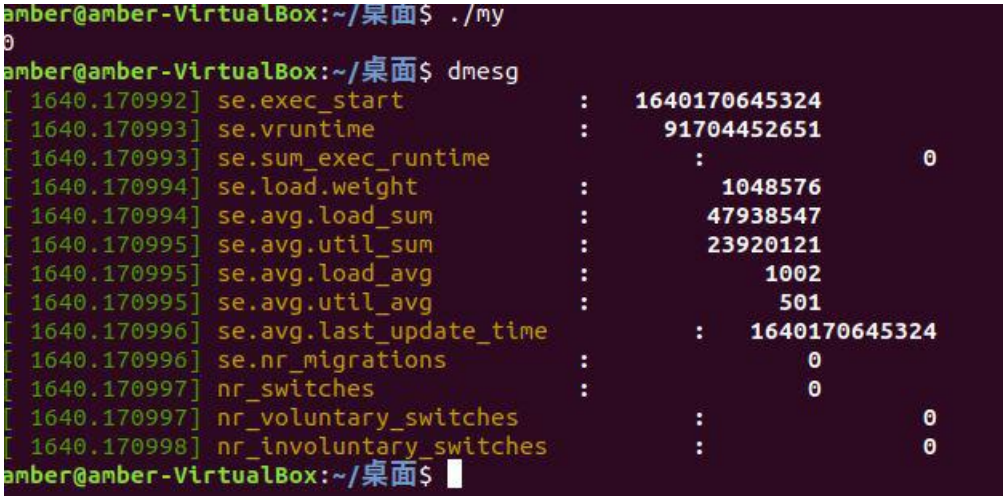


打开新内核打印系统信息：

确认无误之后编译并运行 mycall.c

（具体详见代码）

打印结果：



搞定。