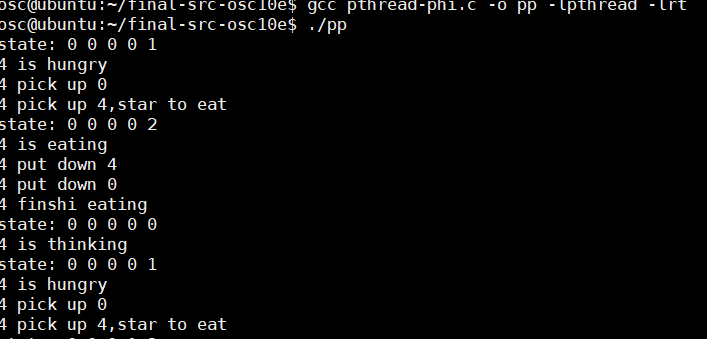
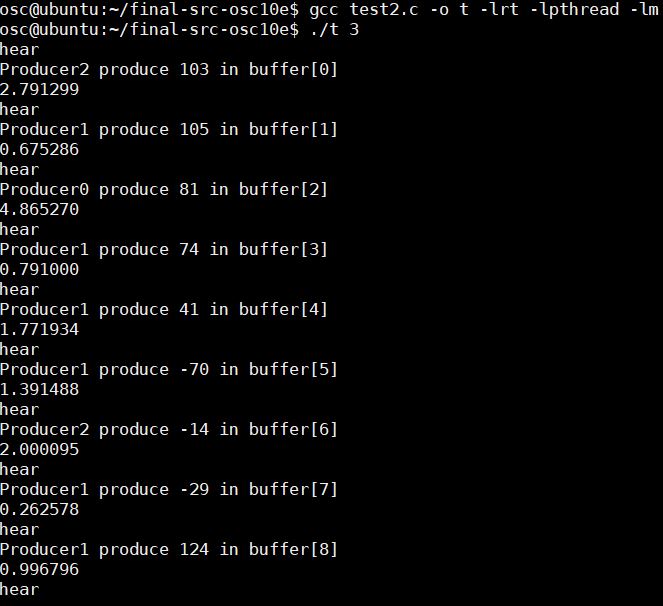
1. **哲学家就餐问题**

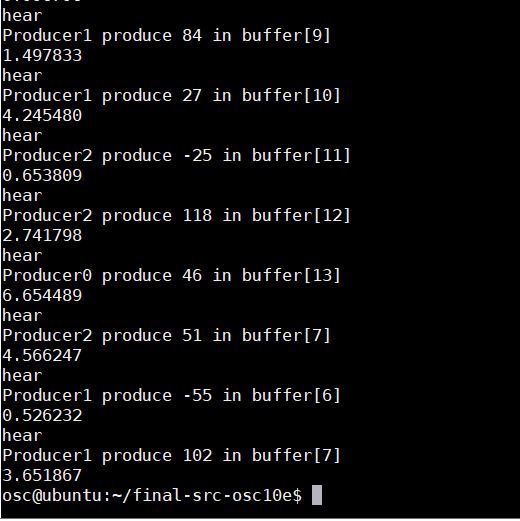


**（具体解释详见代码）**

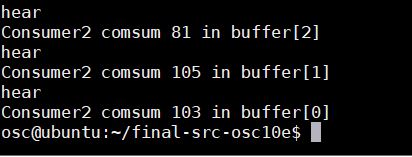
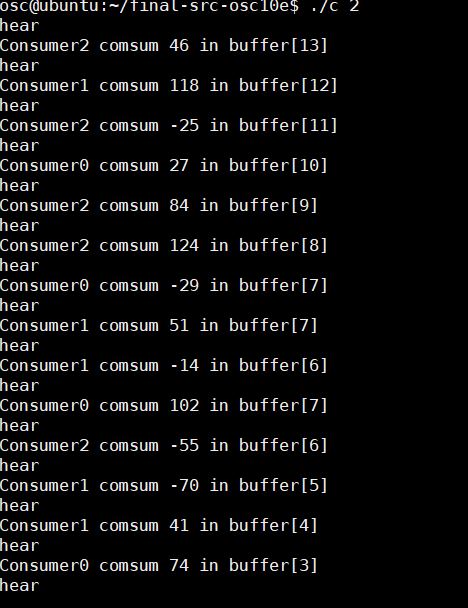
1. **生产者消费者问题**

生产者

****

****

消费者



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](https://www.kernel.org/)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;//
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
12. u64         nr\_migrations;
14. #ifdef CONFIG\_SCHEDSTATS
15. **struct** sched\_statistics statistics;
16. #endif
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
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;
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**;
14. curr->exec\_start = now;//更新进程的开始时间 ，以备下次使用
16. schedstat\_set(curr->statistics.exec\_max,
17. max(delta\_exec, curr->statistics.exec\_max));
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);
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
25. **if** (entity\_is\_task(curr)) {
26. **struct** task\_struct \*curtask = task\_of(curr);
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. }
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);
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;
6. \_\_update\_inv\_weight(lw);
8. **if** (unlikely(fact >> 32)) {
9. **while** (fact >> 32) {
10. fact >>= 1;
11. shift--;
12. }
13. }
15. /\* hint to use a 32x32->64 mul \*/
16. fact = (u64)(u32)fact \* lw->inv\_weight;
18. **while** (fact >> 32) {
19. fact >>= 1;
20. shift--;
21. }
23. **return** mul\_u64\_u32\_shr(delta\_exec, fact, shift);
24. }

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

1. **static** **void** update\_min\_vruntime(**struct** cfs\_rq \*cfs\_rq)
2. {
3. u64 vruntime = cfs\_rq->min\_vruntime;//给当前初始化一个vruntim
5. **if** (cfs\_rq->curr)//如果当前CFS队列上有进程
6. vruntime = cfs\_rq->curr->vruntime;//则当前被选择的vruntime即为该进程的vruntime
8. **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
9. **struct** sched\_entity \*se = rb\_entry(cfs\_rq->rb\_leftmost,
10. **struct** sched\_entity,
11. run\_node);//登记最左节点进程信息
13. **if** (!cfs\_rq->curr)//如果当前CFS队列上没有正在运行的进程
14. vruntime = se->vruntime;//则vruntime为左节点进程的vruntime
15. **else**////如果当前CFS队列上有进程
16. 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
17. }
19. /\* ensure we never gain time by being placed backwards. \*/
20. /\*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)，
21. 所以只有在当前的vruntime超出的cfs\_rq->min\_vruntime的时候才更新\*/
22. cfs\_rq->min\_vruntime = max\_vruntime(cfs\_rq->min\_vruntime, vruntime);
23. #ifndef CONFIG\_64BIT
24. smp\_wmb();
25. cfs\_rq->min\_vruntime\_copy = cfs\_rq->min\_vruntime;
26. #endif
27. }
28. **前面函数中涉及到了CFS队列实例cfs\_rq**

<kernel\sched\sched.h>-line343

1. **struct** cfs\_rq {
2. **struct** load\_weight load;//CFS队列总进程权重
3. unsigned **int** nr\_running, h\_nr\_running;//队列中进程的个数
5. u64 exec\_clock;//运行的时钟
6. u64 min\_vruntime;//CFS队列的vruntime值， 即上一次被选中的队列中vruntime最小值
7. #ifndef CONFIG\_64BIT
8. u64 min\_vruntime\_copy;
9. #endif
11. **struct** rb\_root tasks\_timeline;//红黑树的根节点
12. **struct** rb\_node \*rb\_leftmost;//红黑树的最左节点，即vruntime值最小的节点
14. /\*
15. \* 'curr' points to currently running entity on this cfs\_rq.
16. \* It is set to NULL otherwise (i.e when none are currently running).
17. \*/
18. /\*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 \*/
19. **struct** sched\_entity \*curr, \*next, \*last, \*skip;
21. #ifdef  CONFIG\_SCHED\_DEBUG
22. unsigned **int** nr\_spread\_over;
23. #endif
25. #ifdef CONFIG\_SMP
26. /\*
27. \* CFS load tracking
28. \*/
29. **struct** sched\_avg avg;
30. u64 runnable\_load\_sum;
31. unsigned **long** runnable\_load\_avg;
32. #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
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 \*/
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\*/
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 \*/
68. #ifdef CONFIG\_CFS\_BANDWIDTH
69. **int** runtime\_enabled;
70. u64 runtime\_expires;
71. s64 runtime\_remaining;
73. u64 throttled\_clock, throttled\_clock\_task;
74. u64 throttled\_clock\_task\_time;
75. **int** throttled, throttle\_count;
76. **struct** list\_head throttled\_list;
77. #endif /\* CONFIG\_CFS\_BANDWIDTH \*/
78. #endif /\* CONFIG\_FAIR\_GROUP\_SCHED \*/
79. };
80. **对该队列插入新进程由函数enqueue\_task\_fair实现**

<kernel\sched\fari.c>-line4079

1. **static** **void** enqueue\_task\_fair(**struct** rq \*rq, **struct** task\_struct \*p, **int** flags)
2. {
3. **struct** cfs\_rq \*cfs\_rq;
4. **struct** sched\_entity \*se = &p->se;
6. for\_each\_sched\_entity(se) {//循环插入节点
7. **if** (se->on\_rq)//判断当前节点是否已在队列上
8. **break**;//如果在的话就不用重新插入一遍
9. cfs\_rq = cfs\_rq\_of(se);//得到当前CFS运行队列
10. 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
12. /\*
13. \* end evaluation on encountering a throttled cfs\_rq
14. \*
15. \* note: in the case of encountering a throttled cfs\_rq we will
16. \* post the final h\_nr\_running increment below.
17. \*/
18. **if** (cfs\_rq\_throttled(cfs\_rq))
19. **break**;
20. cfs\_rq->h\_nr\_running++;
22. flags = ENQUEUE\_WAKEUP;
23. }
25. for\_each\_sched\_entity(se) {
26. cfs\_rq = cfs\_rq\_of(se);
27. cfs\_rq->h\_nr\_running++;
29. **if** (cfs\_rq\_throttled(cfs\_rq))
30. **break**;
32. update\_load\_avg(se, 1);
33. update\_cfs\_shares(cfs\_rq);
34. }
36. **if** (!se)
37. add\_nr\_running(rq, 1);
39. hrtick\_update(rq);
40. }

主要操作由函数enqueue\_entity实现

<kernel\sched\fari.c>-line2943

1. **static** **void** enqueue\_entity(**struct** cfs\_rq \*cfs\_rq, **struct** sched\_entity \*se, **int** flags)
2. {
3. /\*
4. \* Update the normalized vruntime before updating min\_vruntime
5. \* through calling update\_curr().
6. \*/
7. **if** (!(flags & ENQUEUE\_WAKEUP) || (flags & ENQUEUE\_WAKING))
8. se->vruntime += cfs\_rq->min\_vruntime;
10. /\*
11. \* Update run-time statistics of the 'current'.
12. 更新当前进程的统计信息
13. \*/
14. update\_curr(cfs\_rq);
15. enqueue\_entity\_load\_avg(cfs\_rq, se);
16. account\_entity\_enqueue(cfs\_rq, se);
17. update\_cfs\_shares(cfs\_rq);
19. **if** (flags & ENQUEUE\_WAKEUP) {//如果当前进程是从休眠中被唤醒的
20. place\_entity(cfs\_rq, se, 0);// 调用函数处理其vruntime
21. enqueue\_sleeper(cfs\_rq, se);
22. }
24. update\_stats\_enqueue(cfs\_rq, se);
25. check\_spread(cfs\_rq, se);
26. **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
27. \_\_enqueue\_entity(cfs\_rq, se);  //具体实现插入操作的函数
28. se->on\_rq = 1;
30. **if** (cfs\_rq->nr\_running == 1) {
31. list\_add\_leaf\_cfs\_rq(cfs\_rq);
32. check\_enqueue\_throttle(cfs\_rq);//抢占？？？
33. }
34. }

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

1. **static** **void** place\_entity(**struct** cfs\_rq \*cfs\_rq, **struct** sched\_entity \*se, **int** initial)
2. {
3. u64 vruntime = cfs\_rq->min\_vruntime;
4. /\*
5. \* The 'current' period is already promised to the current tasks,
6. \* however the extra weight of the new task will slow them down a
7. \* little, place the new task so that it fits in the slot that
8. \* stays open at the end.
9. \*/
10. **if** (initial && sched\_feat(START\_DEBIT))
11. vruntime += sched\_vslice(cfs\_rq, se);
12. /\* sleeps up to a single latency don't count. \*/
13. **if** (!initial) {
14. unsigned **long** thresh = sysctl\_sched\_latency;
15. /\*
16. \* Halve their sleep time's effect, to allow
17. \* for a gentler effect of sleepers:
18. \*/
19. **if** (sched\_feat(GENTLE\_FAIR\_SLEEPERS))
20. thresh >>= 1;
22. vruntime -= thresh;
23. }
24. /\* ensure we never gain time by being placed backwards. \*/
25. se->vruntime = max\_vruntime(se->vruntime, vruntime);
26. }

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

1. **static** **void** \_\_enqueue\_entity(**struct** cfs\_rq \*cfs\_rq, **struct** sched\_entity \*se)
2. {
3. **struct** rb\_node \*\*link = &cfs\_rq->tasks\_timeline.rb\_node;
4. **struct** rb\_node \*parent = NULL;
5. **struct** sched\_entity \*entry;
6. **int** leftmost = 1;
7. /\*
8. \* Find the right place in the rbtree:
9. \*在红黑数中为当前进程找到正确的节点位置并插入‘
10. \*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
11. \*/
12. **while** (\*link) {
13. parent = \*link;
14. entry = rb\_entry(parent, **struct** sched\_entity, run\_node);
15. /\*
16. \* We dont care about collisions. Nodes with
17. \* the same key stay together.
18. \*/
19. **if** (entity\_before(se, entry)) {
20. link = &parent->rb\_left;
21. } **else** {
22. link = &parent->rb\_right;
23. leftmost = 0;
24. }
25. }
26. /\*
27. \* Maintain a cache of leftmost tree entries (it is frequently
28. \* used):
29. \*/
30. **if** (leftmost)
31. cfs\_rq->rb\_leftmost = &se->run\_node;
33. rb\_link\_node(&se->run\_node, parent, link);
34. rb\_insert\_color(&se->run\_node, &cfs\_rq->tasks\_timeline);
35. }

**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）

1. **static** **void** dequeue\_task\_fair(**struct** rq \*rq, **struct** task\_struct \*p, **int** flags)
2. {
3. **struct** cfs\_rq \*cfs\_rq;
4. **struct** sched\_entity \*se = &p->se;
5. **int** task\_sleep = flags & DEQUEUE\_SLEEP;
7. for\_each\_sched\_entity(se) {
8. cfs\_rq = cfs\_rq\_of(se);
9. dequeue\_entity(cfs\_rq, se, flags);
10. ······
11. }

主要操作由 dequeue\_entit完成

<kernel\sched\fari.c>-line4127

1. **static** **void** dequeue\_entity(**struct** cfs\_rq \*cfs\_rq, **struct** sched\_entity \*se, **int** flags)
2. {
3. /\*
4. \* Update run-time statistics of the 'current'.
5. \*/
6. update\_curr(cfs\_rq);
7. dequeue\_entity\_load\_avg(cfs\_rq, se);
9. update\_stats\_dequeue(cfs\_rq, se);
10. clear\_buddies(cfs\_rq, se);
11. ············
12. **if** (se != cfs\_rq->curr)
13. \_\_dequeue\_entity(cfs\_rq, se);  //具体实现移出操作的函数
14. se->on\_rq = 0;
15. account\_entity\_dequeue(cfs\_rq, se);
17. /\*
18. \* Normalize the entity after updating the min\_vruntime because the
19. \* update can refer to the ->curr item and we need to reflect this
20. \* movement in our normalized position.
21. \*/
22. **if** (!(flags & DEQUEUE\_SLEEP))
23. se->vruntime -= cfs\_rq->min\_vruntime;
25. /\* return excess runtime on last dequeue \*/
26. return\_cfs\_rq\_runtime(cfs\_rq);
28. update\_min\_vruntime(cfs\_rq);  //更新min\_vruntime
29. update\_cfs\_shares(cfs\_rq);
30. }

具体实现在 \_\_dequeue\_entity

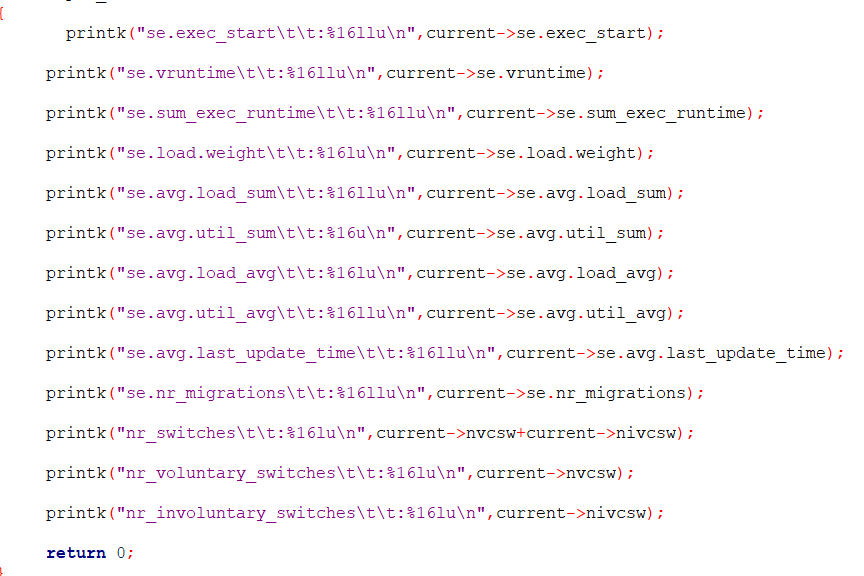
<kernel\sched\fari.c>-line526

1. **static** **void** \_\_dequeue\_entity(**struct** cfs\_rq \*cfs\_rq, **struct** sched\_entity \*se)
2. {
3. **if** (cfs\_rq->rb\_leftmost == &se->run\_node) {
4. **struct** rb\_node \*next\_node;
6. next\_node = rb\_next(&se->run\_node);
7. cfs\_rq->rb\_leftmost = next\_node;
8. }
10. rb\_erase(&se->run\_node, &cfs\_rq->tasks\_timeline);
11. }

**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）

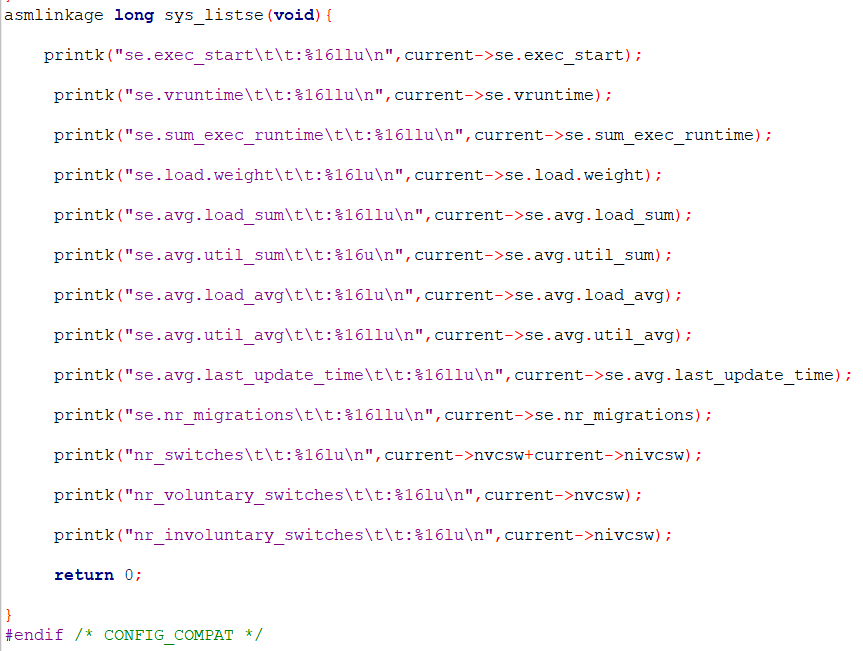


加载到内核中后dmesgm查看有：

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。

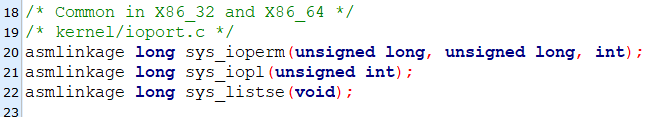
（1）添加系统调用函数

<kernel\sys.c>



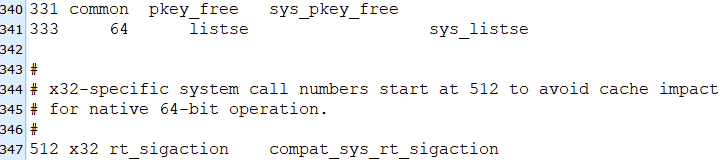
1. 添加声明

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

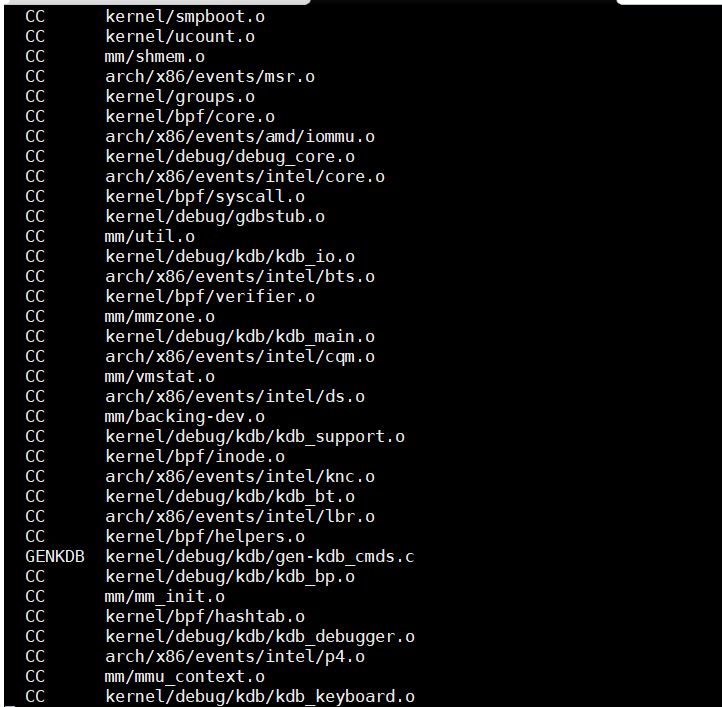


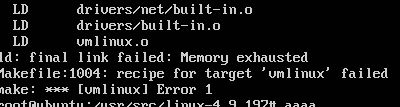
1. 添加系统调动ID

<arch\x86\entry\syscalls\syscall\_64.tbl>

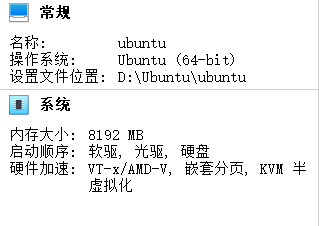


1. 编译并安装新内核

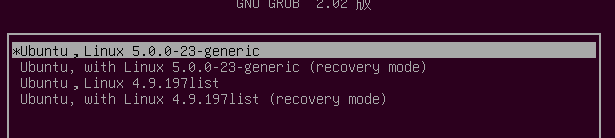


然后就报错了：  


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：



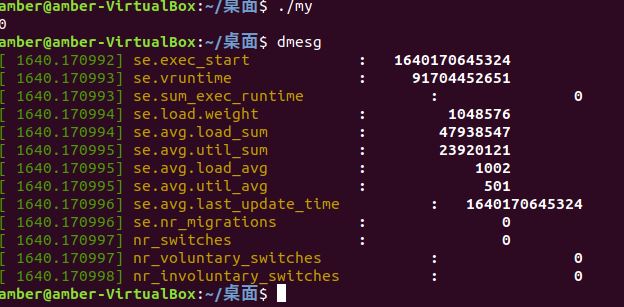
打开新内核打印系统信息：



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。