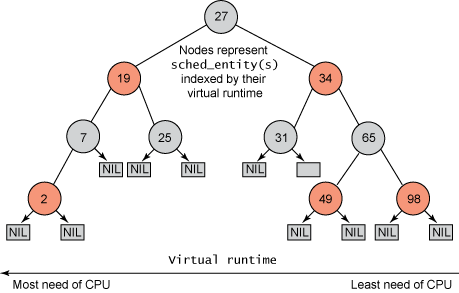
李沛冀 1212300209 unix系统分析作业2

1.概述  
      CFS（completely fair schedule）是最终被内核采纳的调度器。它从RSDL/SD中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。CFS的算法和实现都相当简单，众多的测试表明其性能也非常优越。  
  
      CFS 背后的主要想法是维护为任务提供处理器时间方面的平衡（公平性）。这意味着应给进程分配相当数量的处理器。分给某个任务的时间失去平衡时（意味着一个或多个任务相对于其他任务而言未被给予相当数量的时间），应给失去平衡的任务分配时间，让其执行。   
  
      CFS抛弃了时间片，抛弃了复杂的算法，从一个新的起点开始了调度器的新时代，最开始的2.6.23版本，CFS提供一个虚拟的时钟，所有进程复用这个虚拟时钟的时间，CFS将时钟的概念从底层体系相关的硬件中抽象出来，进程调度模块直接和这个虚拟的时钟接口 而不必再为硬件时钟操作而操心，如此一来，整个进程调度模块就完整了，从时钟到调度算法，到不同进程的不同策略，全部都由虚拟系统提供，也正是在这个新的内核，引入了调度类。因此新的调度器就是不同特性的进程在统一的虚拟时钟下按照不同的策略被调度。  
  
      按照作者Ingo Molnar的说法："CFS百分之八十的工作可以用一句话概括：CFS在真实的硬件上模拟了完全理想的多任务处理器"。在“完全理想的多任务处理器 “下，每个进程都能同时获得CPU的执行时间。当系统中有两个进程时，CPU的计算时间被分成两份，每个进程获得50%。然而在实际的硬件上，当一个进程占用CPU时，其它进程就必须等待。这就产生了不公平。  
  
2.相关概念  
调度实体(sched entiy)：就是调度的对象，可以理解为进程。  
虚拟运行时间(vruntime)：即每个调度实体的运行时间。任务的虚拟运行时间越小， 意味着任务被允许访问服务器的时间越短 — 其对处理器的需求越高。  
公平调度队列(cfs\_rq)：采取公平调度的调度实体的运行队列。  
  
3.CFS的核心思想  
      全公平调度器（CFS）的设计思想是：在一个真实的硬件上模型化一个理想的、精确的多任务CPU。该理想CPU模型运行在100%的负荷、在精确 平等速度下并行运行每个任务，每个任务运行在1/n速度下，即理想CPU有n个任务运行，每个任务的速度为CPU整个负荷的1/n。  
      由于真实硬件上，每次只能运行一个任务，这就得引入"虚拟运行时间"（virtual runtime）的概念，虚拟运行时间为一个任务在理想CPU模型上执行的下一个时间片(timeslice)。实际上，一个任务的虚拟运行时间为考虑到运行任务总数的实际运行时间。   
  
      CFS 背后的主要想法是维护为任务提供处理器时间方面的平衡（公平性）。CFS为了体现的公平表现在2个方面  
(1)进程的运行时间相等  
      CFS 在叫做虚拟运行时 的地方维持提供给某个任务的时间量。任务的虚拟运行时越小， 意味着任务被允许访问服务器的时间越短 — 其对处理器的需求越高。  
      例如，如果具有 4 个可运行的任务，那么 fair\_clock 将按照实际时间速度的四分之一增加。每个任务将设法跟上这个速度。这是由分时多任务处理的量子化特性决定的。也就是说，在任何一个时间段内只有一个任务可以运行；因此， 其他进程在时间上的拖欠将增大（wait\_runtime）。因此，一旦某个任务进入调度，它将努力赶上它所欠下的时间（并且要比所欠时间多一点，因为在追赶时间期间，fair\_clock 不会停止计时）。

       加权任务引入了优先级。假设我们有两个任务：其中一个任务占用 CPU 的时间量是另一个任务的两倍，比例为 2:1。执行数学转换后，对于权重为 0.5 的任务，时间流逝的速度是以前的两倍。

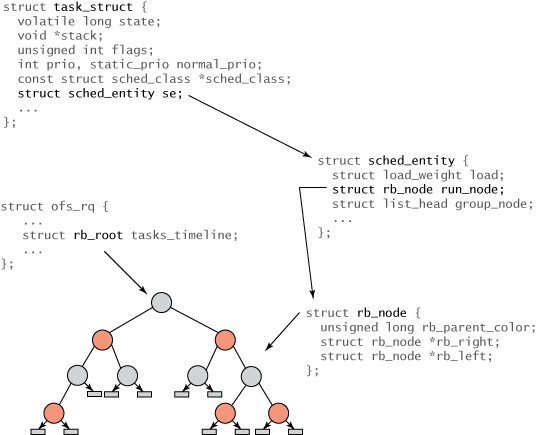
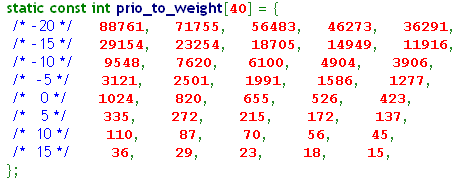
(2)睡眠的进程进行补偿  
      CFS 还包含睡眠公平概念以便确保那些目前没有运行的任务（例如，等待 I/O）在其最终需要时获得相当份额的处理器。   
  
      CFS调度器的运行时间是O(logN)，而以前的调度器的运行时间是O(1)，这是不是就是说CFS的效率比O(1)的更差呢？  
      答案并不是那样，我们知道 CFS调度器下的运行队列是基于红黑树组织的，找出下一个进程就是截下左下角的节点，固定时间完成，所谓的O(logN)指的是插入时间，可是红黑树的统 计性能是不错的，没有多大概率真的用得了那么多时间，因为红节点和黑节点的特殊排列方式既保证了树的一定程度的平衡，又不至于花太多的时间来维持这种平 衡，插入操作大多数情况下都可以很快的完成，特别是对于组织得相当好的数据。  
  
4.CFS工作原理  
      CFS 调度程序使用安抚（appeasement）策略确保公平性。当某个任务进入运行队列后，将记录当前时间，当某个进程等待 CPU 时，将对这个进程的 wait\_runtime 值加一个数，这个数取决于运行队列当前的进程数。当执行这些计算时，也将考虑不同任务的优先级值。 将这个任务调度到 CPU 后，它的 wait\_runtime 值开始递减，当这个值递减到其他任务成为红黑树的最左侧任务时，当前任务将被抢占。通过这种方式，CFS 努力实现一种理想 状态，即 wait\_runtime 值为 0！  
  
      CFS 维护任务运行时（相对于运行队列级时钟，称为 fair\_clock(cfs\_rq->fair\_clock)），它在某个实际时间的片段内运行，因此，对于单个任务可以按照理想的速度运行。   
      最后，根据 fair\_clock 对树进行排队。

**5.CFS的实现**  
**5.1 2.6.23 VS 2.6.25**  
      在2.6.23内核中，刚刚实现的CFS调度器显得很淳朴，每次的时钟滴答中都会将当前进程先出队，推进其虚拟时钟和系统虚拟时钟后再入队，然后判断红黑 树的左下角的进程是否还是当前进程而抉择是否要调度，这种调度器的key的计算是用当前的虚拟时钟减去待计算进程的等待时间，如果该计算进程在运行，那么其等待时间就是负值，这样，等待越长的进程key越小，从而越容易被选中投入运行；  
      在2.6.25内核以后实现了一种更为简单的方式，就是设置一个运行队列的虚拟时钟，它单调增长并且跟踪该队列的最小虚拟时钟的进程，key值由进程的vruntime和队列的虚拟时钟的差值计算，这种方式就是真正的追赶， 比2.6.23实现的简单，但是很巧妙，不必在每次时钟滴答中都将当前进程出队，入队，而是根据当前进程实际运行的时间和理想应该运行的时间判断是否应该调度。  
  
**5.2红黑树**  
      与之前的 Linux 调度器不同，它没有将任务维护在运行队列中，CFS 维护了一个以时间为顺序的红黑树（参见下图）。 红黑树 是一个树，具有很多有趣、有用的属性。首先，它是自平衡的，这意味着树上没有路径比任何其他路径长两倍以上。 第二，树上的运行按 O(log n) 时间发生（其中 n 是树中节点的数量）。这意味着您可以快速高效地插入或删除任务。   
[](http://blog.chinaunix.net/attachment/201109/19/122937_1316415385i84g.gif)  
  
      任务存储在以时间为顺序的红黑树中（由 sched\_entity 对象表示），对处理器需求最多的任务 （最低虚拟运行时）存储在树的左侧，处理器需求最少的任务（最高虚拟运行时）存储在树的右侧。 为了公平，调度器先选取红黑树最左端的节点调度为下一个以便保持公平性。任务通过将其运行时间添加到虚拟运行时， 说明其占用 CPU 的时间，然后如果可运行，再插回到树中。这样，树左侧的任务就被给予时间运行了，树的内容从右侧迁移到左侧以保持公平。 因此，每个可运行的任务都会追赶其他任务以维持整个可运行任务集合的执行平衡。   
  
**5.3调度类**  
      调度类类似于一个模块链，协助内核调度程序工作。每个调度程序模块需要实现 struct sched\_class 建议的一组函数。  
代码如下

1. struct sched\_class { /\* Defined in 2.6.23:/usr/include/linux/sched.h \*/
2. struct sched\_class \*next;
3. void (\*enqueue\_task) (struct rq \*rq, struct task\_struct \*p, int wakeup);
4. void (\*dequeue\_task) (struct rq \*rq, struct task\_struct \*p, int sleep);
5. void (\*yield\_task) (struct rq \*rq, struct task\_struct \*p);
6. void (\*check\_preempt\_curr) (struct rq \*rq, struct task\_struct \*p);
7. struct task\_struct \* (\*pick\_next\_task) (struct rq \*rq);
8. void (\*put\_prev\_task) (struct rq \*rq, struct task\_struct \*p);
9. unsigned long (\*load\_balance) (struct rq \*this\_rq, int this\_cpu,
10. struct rq \*busiest,
11. unsigned long max\_nr\_move, unsigned long max\_load\_move,
12. struct sched\_domain \*sd, enum cpu\_idle\_type idle,
13. int \*all\_pinned, int \*this\_best\_prio);
14. void (\*set\_curr\_task) (struct rq \*rq);
15. void (\*task\_tick) (struct rq \*rq, struct task\_struct \*p);
16. void (\*task\_new) (struct rq \*rq, struct task\_struct \*p);
17. };

**函数描述**

* enqueue\_task：当某个任务进入可运行状态时，该函数将得到调用。它将调度实体（进程）放入红黑树中，并对 nr\_running 变量加 1。
* dequeue\_task：当某个任务退出可运行状态时调用该函数，它将从红黑树中去掉对应的调度实体，并从 nr\_running 变量中减 1。
* yield\_task：在 compat\_yield sysctl 关闭的情况下，该函数实际上执行先出队后入队；在这种情况下，它将调度实体放在红黑树的最右端。
* check\_preempt\_curr：该函数将检查当前运行的任务是否被抢占。在实际抢占正在运行的任务之前，CFS 调度程序模块将执行公平性测试。这将驱动唤醒式（wakeup）抢占。
* pick\_next\_task：该函数选择接下来要运行的最合适的进程。
* load\_balance：每个调度程序模块实现两个函数，load\_balance\_start() 和 load\_balance\_next()，使用这两个函数实现一个迭代器，在模块的 load\_balance 例程中调用。内核调度程序使用这种方法实现由调度模块管理的进程的负载平衡。
* set\_curr\_task：当任务修改其调度类或修改其任务组时，将调用这个函数。
* task\_tick：该函数通常调用自 time tick 函数；它可能引起进程切换。这将驱动运行时（running）抢占。
* task\_new：内核调度程序为调度模块提供了管理新任务启动的机会。CFS 调度模块使用它进行组调度，而用于实时任务的调度模块则不会使用这个函数。

**4.4 CFS内部原理**  
      CFS 去掉了 struct prio\_array，并引入调度实体（scheduling entity）和调度类 （scheduling classes），分别由 struct sched\_entity 和 struct sched\_class 定义。因此，task\_struct 包含关于 sched\_entity 和 sched\_class 这两种结构的信息。详见 ./linux/include/linux/sched.h   
  
[](http://blog.chinaunix.net/attachment/201109/19/122937_1316416769MTZY.gif)  
  
      树的根通过 rb\_root 元素通过 cfs\_rq 结构（在 ./kernel/sched.c 中）引用。红黑树的叶子不包含信息，但是内部节点代表一个或多个可运行的任务。红黑树的每个节点都由 rb\_node 表示，它只包含子引用和父对象的颜色。 rb\_node 包含在 sched\_entity 结构中，该结构包含 rb\_node 引用、负载权重以及各种统计数据。最重要的是， sched\_entity 包含 vruntime（64 位字段），它表示任务运行的时间量，并作为红黑树的索引。 最后，task\_struct 位于顶端，它完整地描述任务并包含 sched\_entity 结构。   
  
      CFS 调度函数非常简单。 在 ./kernel/sched.c 中的 schedule() 函数中，它会先抢占当前运行任务（除非它通过 yield() 代码先抢占自己）。注意 CFS 没有真正的时间切片概念用于抢占，因为抢占时间是可变的。 当前运行任务（现在被抢占的任务）通过对 put\_prev\_task 调用（通过调度类）返回到红黑树。 当 schedule 函数开始确定下一个要调度的任务时，它会调用 pick\_next\_task 函数。此函数也是通用的（在 ./kernel/sched.c 中），但它会通过调度器类调用 CFS 调度器。 CFS 中的 pick\_next\_task 函数可以在 ./kernel/sched\_fair.c（称为 pick\_next\_task\_fair()）中找到。 此函数只是从红黑树中获取最左端的任务并返回相关 sched\_entity。通过此引用，一个简单的 task\_of() 调用确定返回的 task\_struct 引用。通用调度器最后为此任务提供处理器。  
  
**4.4 CFS的优先级**  
      CFS 不直接使用优先级而是将其用作允许任务执行的时间的衰减系数。 低优先级任务具有更高的衰减系数，而高优先级任务具有较低的衰减系数。 这意味着与高优先级任务相比，低优先级任务允许任务执行的时间消耗得更快。 这是一个绝妙的解决方案，可以避免维护按优先级调度的运行队列。  
  
**2.CFS是如何实现公正调度的**  
**2.2每个进程的weight值的确定**  
      CFS的公平依据就是每个调度实体的权重(weight)，这个权重是有优先级来决定的，即优先级越高权重越高，linux内核采用了从nice 到 prio 到 weight的一个转换关系来实现了每个调度实体权重的确定。  
      进程被创建的时候他的优先级是继承自父进程的，如果想改变有优先级，linux内核提供了几个系统调用来改变进程的nice值，从而改变权重，不如sys\_nice()系统调用，下面来看一下他们之间的转换关系：  
                #define NICE\_TO\_PRIO(MAX\_RT\_PRIO + (nice) + 20)  
                #define PRIO\_TO\_NICE((prio) - MAX\_RT\_PRIO - 20)  
       其中，MAX\_RT\_PRIO=100,nice的值在-20到19之前，那么优先级就在100 - 139之间。  
       说明：用户进程的优先级为101-140，前100个是分给实现实时进程的。  
  
prio和weight之间的转换关系，这是个经验公式，如下图所示。  
[](http://blog.chinaunix.net/attachment/201109/19/122937_13164037567xTq.png)  
      通过以上分析我们就可以通过修改nice来修改weight了。(未讲解清楚)  
  
  
CFS 组调度  
考虑一个两用户示例，用户 A 和用户 B 在一台机器上运行作业。用户 A 只有两个作业正在运行，而用户 B 正在运行 48 个作业。组调度使 CFS 能够对用户 A 和用户 B 进行公平调度，而不是对系统中运行的 50 个作业进行公平调度。每个用户各拥有 50% 的 CPU 使用。用户 B 使用自己 50% 的 CPU 分配运行他的 48 个作业，而不会占用属于用户 A 的另外 50% 的 CPU 分配。