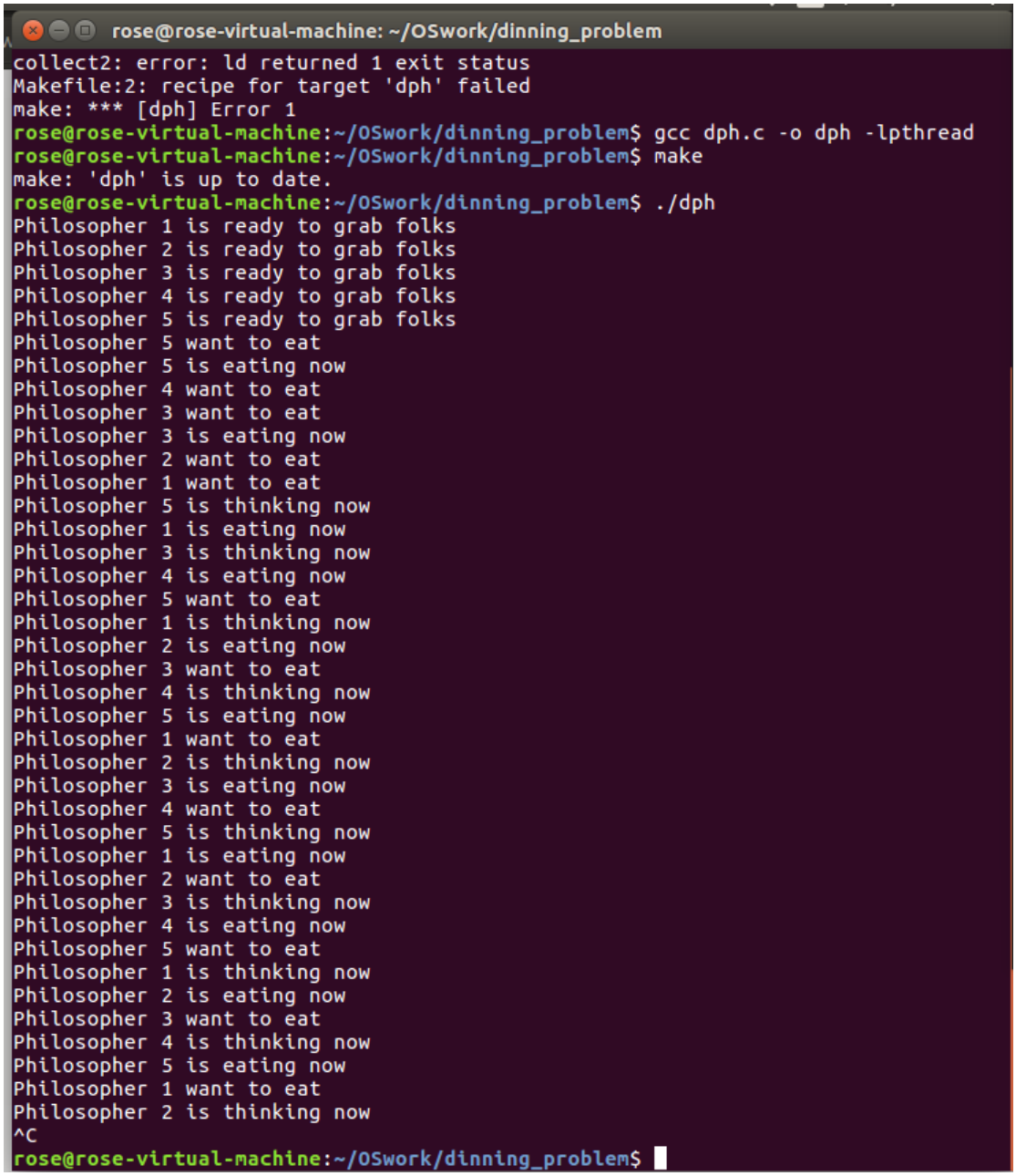
第一题：哲学家问题运行结果展示



第二题：生产者消费者问题代码运行结果展示

```
rose@rose-virtual-machine:~/OSwork/pro_cons_problem$ ./prod 7
-46.000000put 3 ok
process id: 5497  thread id: 139989640886016
put 6 ok
process id: 5497  thread id: 139989649278720
put 7 ok
process id: 5497  thread id: 139989657671424
put 5 ok
process id: 5497  thread id: 139989640886016
put 3 ok
process id: 5497  thread id: 139989649278720
put 5 ok
process id: 5497  thread id: 139989657671424
put 6 ok
process id: 5497  thread id: 139989640886016
put 2 ok
process id: 5497  thread id: 139989649278720
put 9 ok
process id: 5497  thread id: 139989657671424
put 1 ok
process id: 5497  thread id: 139989640886016
put 2 ok
process id: 5497  thread id: 139989649278720
put 7 ok
process id: 5497  thread id: 139989657671424
put 0 ok
process id: 5497  thread id: 139989640886016
put 9 ok
process id: 5497  thread id: 139989649278720
put 3 ok
process id: 5497  thread id: 139989657671424
put 6 ok
process id: 5497  thread id: 139989640886016
put 0 ok
process id: 5497  thread id: 139989649278720
put 6 ok
process id: 5497  thread id: 139989657671424
```

1

```
rose@rose-virtual-machine:~/OSwork/pro_cons_problem$ ./cons 5
process id: 5493  thread id: 155469568
process id: 5493  thread id: 163862272
process id: 5493  thread id: 172254976
process id: 5493  thread id: 155469568
process id: 5493  thread id: 163862272
process id: 5493  thread id: 172254976
process id: 5493  thread id: 155469568
process id: 5493  thread id: 163862272
process id: 5493  thread id: 172254976
process id: 5493  thread id: 155469568
process id: 5493  thread id: 163862272
process id: 5493  thread id: 172254976
process id: 5493  thread id: 163862272
process id: 5493  thread id: 155469568
process id: 5493  thread id: 172254976
process id: 5493  thread id: 163862272
process id: 5493  thread id: 155469568
process id: 5493  thread id: 172254976
```

2

图 1 为生产者，2 为消费者。由结果可看出有两个进程（./prod、./cons），运行时输入参数 lambda 每个进程中有三个线程，即分别有三个消费者和三个生产者。共产生 20 个数据。

第三题：

a) 以 CFS 的主要文件 Fair.c 为起点，浏览相关联的文件。理解 Linux 进程的基本结构、状态设置，CPU 的调度基本架构，理解 CFS 调度算法的基本流程和主要数据结构。摘取关键代码片段，用自己的的方式描述出来。不要求理解每一条语句，但需要陈述主要脉络。此外，单独回答以下问题：

说明：
主要阅读的代码是 fair.c 和 shed.h。使用的内核函数版本是 linux-5.3.8。

一、Linux 进程的基本结构：

linux 把进程区分为**实时进程**和**非实时进程**，其中非实时进程进一步划分为**交互式进程**和**批处理进程**，调度算法可以明确的确认所有实时进程的身份。对于不同的进程采用不同的调度策略，以下举例其一：

1. **完全公平运行队列(CFS)策略：** 描述运行在同一个 cpu 上的处于 TASK_RUNNING 状态的普通进程的各种运行信息：

```
482  struct cfs_rq {
483      struct load_weight  load;//运行队列总的进程权重
484      unsigned long       runnable_weight;
485      unsigned int        nr_running;
486      unsigned int        h_nr_running;//进程的个数
487      u64         exec_clock;//运行的时钟
488      u64         min_vruntime;//红黑树中最小的vruntime
489  #ifndef CONFIG_64BIT
490      u64         min_vruntime_copy;
491  #endif
492      struct rb_root_cached   tasks_timeline;//红黑树的根节点
493      struct sched_entity *curr;
494      struct sched_entity *next;
495      struct sched_entity *last;
496      struct sched_entity *skip;//当前运行的进程、下一个要调度的进程、马上要抢占的进程
497  #ifdef  CONFIG_SCHED_DEBUG
498      unsigned int        nr_spread_over;
499  #endif
500  #ifdef CONFIG_SMP
501      struct sched_avg    avg;
502      struct rq       *rq;    /* CPU runqueue to which this cfs_rq is attached */
503      int         on_list;
504      struct list_head    leaf_cfs_rq_list;
```

基本流程：在系统中至少有一个 CFS 运行队列，其就是根 CFS 运行队列，而其他的进程组和进程都包含在此运行队列中，不同的是进程组又有它自己的 CFS 运行队列，其运行队列中包含的是此进程组中的所有进程。当调度器从根 CFS 运行队列中选择了一个进程组进行调度时，进程组会从自己的 CFS 运行队列中选择一个调度实体进行调度(这个调度实体可能为进程，也可能又是一个子进程组)，就这样一直深入，直到最后选出一个进程进行运行为止。

二、**调度实体：**（CFS 的主要数据结构）

调度实体用于记录一个进程的运行状态信息，调度器不限于调度进程，还可以调度更大的实体，比如实现组调度：可用的 CPUI 时间首先在一半的进程组(比如，所有进程按照所有者分组)之间分配，接下来分配的时间再在组内进行二次分配。

这种一般性要求调度器不直接操作进程，而是处理可调度实体，因此需要一个通用的数据结构描述这个调度实体,即 seched_entity 结构，其实际上就代表了一个调度对象，可以为一个进程，也可以为一个进程组。

调度实体的类型为 seched_entity,linux 据此定义了 sched_dl_entity，sched_rt_entity，sched_dl_entity 三个调度实体。

```
567  struct sched_entity {
568      struct load_weight  load; //进程的权重
569      struct rb_node      run_node; //运行队列中的红黑树结点
570      struct list_head    group_node; //与组调度有关
571      unsigned int        on_rq; //进程现在是否处于TASK_RUNNING状态
572
573      u64         exec_start; //一个调度tick的开始时间
574      u64         sum_exec_runtime; //进程从出生开始，已经运行的实际时间
575      u64         vruntime; //虚拟运行时间
576      u64         prev_sum_exec_runtime; //本次调度之前，进程已经运行的实际时间
577      struct sched_entity *parent; //组调度中的父进程
578      struct cfs_rq       *cfs_rq; //进程此时在哪个运行队列中
579  };
```

三、Linux 进程调度的主要过程:

1. 进程的创建

创建新进程时，需要设置新进程的 vruntime 值(虚拟运行时间，调度的关键)以及将新进程加入红黑树中．并判断是否需要抢占当前进程，主要代码如下:

a. 创建新进程并且设置 vruntime 值

Vruntime: 一次调度间隔的虚拟运行时间 = 实际运行时间 *(NICE_0_LOAD / 权重);

```
6578  static void task_fork_fair(struct task_struct *p)
6579  {
6580      struct cfs_rq *cfs_rq;
6581      struct sched_entity *se = &p->se, *curr;
6582      int this_cpu = smp_processor_id();
6583      struct rq *rq = this_rq();
6584      unsigned long flags;
6585
6586      raw_spin_lock_irqsave(&rq->lock, flags);
6587
6588      update_rq_clock(rq);
6589
6590      cfs_rq = task_cfs_rq(current);
6591      curr = cfs_rq->curr;
6592
6593      rcu_read_lock();
6594      __set_task_cpu(p, this_cpu); //设置新进程在哪个cpu上运行
6595      rcu_read_unlock();
6596
6597      update_curr(cfs_rq); //更新当前进程的vruntime值
6598
6599      if (curr)
6600          se->vruntime = curr->vruntime; //先以父进程的vruntime为基础
6601      place_entity(cfs_rq, se, 1); //设置新进程的vruntime值，1表示是新进程
6602
6603      if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) { //sysctl_sched_child_runs_first值表示是否设置了让子进程
6604
6605          swap(curr->vruntime, se->vruntime); //当子进程的vruntime值大于父进程的vruntime时，交换两个进程的vruntime值
6606          resched_task(rq->curr); //设置重新调度标志TIF_NEED_RESCHED
6607      }
6608
6609      se->vruntime -= cfs_rq->min_vruntime; //防止新进程运行时是在其他cpu上运行的，这样在加入另一个cfs_rq时再加上另一个cfs_rq队列的min_vr
6610
6611      raw_spin_unlock_irqrestore(&rq->lock, flags);
6612  }
```

以上函数 task_fork_fair 调用 place_entity 函数来设置新进程的 vruntime 值:

```
3865  static void
3866  place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
3867  {
3868      u64 vruntime = cfs_rq->min_vruntime;//以当前运行队列的min_vruntime为基础
3869      if (initial && sched_feat(START_DEBIT))
3870          vruntime += sched_vslice(cfs_rq, se);//sched_vslice函数计算一个调度周期内新进程的vruntime大小
3871  //START_DEBIT表示新进程在一个调度周期内是否已经运行过了，如果是则需要加上该进程一个调度周期内的vruntime值大小
3872      /* sleeps up to a single latency don't count. */
3873      if (!initial) {//睡眠时传入的initial值为0
3874          unsigned long thresh = sysctl_sched_latency;//一个调度周期时间值
3875          if (sched_feat(GENTLE_FAIR_SLEEPERS))
3876              thresh >>= 1;
3877          vruntime -= thresh;//对睡眠进程的vruntime补偿
3878      }
3879      /* ensure we never gain time by being placed backwards. */
3880      se->vruntime = max_vruntime(se->vruntime, vruntime);//防止短期睡眠的进程vruntime值获得补偿
3881  }
```

b. 设置完新进程的 vruntime 之后，判断新进程是否可以抢占当前进程

```
6624  static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
6625  {
6626      struct task_struct *curr = rq->curr;
6627      struct sched_entity *se = &curr->se, *pse = &p->se;//se是当前进程。pse是新进程
6628      struct cfs_rq *cfs_rq = task_cfs_rq(curr);
6629      int scale = cfs_rq->nr_running >= sched_nr_latency;
6630      int next_buddy_marked = 0;
6631      if (unlikely(se == pse))
6632          return;
6633      if (unlikely(throttled_hierarchy(cfs_rq_of(pse))))
6634          return;
6635      if (sched_feat(NEXT_BUDDY) && scale && !(wake_flags & WF_FORK)) {
6636          set_next_buddy(pse);
6637          next_buddy_marked = 1;  }
6638      if (test_tsk_need_resched(curr))
6639          return;
6640      if (unlikely(task_has_idle_policy(curr)) &&
6641          likely(!task_has_idle_policy(p)))
6642          goto preempt;
6643      if (unlikely(p->policy != SCHED_NORMAL) || !sched_feat(WAKEUP_PREEMPTION))
6644          return;
6645      find_matching_se(&se, &pse);
6646      update_curr(cfs_rq_of(se));
6647      BUG_ON(!pse);
6648      if (wakeup_preempt_entity(se, pse) == 1) {//判断新进程是否可以抢占当前进程
6649          if (!next_buddy_marked)
6650              set_next_buddy(pse);
6651          goto preempt;
6652      }
6653      return;
6654  preempt:
6655      resched_curr(rq);
6656      if (unlikely(!se->on_rq || curr == rq->idle))
6657          return;
6658      if (sched_feat(LAST_BUDDY) && scale && entity_is_task(se))
6659          set_last_buddy(se);
6660  }
```

其中调用的 weakup_preempt_entity 函数判断新进程是否可以抢占当前进程

```
6576  static int
6577  wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
6578  {
6579      s64 gran, vdiff = curr->vruntime - se->vruntime;
6580      if (vdiff <= 0)//新进程的vruntime值比当前进程大，不发生抢占
6581          return -1;
6582      gran = wakeup_gran(se);//判断发生抢占时候的调度粒度
6583      if (vdiff > gran)//两个进程之间的差值大于调度粒度的时候发生抢占
6584          return 1;
6585      return 0;//小于调度粒度则不发生抢占
6586      //调度粒度相当于一个阈值，如果没有设置调度粒度而两个进程之间的差值比较小的话，
6587      //系统会在这两个进程之间进行频繁的调度工作，耗费了大量的时间和资源。
6588  }
```

2. 进程的唤醒

唤醒进程时，需要调整睡眠进程的 vruntime 值，并且将睡眠进程加入红黑树中．并判断是否需要抢占当前进程

```
3955  static void
3956  enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
3957  {
3958      bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
3959      bool curr = cfs_rq->curr == se;
3960      /*   * If we're the current task, we must renormalise before calling
3961       * update_curr().    */
3962      if (renorm && curr)
3963          se->vruntime += cfs_rq->min_vruntime;
3964      update_curr(cfs_rq);//更新当前的进程时间值
3965      /*   * Otherwise, renormalise after, such that we're placed at the current
3966       * moment in time, instead of some random moment in the past. Being
3967       * placed in the past could significantly boost this task to the
3968       * fairness detriment of existing tasks.     */
3969      if (renorm && !curr)
3970          se->vruntime += cfs_rq->min_vruntime;
3971      update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
3972      update_cfs_group(se);
3973      enqueue_runnable_load_avg(cfs_rq, se);
3974      account_entity_enqueue(cfs_rq, se);
3975      if (flags & ENQUEUE_WAKEUP)
3976          place_entity(cfs_rq, se, 0);
3977      check_schedstat_required();
3978      update_stats_enqueue(cfs_rq, se, flags);
3979      check_spread(cfs_rq, se);
3980      if (!curr)
3981          __enqueue_entity(cfs_rq, se);//将进程加入到红黑树中
3982      se->on_rq = 1;
3983      if (cfs_rq->nr_running == 1) {
3984          list_add_leaf_cfs_rq(cfs_rq);
3985          check_enqueue_throttle(cfs_rq);
3986      }
3987  }
```

以上函数调用_enqueue_entity（）

```
565  static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
566  {
567      struct rb_node **link = &cfs_rq->tasks_timeline.rb_root.rb_node;
568      struct rb_node *parent = NULL;
569      struct sched_entity *entry;
570      bool leftmost = true;
571      while (*link) {
572          parent = *link;
573          entry = rb_entry(parent, struct sched_entity, run_node);
574          /*
575           * We dont care about collisions. Nodes with
576           * the same key stay together.
577           */
578          if (entity_before(se, entry)) {
579              link = &parent->rb_left;
580          } else {
581              link = &parent->rb_right;
582              leftmost = false;
583          }
584      }
585      rb_link_node(&se->run_node, parent, link);
586      rb_insert_color_cached(&se->run_node,
587                             &cfs_rq->tasks_timeline, leftmost);
588  }
```

判断是否需要抢占当前进程的代码和 a 中的部分相同

3. 进程的调度

进程调度时，需要把当前进程加入红黑树中，还要从红黑树中挑选出下一个要运行的进程.

a. 把当前进程加入红黑树中

```
4192  static void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
4193  {
4194      /*   * If still on the runqueue then deactivate_task()
4195       * was not called and update_curr() has to be done:  */
4196      if (prev->on_rq)
4197          update_curr(cfs_rq);
4198      /* throttle cfs_rqs exceeding runtime */
4199      check_cfs_rq_runtime(cfs_rq);
4200      check_spread(cfs_rq, prev);
4201      if (prev->on_rq) {
4202          //把还处于运行状态的进程加入红黑树中
4203          update_stats_wait_start(cfs_rq, prev);
4204          /* Put 'current' back into the tree. */
4205          __enqueue_entity(cfs_rq, prev);//加入红黑树
4206          /* in !on_rq case, update occurred at dequeue */
4207          update_load_avg(cfs_rq, prev, 0);
4208      }
4209      //没有当前进程了，这个当前进程将在pick_next_task中更新
4210      cfs_rq->curr = NULL;
```

b. 选出下一个要运行的进程

```
6723  static struct task_struct *
6724  pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
6725  {
6726      struct task_struct *p;
6727      struct cfs_rq *cfs_rq = &rq->cfs;
6728      struct sched_entity *se;
6729      if (unlikely(!cfs_rq->nr_running))
6730          return NULL;
6731      do {
6732          se = pick_next_entity(cfs_rq); // 选出下一个要运行的进程
6733          set_next_entity(cfs_rq, se);  //将选出的进程设置为当前进程
6734          cfs_rq = group_cfs_rq(se);
6735      } while (cfs_rq);
6736      p = task_of(se);
6737      hrtick_start_fair(rq, p);
6738      return p;
6739  }
```

上面的函数调用 pick_next_entity 和 set_next_entity 来完成选择下一个运行的进程的操作:

```
4154  static void
4155  set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
4156  {   /* 'current' is not kept within the tree. */
4157      if (se->on_rq) {
4158          /*
4159           * Any task has to be enqueued before it get to execute on
4160           * a CPU. So account for the time it spent waiting on the
4161           * runqueue.
4162           */
4163          update_stats_wait_end(cfs_rq, se);//把结点从红黑树上取下来
4164          __dequeue_entity(cfs_rq, se);//把新选择出来的进程移除红黑树
4165          update_load_avg(cfs_rq, se, UPDATE_TG);
4166      }
4167      update_stats_curr_start(cfs_rq, se);
4168      cfs_rq->curr = se;//设置为当前进程
4169      /*
4170       * Track our maximum slice length, if the CPU's load is at
4171       * least twice that of our own weight (i.e. dont track it
4172       * when there are only lesser-weight tasks around):
4173       */
4174      if (schedstat_enabled() &&
4175          rq_of(cfs_rq)->cfs.load.weight >= 2*se->load.weight) {
4176          schedstat_set(se->statistics.slice_max,
4177              max((u64)schedstat_val(se->statistics.slice_max),
4178                  se->sum_exec_runtime - se->prev_sum_exec_runtime));
4179      }
4180      se->prev_sum_exec_runtime = se->sum_exec_runtime;//记录本次调度之前已经运行的时间
4181  }
```

```
4193  static struct sched_entity *
4194  pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
4195  {
4196      struct sched_entity *left = __pick_first_entity(cfs_rq);
4197      struct sched_entity *se;
4198      /** If curr is set we have to see if its left of the leftmost entity
4199       * still in the tree, provided there was anything in the tree at all.*/
4200      if (!left || (curr && entity_before(curr, left)))
4201          left = curr;
4202      se = left; /* ideally we run the leftmost entity */
4203      /** Avoid running the skip buddy, if running something else can
4204       * be done without getting too unfair.*/
4205      if (cfs_rq->skip == se) {
4206          struct sched_entity *second;
4207          if (se == curr) {
4208              second = __pick_first_entity(cfs_rq);
4209          } else {
4210              second = __pick_next_entity(se);
4211              if (!second || (curr && entity_before(curr, second)))
4212                  second = curr;
4213          }
4214          if (second && wakeup_preempt_entity(second, left) < 1)
4215              se = second;
4216      }
4217      /* * Prefer last buddy, try to return the CPU to a preempted task.
4218       */
4219      if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
4220          se = cfs_rq->last;
4221      /** Someone really wants this to run. If it's not unfair, run it.*/
4222      if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
4223          se = cfs_rq->next;
4224      clear_buddies(cfs_rq, se);
4225      return se;//如果选出的进程的vruntime值比next和last指向的进程的vruntime值小到粒度之外，则返回新选出的进程
4226  }
```

4. 时钟周期中断

该部分的函数主要是更新当前进程的 vruntime 值和实际运行的时间，并且判断当前进程在本次调度中实际的运行时间是否超过了调度周期分配的实际运行时间。如果超过则设置重新调度标志 TIF_NEED_RESCHED 。

```
4254  static void
4255  entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
4256  {
4257      /*    * Update run-time statistics of the 'current'.*/
4258      update_curr(cfs_rq);//更新当前进程的时间值
4259      /** Ensure that runnable average is periodically updated.*/
4260      update_load_avg(cfs_rq, curr, UPDATE_TG);
4261      update_cfs_group(curr);
4262  #ifdef CONFIG_SCHED_HRTICK
4263      /*
4264       * queued ticks are scheduled to match the slice, so don't bother
4265       * validating it and just reschedule.
4266       */
4267      if (queued) {
4268          resched_curr(rq_of(cfs_rq));
4269          return;
4270      }
4271      /** don't let the period tick interfere with the hrtick preemption */
4272      if (!sched_feat(DOUBLE_TICK) &&
4273              hrtimer_active(&rq_of(cfs_rq)->hrtick_timer))
4274          return;
4275  #endif
4276      if (cfs_rq->nr_running > 1)
4277          check_preempt_tick(cfs_rq, curr);//判断是否需要设置重新调度标志
4278  }
```

a. 上面的函数 entity_tick 调用 update_curr 函数更新当前进程的 vruntime 和实际运行时间。

```
834  static void update_curr(struct cfs_rq *cfs_rq)
835  {
836      struct sched_entity *curr = cfs_rq->curr;
837      u64 now = rq_clock_task(rq_of(cfs_rq));
838      u64 delta_exec;
839      if (unlikely(!curr))
840          return;
841      delta_exec = now - curr->exec_start;//得到本次tick实际运行的时间值
842      if (unlikely((s64)delta_exec <= 0))
843          return;
844      curr->exec_start = now;//设置下次tick的开始时间
845      schedstat_set(curr->statistics.exec_max,
846                    max(delta_exec, curr->statistics.exec_max));
847      //将本次tick实际运行时间值更新到vruntime和实际运行时间
848      curr->sum_exec_runtime += delta_exec;//sum_exec_runtime等于进程从创建开始占用cpu的总时间
849      schedstat_add(cfs_rq->exec_clock, delta_exec);
850
851      curr->vruntime += calc_delta_fair(delta_exec, curr);
852      update_min_vruntime(cfs_rq);//更新cfs_rq的min_vruntime
853      if (entity_is_task(curr)) {
854          struct task_struct *curtask = task_of(curr);
855          trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
856          cgroup_account_cputime(curtask, delta_exec);
857          account_group_exec_runtime(curtask, delta_exec);
858      }
859      account_cfs_rq_runtime(cfs_rq, delta_exec);
860  }
```

函数 update_curr 调用函数 cals_delta_fair 函数计算考虑权重计算得到的 vruntime，该计算公式为 Vruntime=实际运行时间（delta_exe）*nice 值为 0 的进程的权重/进程的权重之和

```
662  static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
663  {
664      if (unlikely(se->load.weight != NICE_0_LOAD))
665          delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
666
667      return delta;
668  }
```

b. 函数 entity_tick 调用 check_preempt_tick()函数判断当前进程本次调度运行的实际时间是否已经大于其在任何一个调度周期分配得到的实际时间值，如果是则重新设置调度标志

```
4110  static void
4111  check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
4112  {
4113      unsigned long ideal_runtime, delta_exec;
4114      struct sched_entity *se;
4115      s64 delta;
4116
4117      ideal_runtime = sched_slice(cfs_rq, curr);
4118      delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
4119      //得到本次调度已经运行的实际时间
4120      if (delta_exec > ideal_runtime) {
4121          resched_curr(rq_of(cfs_rq));//设置TIF_NEED_RESCHED标志值
4122          /*       * The current task ran long enough, ensure it doesn't get
4123           * re-elected due to buddy favours.*/
4124          clear_buddies(cfs_rq, curr);
4125          return;
4126      }
4127      /*   * Ensure that a task that missed wakeup preemption by a
4128       * narrow margin doesn't have to wait for a full slice.
4129       * This also mitigates buddy induced latencies under load.   */
4130      if (delta_exec < sysctl_sched_min_granularity)
4131          return;
4132      se = __pick_first_entity(cfs_rq);
4133      delta = curr->vruntime - se->vruntime;
4134      if (delta < 0)
4135          return;
4136      if (delta > ideal_runtime)
4137          resched_curr(rq_of(cfs_rq));//设置TIF_NEED_RESCHED标志值
4138  }
```

问题：

1）简述进程优先级、nice 值和权重之间的关系

vruntime 的值表示进程运行的虚拟时间（在处理器上跑的时间累加和），这个 vruntime 值越小，说明该进程应该被优先执行（或者获得更多的处理器时间片），Vruntime=调度周期*nice 值为 0 的进程的权重/所有进程总权重。

其中，NICE_0_LOAD 是 nice 为 0 时的权重。nice 越小即表示权重越大，也就表示该进程有越高的优先级。当 nice 值为 0 证明虚拟运行时间=实际运行时间。

2）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？

a）基本思想：vruntime 的作用是根据进程的权重将运行时间放大或者缩小一个比例，用于记录进程已经运行的时间，CFS 再根据各个进程的权重分配每个进程的运行时间。Vruntime 小的进程说明之前占用 cpu 的时间比较短，优先被选为下一个运行的进程。

b）计算方法：

i. 分配给进程的实际运行时间=调度周期*进程权重/所有进程的权重之和

ii. Vruntime=实际运行时间*nice 值为 0 的进程的权重/进程的权重之和

iii. 综合以上二式，可得 vruntime 的公式：

Vruntime=调度周期*nice 值为 0 的进程的权重/所有进程总权重

此式子说明，在一个调度周期中，所有进程的 vruntime 是一样大的，当每个进程都把分配给自己的运行时间运行完的时候，他们的 vruntime 值是一样的，所以一个进程的 vruntime 值越大，表示进程已经运行的时间占调度器分配给它的运行时间

的比重也越大。因此可以用 vruntime 来选择运行的时间。Vruntime 小的进程说明之前占用 cpu 的时间比较短，CFS 秉着公平的原则，下一个运行的进程就会选择它来达到公平。

这种又考虑的进程已经运行的时间，又考虑了进程的优先级（权重）的调度方式就是 CFS。可以实现公平的选择进程，又能保证高优先级的进程获得比较多的运行时间。

c) 何时得到更新

i. 定时器 tick 中断中

ii. 进程入 CFS 调度队列的时候

iii. 进程出 CFS 调度队列的时候

iv. 进程创建的时候

d) min_vruntime 的作用

Min_vruntime 是当前红黑树中最小的 key 值，使用 min_vruntime 的作用主要是解决 vruntime 的溢出问题。

在代码中，vruntime 是 usigned long 类型的，而红黑树的 key 是 signed long 类型的。因为进程的虚拟时间是一个递增的正值，因此不可能是负数，但是 vruntime 有上限，也就是 usigned long 所能表示的最大值。如果溢出了会从 0 开始回滚。因此在红黑树的 key 中将所有进程的 vruntime 统一减去所有进程中最小的 vruntime，将所有进程的 key 围绕在最小的 vruntime 周围，这样就能避免 vruntime 的溢出。

b) 添加一个内核系统调用，重新编译内核，启动后运行 screenfetch 命令（可能需要安装），截屏显示结果，需要显示出运行主机的内核版本、CPU 等信息（注意：每个同学在自己的机器上编译，这些信息会有所差异，以此作为同学们的作业区分。）

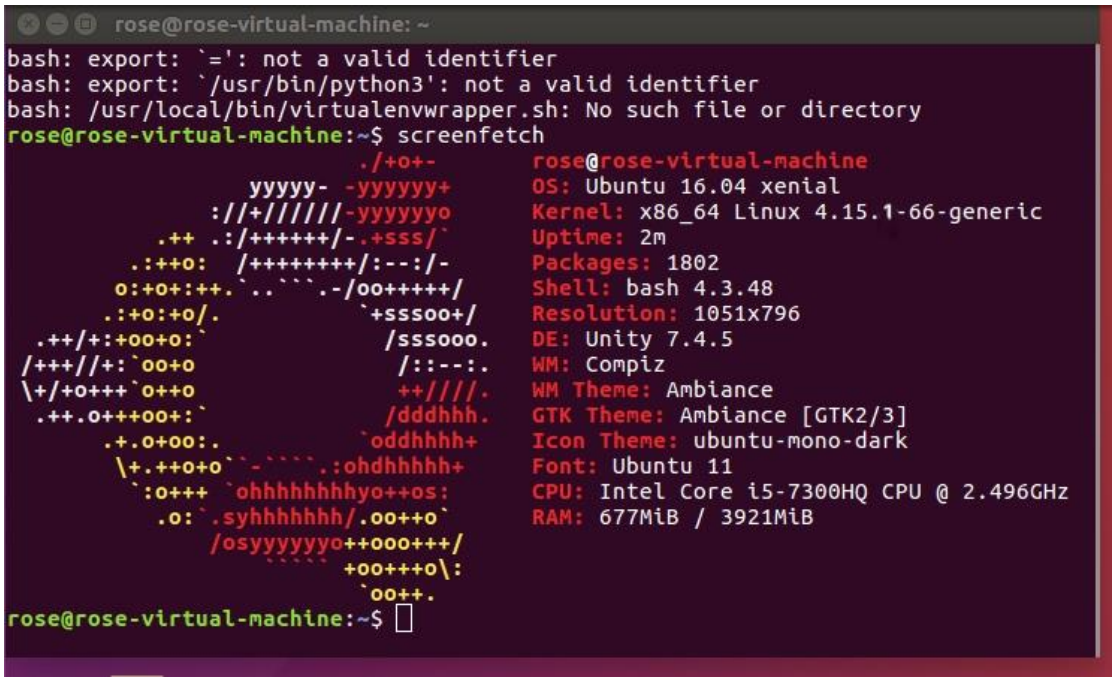
编写用户层程序 mycall.c 调用该调用，要求打印出当前进程的调度信息（如下图所示），通过 dmesg 可以查看。实现时，可以通过 current 访问 sched_entity 的数据成员。

1. 重新编译内核:

在重新编译内核前运行 screenfetch 命令得到的 Linux 系统的各个参数为：

如上图显示在本机中 linux 内核的型号为 4.15.0

在重新编译内核之后运行 screenfetch 命令得到以下结果，说明此时 linux 的内核已经被更换成 4.15.1 的型号。



2. 编写程序后得到结果如下图：（当前正在进行的部分进程的信息）