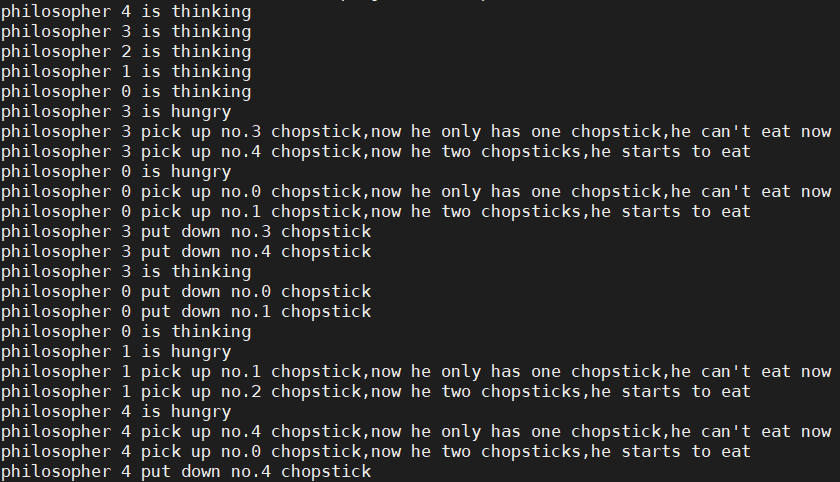
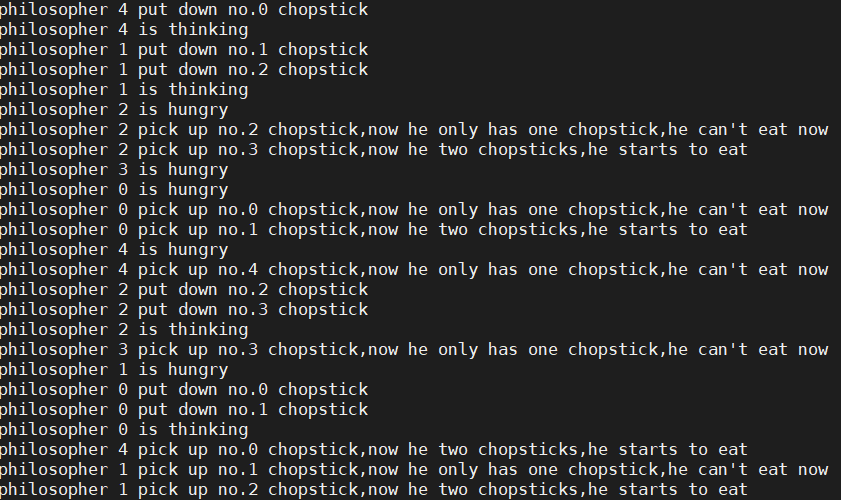
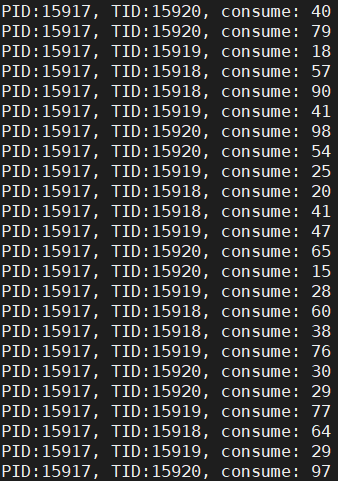
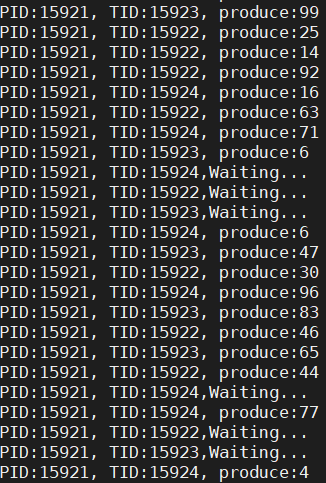
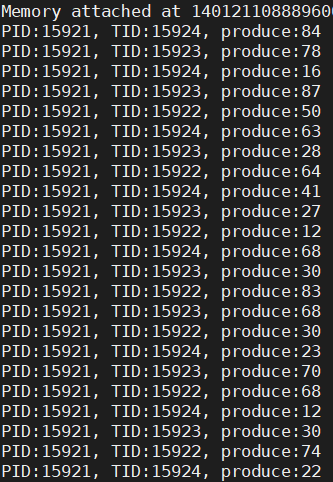
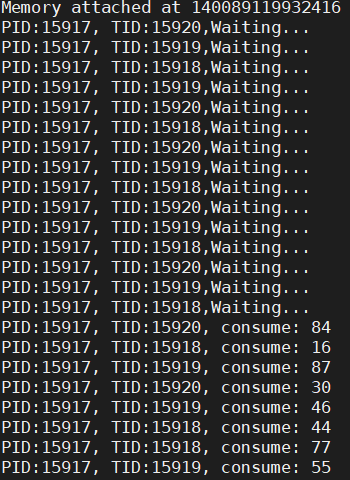
**OS Assignment 01 实验报告**

1. **哲学家就餐问题**

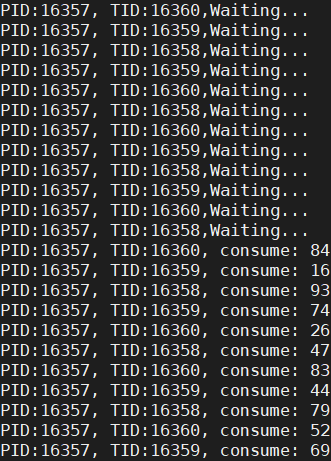
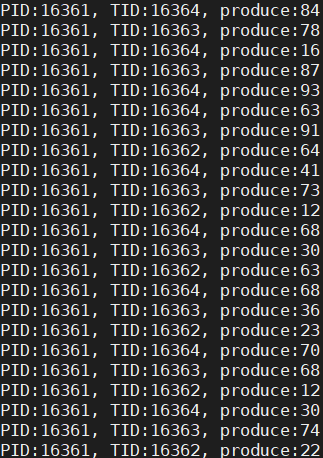


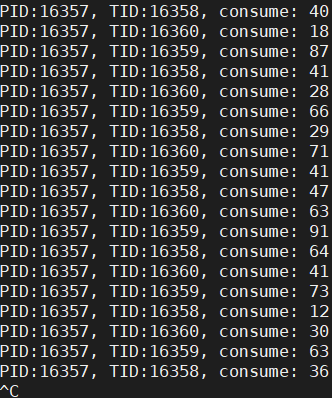
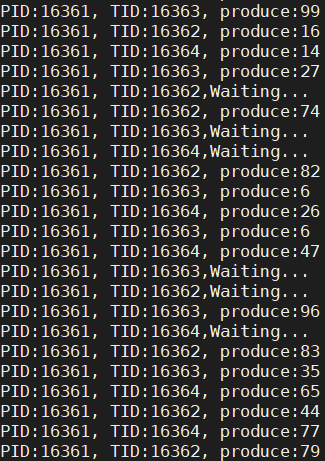


1. **生产者-消费者问题**
2. λp=1,λc=5



1. λp=8,λc=1

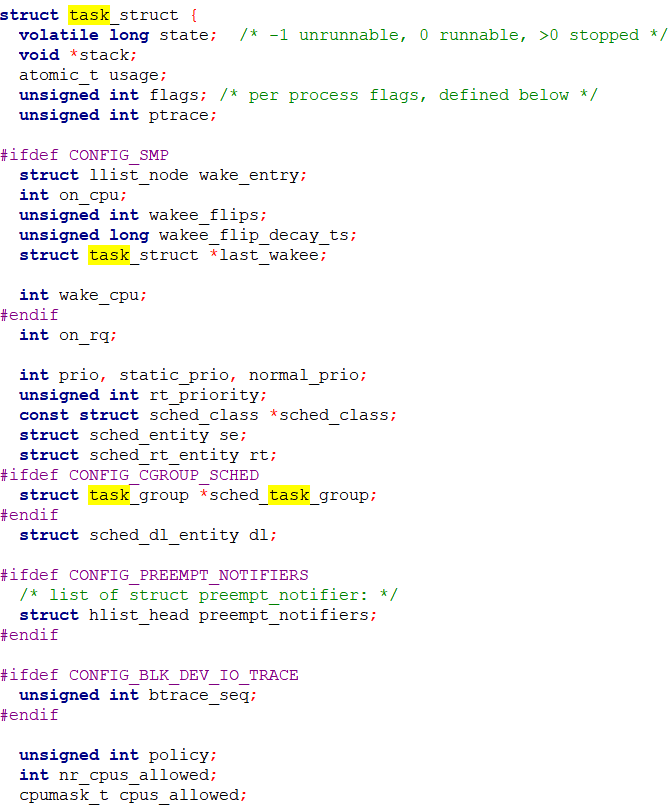




1. **Linux内核实验**

（a）内核代码fair.c阅读与CFS调度算法理解

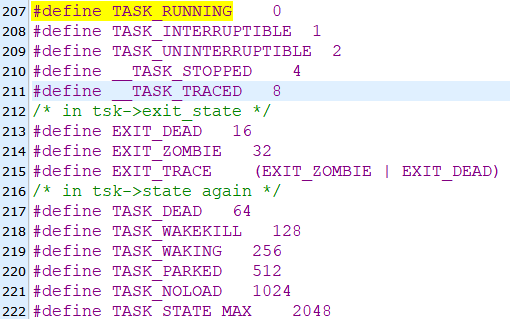
**①进程基本结构：**



……

它包括state进程状态、stack进程内核栈、usage进程描述符使用计数、flags进程当前的状态标志、ptrace系统调用、lock\_depth获取大内核锁的次数、oncpu等等。

**②进程状态：**



1) TASK\_RUNNING: 可运行

处于这种状态的进程，只有两种状态:

1.1) 正在运行

正在运行的进程就是当前进程(由current所指向的进程)

1.2) 正准备运行

准备运行的进程只要得到CPU就可以立即投入运行，CPU是这些进程唯一等待的系统资源，系统中有一个运行队列(run\_queue)，用来容纳所有处于可运行状态的进程，调度程序执行时，从中选择一个进程投入运行

1. TASK\_INTERRUPTIBLE: 可中断的等待状态

它是针对等待某事件或其他资源的睡眠进程设置的，在内核发送信号给该进程表明事件已经发生时，进程状态变为TASK\_RUNNING，它只要调度器选中该进程即可恢复执行

1. TASK\_UNINTERRUPTIBLE: 不可中断的等待状态

处于该状态的进程正在等待某个事件(event)或某个资源，它肯定位于系统中的某个等待队列(wait\_queue)中，处于不可中断等待态的进程是因为硬件环境不能满足而等待，例如等待特定的系统资源，它任何情况下都不能被打断，只能用特定的方式来唤醒它，例如唤醒函数wake\_up()等，它们不能由外部信号唤醒，只能由内核亲自唤醒

4) TASK\_ZOMBIE: 僵死

进程虽然已经终止，但由于某种原因，父进程还没有执行wait()系统调用，终止进程的信息也还没有回收。顾名思义，处于该状态的进程就是死进程，这种进程实际上是系统中的垃圾，必须进行相应处理以释放其占用的资源。

5) TASK\_STOPPED: 暂停

此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到SIGSTOP、SIGTSTP、SIGTTIN或 SIGTTOU信号后就处于这种状态。例如，正接受调试的进程就处于这种状态

6) TASK\_TRACED

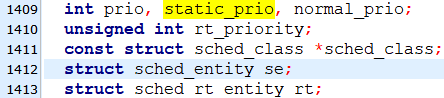
　　从本质上来说，这属于TASK\_STOPPED状态，用于从停止的进程中，将当前被调试的进程与常规的进程区分开来

7) TASK\_DEAD

　　父进程wait系统调用发出后，当子进程退出时，父进程负责回收子进程的全部资源，子进程进入TASK\_DEAD状态

1. TASK\_SWAPPING: 换入/换出

**③CPU调度基本架构：**



1. prio: 保存调度器考虑的优先级
2. static\_prio: 用于保存进程的"静态优先级"
3. normal\_prio: 表示基于进程的"静态优先级"和"调度策略"计算出的优先级
4. rt\_priority: 表示实时进程的优先级
5. sched\_class: 该进程所属的调度类，目前内核中有实现以下四种：

5.1) static const struct sched\_class fair\_sched\_class;

5.2) static const struct sched\_class rt\_sched\_class;

5.3) static const struct sched\_class idle\_sched\_class;

5.4) static const struct sched\_class stop\_sched\_class;

6）se: 用于普通进程的调用实体

7）rt: 用于实时进程的调用实体

**CFS调度算法基本流程：**

CFS的总体思路是根据各个进程的权重分配运行时间。它给cfs\_rq中的每一个进程安排一个虚拟时钟，vruntime。如果一个进程得以执行，随着时间的增长，其vruntime将不断增大。没有得到执行的进程vruntime不变。而调度器总是选择vruntime跑得最慢的那个进程来执行。为了区别不同优先级的进程，优先级高的进程vruntime增长得慢，以至于它可能得到更多的运行机会。

实现vruntime的方式为

Vruntime = 实际运行时间 \* nice为0的进程权重 / 进程权重

而又因为

分配给进程的运行时间 = 调度周期 \* 进程权重 / 所有进程权重之和

所以宏观上

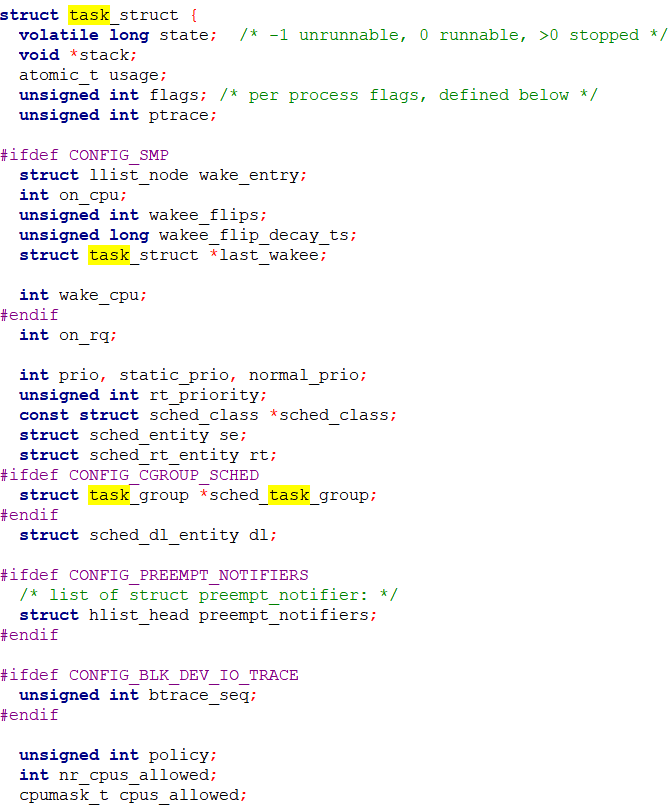
vruntime = 调度周期 \* nice为0的进程权重 / 所有进程总权重

具有相同的增长速度，与进程权重无关，因此用vruntime选择运行的进程，既能公平选择进程，又能保证高优先级进程获得较多的运行时间。

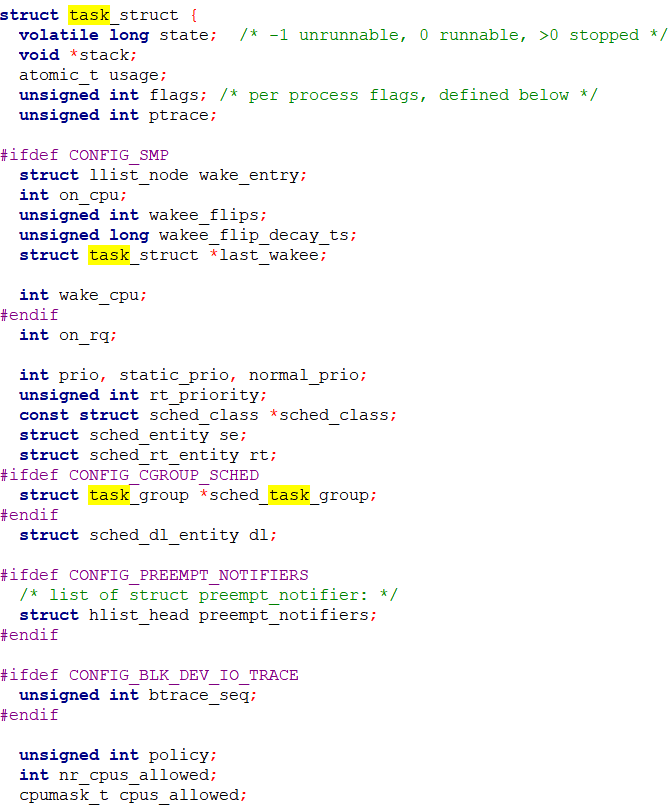
**CFS数据结构：**

调度实体sched\_entity，它代表一个调度单位，每一个task\_struct中都有一个sched\_entity，进程的vruntime和权重都保存在这个结构中。所有的sched\_entity通过红黑树组织在一起，所有的sched\_entity以vruntime-vruntime-min为key插入到红黑树中，同时缓存树的最左侧节点，也就是vruntime最小的节点，这样可以迅速选中vruntime最小的进程。

**Task\_struct:**

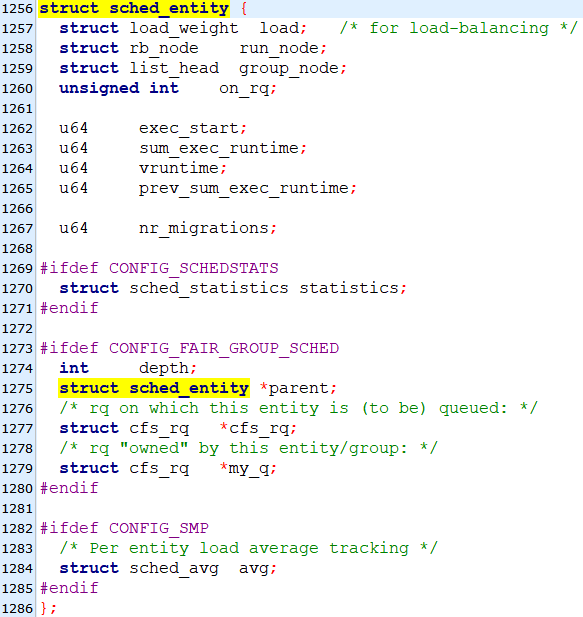


……

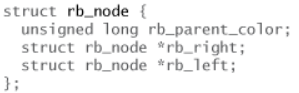


……

**Sched\_entity:**

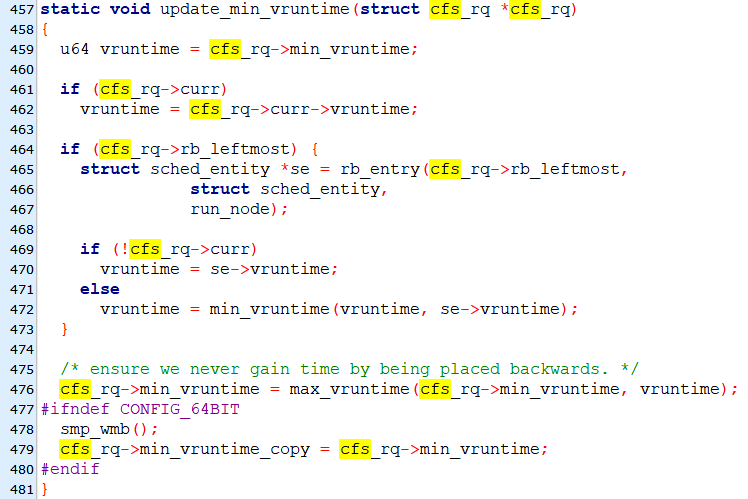


**rb\_node:**

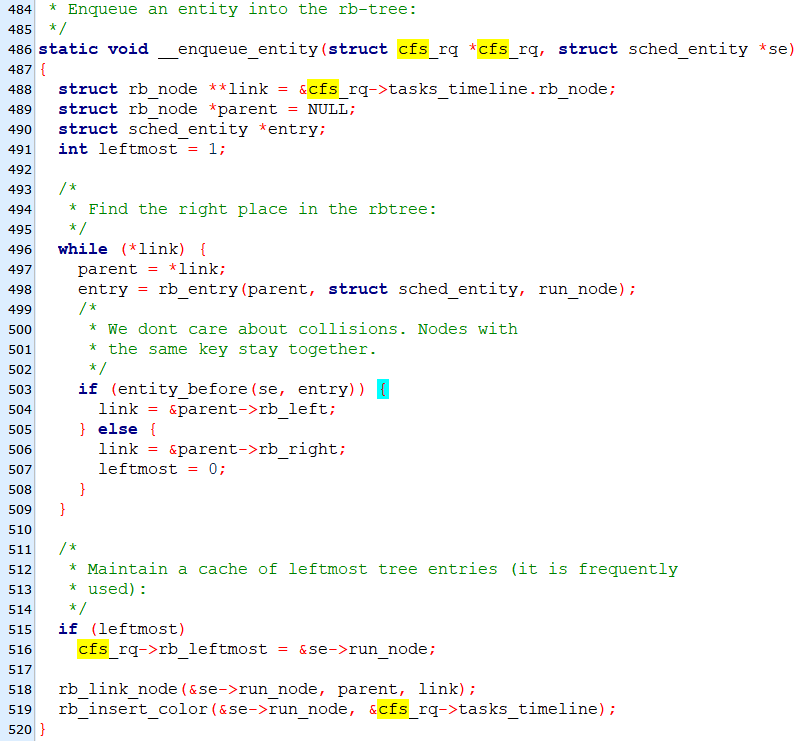


**关键代码：**

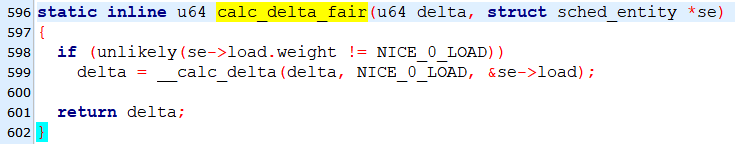
1）更新最小进程虚拟时间：



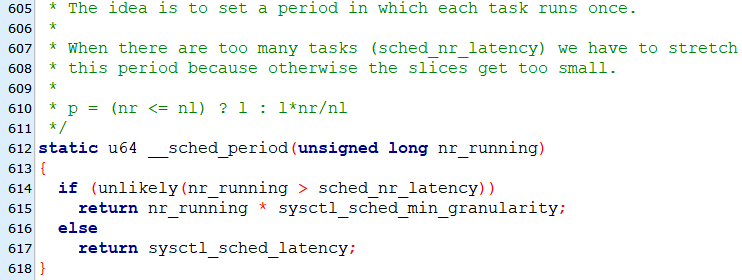
2）插入数据项到红黑树中：



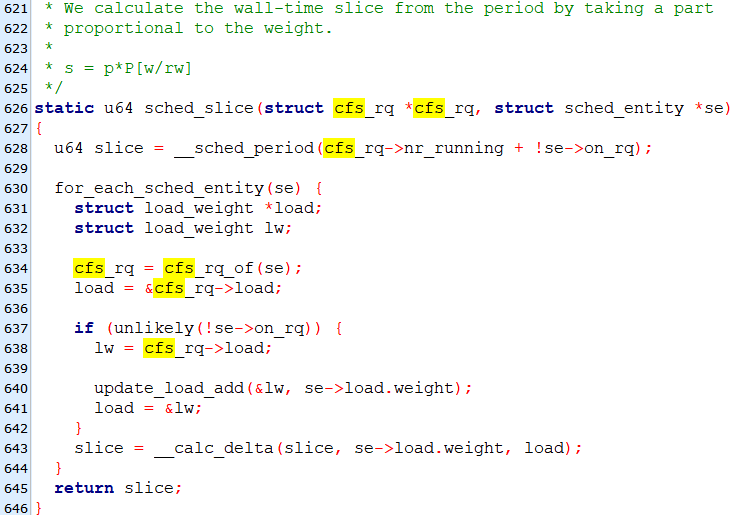
3）计算vruntime：



4）计算调度延迟时间片：



5）计算分配时间片：



**回答问题：**

1. 简述进程优先级、nice值和权重之间的关系？

答：权重跟进程nice值之间有一一对应的关系，可以通过全局数组prio\_to\_weight来转换，nice值越大，权重越低。而权重越低，导致Vruntime = 实际运行时间 \* nice为0的进程权重 / 进程权重算法得到的Vruntime值越高，即Vruntime增长速度越快，这样一来与运行相同时间的进程相比它所获得的cpu分配时间更少，也即优先级更低。

1. 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？

答：vruntime是记录进程运行时间的虚拟时钟，用以为调度算法提供选择进程的依据。如果一个进程得以执行，随着时间的增长，其vruntime将不断增大。没有得到执行的进程vruntime不变。为了区别不同优先级的进程，优先级高的进程vruntime增长得慢。

它的计算方式是

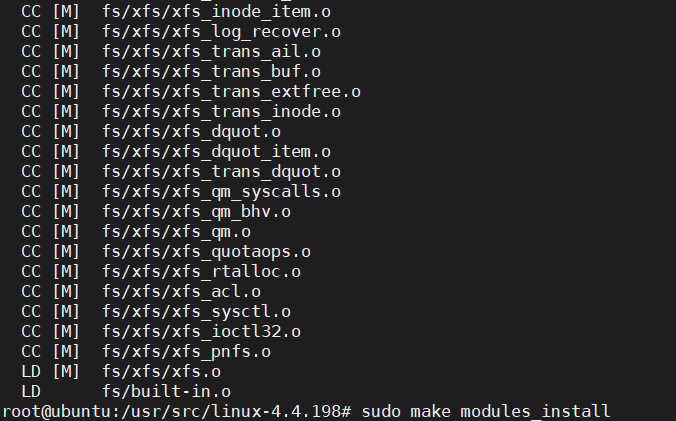
Vruntime = 实际运行时间 \* nice为0的进程权重 / 进程权重

在cfs\_rq稳定的情况下，每隔一个调度延迟时间片vruntime更新一次；但是当cfs\_rq发生变化，随着不断有进程enqueue/dequeue、或者改变优先级，正在执行的进程的时间片是随时在变化的。如果某一时刻，cfs发现正在执行的进程用完了它的时间片，就会更新；或者当正在执行的进程的vruntime与cfs\_rq中最小的vruntime之差大于它的时间片时，也会更新。

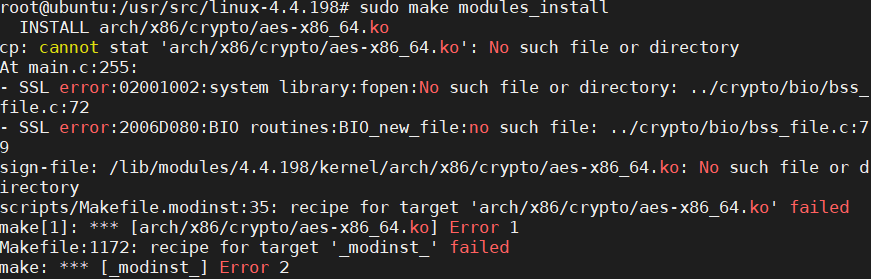
min\_vruntime是用来处理溢出问题的。进程的虚拟时间vruntime是一个递增的正值，为了防止它溢出导致本末倒置的混乱，我们将红黑树中的key值用vruntime-min - vruntime来表示。这样所有进程的key围绕在最小vruntime的周围，更加容易追踪。

（b）编译内核

编译已完成。



但在执行make modules\_install指令时，出现如下错误。



还没来得及解决，我会尽快补上。

**原内核上screenfetch截图：**

