**Linux非实时任务调度CFS算法描述**

130904001 白明辉

**摘要:**CFS算法是基于一个理想的，精确的多任务CPU设计的（在这种CPU下，各个任务的运行速度是一致的），实际上这个CPU并不存在，因此，算法模拟了硬件的实现，通过分割CPU的时间使得每个任务的运行时间总是相等的。

**关键字:** 非实时任务调度算法，任务阻塞，任务唤醒，调度，调度类的实现

**一、CFS算法描述**

Linux非实时任务调度算法在实现上，每个任务控制块的调度实体上有一个变量vruntime保存了本任务的运行时间，调度时找时间最小的任务运行，使得最后每个任务的运行时间都相等。  
 当然，不会当运行时间最小的任务一大于第二小的任务时就触发调度，这样，将会导致系统频繁的进行任务切换，而为了更好的利用cache，如开始t1和t2的vruntime相等，任务t1刚运行非常小的时间，判断t1的运行时间大于t2，会检查这两者的差值是否大于系统设置的调度粒度，即t1->vruntime - t2-  
>vruntime > sysctl\_sched\_min\_granularity才运行触发一次调度。t1，t2的状态为ready。另外一个调度粒度是sysctl\_sched\_wakeup\_granularity，这个表示当前任务的运行时间必须大于被唤醒任务的运行时间+该粒度值，才运行触发调度。所有待调度的任务以vruntime为key值，通过rbtree进行排序。  
任务触发一次yield操作时，依赖于是否使能了sysctl\_sched\_compat\_yield， 如果使能了，则会把任务放到rbtree的最右边，表示最后调度----该动能与以前的yield操作兼容。否则就是重算vruntime并插入红黑树。默认情况下并不使能sysctl\_sched\_compat\_yield。

1.Vruntime

vruntime是以纳秒为单位，指示了进程运行了多长时间（不是真实时间，是一个折算出来的时  
间），是理解cfs调度算法的关键。该值在时间中断或者任务状态发生改变时会更新（入队、出队等）。

2.cfs\_rq->min\_vruntime

当前队列上任务的最低运行时间，一个单调递增值。

两种情况下更新该值：1、更新当前运行任务的累计运行时间时；2、当任务从队列删除去，如任务睡眠或退出，这时候会查看剩下的任务的vruntime是否大于min\_vruntime，如果是则更新该值。

3.理想的任务运行时间片get\_rr\_interval\_fair/\_\_sched\_period

任务个数小于sched\_nr\_latency（8）时，period为sysctl\_sched\_latency，否则period 为任务个数\*sysctl\_sched\_min\_granularity。对于实际任务来说，还需要根据任务优先级（NICE值）转化：即slice = period \* 任务的load值/当前队列的总load值。

4.任务运行时间转为vruntime(sched\_vslice/calc\_delta\_fair)

vslice = runtime \* NICE\_0\_LOAD / 当前任务的load值，可以看出，任务的load值越大（高优先级，即  
NICE值较小），vslice越小，相同运行时间下，优先级高的任务vruntime增加的越慢，从而可以保证优  
先级高的任务运行的时间更长。

1. 在支持hrtick的系统上，任务被挑选运行或者入队列等情况，会计算下一次调度的时间点（就是sched\_slice），然后启动定时器。可以控制定时器的到来，减少无谓的时间中断
2. **CFS算法的具体实现**
3. 任务创建时（task\_fork\_fair/place\_entity，wake\_up\_new\_task）

先更新当前任务（父任务）的运行时间，然后把创建者的运行时间赋予被创建任务（子任务继承了父任务的时间）。为什么要继承，考虑不继承的情况下，初始值赋予0有什么问题？如果不继承的话，父任务啥都不做，只管创建简短的任务，这样，每个子任务将优先得到运行（时间为0啊），对于其他不创建任务者来说，是不公平的。  
 由于创建代码的运行会拖慢创建者的速度（相对于被创建者来说，就是它借用了创建者的时间启动自己），因此，如果在开启\_\_SCHED\_FEAT\_START\_DEBIT特性的情况下，会把这个时间加到被创建者的初始时间值，这个时间简单的计算为任务的运行时间片。然后减去当前队列的最少运行时间min\_vruntime。创建完成后调用wake\_up\_new\_task，入合适的队列，这时候会把当前队列的最小值  
min\_vruntime加回去（见后面第2点，任务唤醒处理）。  
 于是，又公平了，同一起跑线上。

1. 任务被阻塞或者退出时

触发一次调度，任务出队，特别的，这时候会更新最小时间min\_vruntime为当前队列的所有运行任务的最小时间或它本身两个值的较大者。挑选下一个运行任务非常简单，只需要取rbtree的最左节点。

1. 任务被唤醒时

先减去当前队列的最少运行时间min\_vruntime，因为后面入队列时会再加回去（可能不是同一个队列），保证任务都在同一起跑线上；  
 选择合适的队列，这个时间点是个负载均衡的好时机；加上目的队列的最少运行时间min\_vruntime, 更新目的队列的统计信息；入队列（实际就是插入红黑树rbtree的合适位置）。在支持hrtick的系统上，更新定时器超时值，控制调度时间的到来；检查是否可以抢占当前任务，若是，置调度标志。

1. tick中断到来时

更新队列与当前运行任务的统计（vruntime等），看是否需要激活定时器，检查是否可以切换当前运行任务（任务运行时间超过了时间片或大于调度间隔粒度或任务运行时间比小一个任务的运行时间差值大于时间片）。

1. **调度类的实现**

由于调度算法可能会更改，不同的任务有不同的调度算法（实时任务与非实时任务就不一样，任务本身也可以在实时或非实时切换），而调度流程或者调度处理的框架总体上变化不大，不外乎下面几种情况。

1. 任务创建时分配合适的CPU；
2. 任务退出或者睡眠时选择下一个任务运行；
3. 任务被唤醒时放到合适的位置等待调度；
4. 任务状态变化时（如优先级调整，从实时调度改成非实时调度）可能触发调度。

因此，内核引入了调度类的概念，保证调度算法的改变而内核调度处理流程的稳定。目前，内核有调度类的三个实现：实时任务调度、CFS调度、IDLE调度；在任务控制块里有一个指针 const struct sched\_class \*sched\_class;指向了这三个中的一个（依赖于当前任务的状态）。内核还有一个特别的调度类实现（stop\_sched\_class），属于这个类的任务为最高调度优先级，可以抢占其他任何任务，目前看只有迁移任务属于这一种。

1. **调度类的定义**

1. Struct sched class{ const struct sched\_class \*next};

--指向下一个调度类，调度时从rt->cfg->idle顺序往下挑选下一个可执行任务，如果没有，则通过此指针挑选下一类任务,从而保证了实时任务比非实时任务优先获得调度。

1. void (\*enqueue\_task) (struct rq \*rq, struct task\_struct \*p, int flags);

--任务p入队列，p已经就绪。

1. void (\*dequeue\_task) (struct rq \*rq, struct task\_struct \*p, int flags);

--任务p出队列，p被阻塞或者睡眠。

1. void (\*yield\_task) (struct rq \*rq);

--任务放弃CPU，对于rt任务会重新如队列触发调度，cfs会把任务放到rbtree的最右端，然后挑选最左边的任务运行。

1. void (\*check\_preempt\_curr) (struct rq \*rq, struct task\_struct \*p, int flags);

--检查p是否可抢占当前运行任务。

1. struct task\_struct \* (\*pick\_next\_task) (struct rq \*rq);  
    --挑选下一个可运行任务。
2. void (\*put\_prev\_task) (struct rq \*rq, struct task\_struct \*p);  
    --处理上一次运行的任务p。
3. #ifdef CONFIG\_SMP

int  (\*select\_task\_rq)(struct rq \*rq, struct task\_struct \*p,int sd\_flag, int flags);

--选择任务的运行队列，实际就是挑选合适的CPU运行任务。

1. void (\*pre\_schedule) (struct rq \*this\_rq, struct task\_struct \*task);

--调度前的处理，rt实现：如果本队列之前运行的任务为最高优先级，说明本队列没有高优先级任务抢占当前运行任务，则其他队列有可能存在比本队列的任务高的rt任务，尝试pull过来。

1. void (\*post\_schedule) (struct rq \*this\_rq);

--调度完成后的处理。rt实现，把就绪任务放入push队列

1. void (\*task\_waking) (struct rq \*this\_rq, struct task\_struct \*task);

--正在唤醒，准备唤醒任务时的处理。cfs实现，修改任务的vruntime。

1. void (\*task\_woken) (struct rq \*this\_rq, struct task\_struct \*task);

--唤醒任务后的处理。rt实现，如果没有置调度标志且任务可push，则尝试push到其他CPU上。

1. void (\*set\_cpus\_allowed)(struct task\_struct \*p,const struct cpumask \*newmask);

--设置任务的CPU亲和性在本调度类下特殊实现。用户接口、迁移时使用，rt实现，由set\_cpus\_allowed\_ptr调用，dequeue/enqueue可push队列。

1. void (\*rq\_online)(struct rq \*rq);  
    --任务队列状态active
2. void (\*rq\_offline)(struct rq \*rq);

--任务队列状态inactive。

1. void (\*set\_curr\_task) (struct rq \*rq);  
    --设置当前的运行任务，当任务被调度运行时。
2. void (\*task\_tick) (struct rq \*rq, struct task\_struct \*p, int queued);

--时间中断处理，更新任务运行时间，检查时间片是否已到。

1. void (\*task\_fork) (struct task\_struct \*p);  
    --任务p被创建时的处理，CFS有实现，把p入队列合适的位置
2. void (\*switched\_from) (struct rq \*this\_rq, struct task\_struct \*task,int running);  
   --任务切换出当前调度算法，当前调度算法需要做的动作。如rt任务切换成非实时任务，则rt class需要判断当前队列是否还有实时任务，若没有需要从其他队列pull任务过来。
3. void (\*switched\_to) (struct rq \*this\_rq, struct task\_struct \*task,int running);

--任务切换到当前调度算法。

1. void (\*prio\_changed) (struct rq \*this\_rq, struct task\_struct \*task,int oldprio, int running);  
    --任务更改优先级的处理。
2. unsigned int (\*get\_rr\_interval) (struct rq \*rq,  
   struct task\_struct \*task);  
    --获取任务时间片。