文章编号：1671-1742(2010)01-0018-04

完全公平调度算法分析

朱 旭，杨 斌，刘海涛
（西南交通大学信息科学与技术学院，四川 成都 610031）

摘要：Linux调度系统的调度算法几经改进，表现出优异的性能，特别是Linux最新的CFS调度算法。它的设计目的是使进程更加公平地共享处理器资源。在分析Linux 2.6.28内核代码的基础上详细阐述了CFS调度算法的工作流程和主要特性，并从算法分析和Hackbench测试两个方面对O(1)和CFS调度算法的性能进行了对比。

关 键 词：计算机科学与技术；计算机应用技术；完全公平调度算法；公平性
中图分类号：TP391 **文献标识码**：A

1 引言

进程调度是操作系统的核心功能，其主要目的是合理分配处理器资源。调度算法是任务调度具体的实现方法，一个好的调度算法将实现公平、效率、响应时间、周转时间、吞吐量等多个调度目标间的平衡。自Linux 2.4调度器使用基于优先级的调度算法以来，Linux2.6开发系列的内核中又经历了O(1)、RSDL、SD和CFS调度算法。其中CFS是Linux2.6.23内核引入的全新调度算法，相对于O(1)，CFS进行了很大改动，它以完全公平作为调度的核心思想，在性能提升的基础上大大简化了代码，成为内核采用的新一代调度算法。

2 CFS概述

CFS的总体设计可以用一句话来总结：在真实的硬件上模拟“理想的多任务处理器”，使每个进程都尽可能公平的获得CPU。为此，CFS引入了一个新概念“virtual runtime”，它描述了进程在CPU上的执行时间。在调度的过程中，CFS为了使每个进程都获得相近的执行时间，总是选取vruntime最小，也就是执行时间最短的进程来运行，以达到各个任务执行时间的平衡。这就是CFS的核心思想，即每个进程都被公平对待。

CFS调度算法相比于O(1)算法有了很大的变化：不再跟踪进程的睡眠时间、区分交互式进程，因此代码中没有那么多难以理解的经验性公式，思路清晰简单；新版本中增加了组调度功能，实现了对用户和组的公平性；引入了调度类sched_class和调度实体schedu_entity的概念——调度类使不同的进程选择不同的调度模块，调度实体则实现了组调度的功能；不再使用优先级数组，它将所有就绪态的进程都插入红黑树，用红黑树来选择下一个被调度的进程。图1描述了进程的调度模型。类似于以往的调度器，它的主要工作仍旧是在就绪态的进程队列中选择最合适的进程来运行，不同的是，新内核中有了调度类的概念，将不同的进程放入不同的调度模块中执行，例如：普通进程进入CFS调度模块，实时进程进入实时调度模块。因此，每个调度模块都要执行调度类为它指定的一组相应的函数。这样做的好处是，当需要修改相应进程的调度算法时，并不需要修改整个scheduler()函数，只需要修改对应的函数。

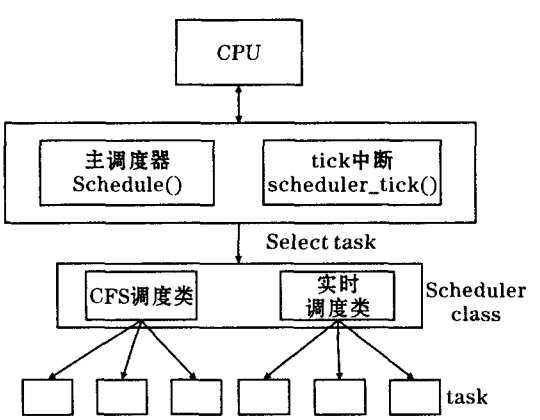

图1 CFS调度结构图

收稿日期：2009-06-29；修订日期：2009-09-09

3 CFS 算法实现

CFS 调度器的执行仍然以周期性调度函数 scheduler_tick()和主调度函数 scheduler()为基础,根据进程的优先级调整进程的执行时间、以公平为原则实现对进程的调度,合理分配 CPU 资源。

3.1 周期性调度函数 scheduler_tick()

系统时钟中断调用 scheduler_tick()函数,更新运行队列信息并执行与调度有关的系列操作。图 2 描述了该函数主要的处理流程。

(1)更新本地 CPU 运行队列的时间戳、负载等信息,然后转入 CFS 调度类的 task_tick 函数——task_tick_fair()。

(2)更新 CFS 运行队列与当前执行进程的相关信息,在 update_curr()中实现。此操作是中断处理函数的核心,包含以下几个重要步骤:

①计算当前运行进程自上次 tick 中断以来的运行时间 delta_exec:

delta_exec = (unsigned long)(now-curr->exec_start)

curr->exec_start 表示上次 tick 中断时设置的时间戳,now 表示本次 tick 发生时的时间戳。

②计算当前运行进程总的执行时间:

curr->sum_exec_runtime += delta_exec。

③根据进程的 nice 值对进程的运行时间 delta_exec 进行修正,获得进程的虚拟执行时间 vruntime。

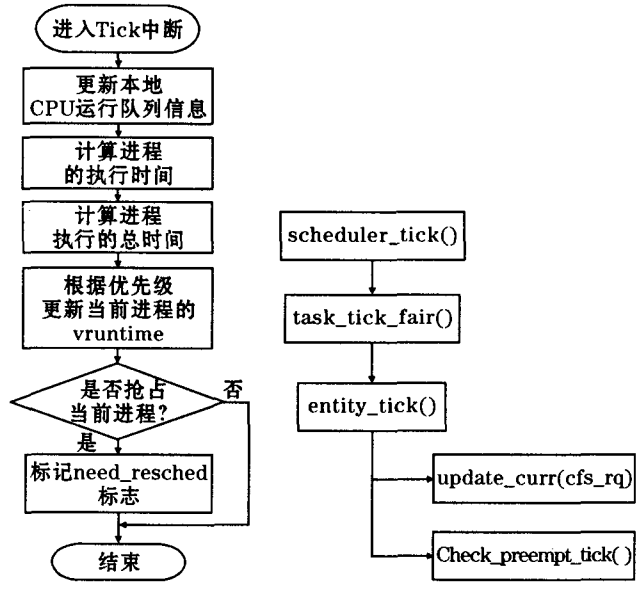

图 2　tick 中断流程图

由于所有就绪态进程都以 vruntime 为键值插入到红黑树中,所以 vruntime 越大,键值越大,从而使得当前进程随着执行时间的增加而向红黑树的右侧移动。在每个调度点,调度器都会优先选择 vruntime 小的进程——也就是红黑树中最左边的叶子结点进行调度。

tick 中断里 vruntime 的变化归纳为以下公式:

$$vruntime += delta\_exec * NICE\_0\_LOAD / curr->load.weight$$

NICE_0_LOAD 表示进程 nice 为 0 时的 weight 值。curr->load.weight 表示进程 nice 对应的 weight 值,nice 越低,值越大,那么在执行相同时间的条件下(delta_exec 相同),高优先级进程计算出的 vruntime 值会比低优先级进程的 vruntime 低。即高优级的进程就会位于红黑树的左边,下次调度的时候就会被优先调度。

④更新当前进程的执行时间戳:curr->exec_start = now。

(3)判断是否需要抢占当前进程,在 check_preempt_tick()中实现。

①计算 CFS 队列中所有进程被调度一次的时间周期 period;

②计算当前进程允许占用的时间 ideal_runtime;

$$ideal\_runtime = period * curr->load.weight / cfs\_rq->load$$

curr-> load.weight 表示进程 nice 对应的 weight 值,而 cfs_rq->load 表示该 cfs_rq 的负载,进程入列时,此值增加,进程出列时,此值减小。由以上公式可以看出,系统负载越高, ideal_runtime 越小;nice 越大, se->load.weight 值越小、ideal_runtime 越小。也就是说优先级越高,进程执行的时间片越长;系统负载越低,进程允许执行的时间片越长。

③计算进程已占用 CPU 的时间

$$delta\_exec = curr->sum\_exec\_runtime - curr->prev\_sum\_exec\_runtime$$

prev_sum_exec_runtime 表示进程被切换进 CPU 时的 sum_exec_runtime。

④比较进程已占用 CPU 的时间 delta_exec 是否大于 ideal_runtime。如果进程执行时间超过 ideal_runtime 时,就会调用 resched_task()设置该进程的抢占标志位,并在 tick 中断返回时调用 schedule()来完成调度过程。

3.2 主调度函数 scheduler()

scheduler()是真正实现调度的函数,它由主动和被动两种方式调用。scheduler()的主要功能是在运行队列中选出下一个被调度的进程,并更新运行队列的调度信息。图 3 描述了 schedule()函数的主要工作流程。它的工作集中在 put_prev_task() 和 pick_next_task()两个函数上。在 CFS 调度模块中,put_prev_task()对应的函数为 put_prev_task_fair(),pick_next_task()对应的函数为 pick_next_task_fair()。

(1)禁用内核抢占、初始化局部变量、执行相关锁操作及清除调度标志位等工作。

(2)将当前执行进程放回运行队列,在 put_prev_task_fair()中实现。

①更新 cfs_rq 和当前运行进程的信息 update_curr()。

②将当前进程插入红黑树,其排序的键值 key 为 se->vruntime-cfs_rq->min_vruntime (在_enqueue_entity()实现)。

(3)选出下一个被调度的进程,并将 CPU 分配给这个进程(在 pick_next_task_fair()中实现)。

①选择下一个要调度的进程(在 pick_next_entity()中实现)。

首先选出红黑树最左侧的结点 se,然后进行两次条件判断,最终决定下一个被调度的进程。

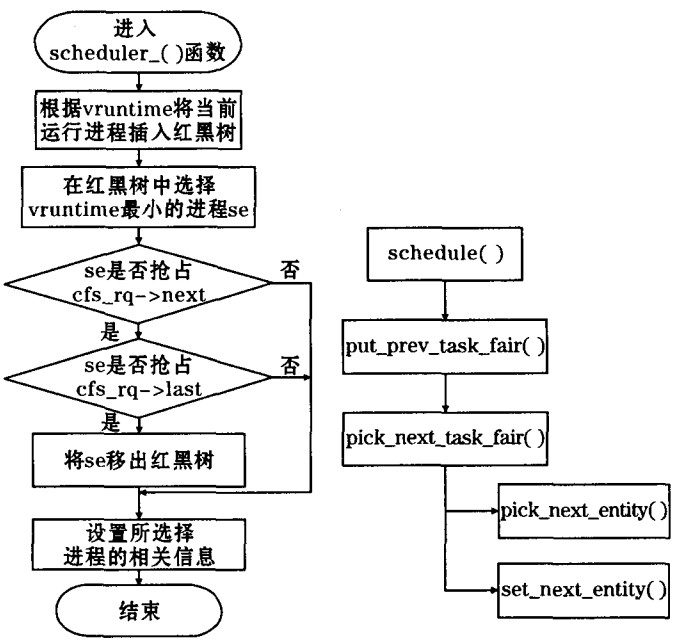

图 3 scheduler()函数流程图

判断 1:是否被 cfs_rq->next 抢占

cfs_rq->next 是被唤醒的进程。内核要优先调度被唤醒的进程,比如说 A 在等待一件事情,这件事情发生了,所以将 A 唤醒,当然是希望 A 尽快去处理这件事情比较好。

判断 2:是否被 cfs_rq->last 抢占

cfs_rq->last 是当前运行的进程。为了避免频繁调度,内核会优先调度唤醒时的当前进程。有进程唤醒时,此时的调度比较频繁,因为可能插入了一个较小的 vruntime 进程。

以上可以看出 schedule()调度的时候,会优先让这两个进程运行。当这两个判断条件都不满足时,才会调用红黑树最左侧的节点 se 运行。

那么怎样判断 se 是否被 cfs_rq->next 或 cfs_rq->last 抢占呢?(在 wakeup_preempt_entity()中实现)

当 se 满足以下两个条件时,se 不会被抢占:

条件一:se->vruntime 小于 curr->vruntime

条件二:curr->vruntime-se->vruntime 大于最小调度粒度

调度粒度,也就是进程理论上占有 CPU 的最短时间。因为一个机器触发调度的点很多,可能会在很短的时间内很频繁的检查当前进程需不需要被切换,考虑最小调度颗粒的问题可以避免频繁调度,浪费 CPU 资源。

②选出下一个被调度的进程后,用 set_next_entity()设置所选进程的相关信息。

将选择的下一个被调度进程 se 从红黑树中移出(调用_dequeue_entity()实现);

更新 se 的开始执行时间 se->exec_start;

更新 cfs_rq 上当前执行的进程为 se 所表示的进程 cfs_rq->curr=se;

更新 se->prev_sum_exec_runtime 做为进程切换到 CPU 上的 sum_exec_runtime,

se->prev_sum_exec_runtime = se->sum_exec_runtime。

4 CFS 与 O(1)算法性能比较

HackBench 是由 Rusty Russell 提出的一种 BenchMark 标准测试工具,用来评估 Linux 进程调度器的调度性能、负载性和可扩展性。该工具创建 N 组进程,每组进程包括 20 个写者和 20 个读者,写者通过管道向读者发送数据,测试 N 组进程管道读写的时间。N 值越大,调度器需要调度的任务就越多,这样就能反应出调度器的性能。在实验中创建了 $N(1-60)$组进程,分别在 Linux2.6.18(O(1)调度算法)和 Linux2.6.24(CFS 调度算法)两个版本上作测试。实验结果如图 4 所示。从图 4 可以看出,在 HackBench 测试程序上运行相同个数的进程时,Linux 2.6.24 的平均时间比 Linux2.6.18 要少,而且 Linux2.6.24 的曲线更接近于一条直线。测试结果表明,CFS 调度器比 O(1)调度器具有更好的性能。

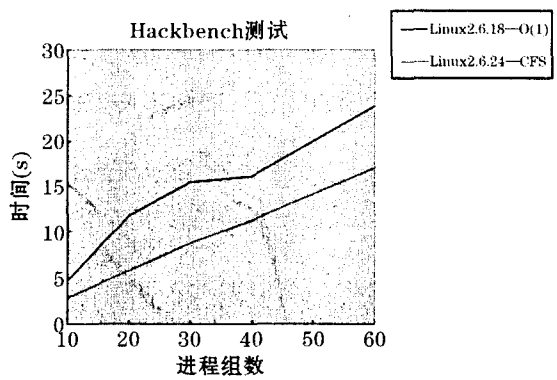

图 4　运行不同进程数的 HackBench 测试程序的平均时间

5 CFS 小结

以上的讨论看出 CFS 对以前的调度器进行了很大改动。以完全公平为核心思想,通过追踪进程的执行时间来调度任务;使用红黑树代替优先级数组来选择下一个被调度的进程;引入调度类,显著增强了内核调度程序的可扩展性和代码的可维护性;代码中不再有难以理解的经验性公式,思路清晰简单、结构灵活、算法适应性更高。当然,任何调度算法都还无法满足所有的应用需要,CFS 也有一些负面的测试报告,由于红黑树的查找执行时间为 $O(\lg N)$,当调度任务大幅增加时,性能会有所下降,但 CFS 在总体性能上还是比 O(1)调度算法有了显著的提高。随着 Linux 内核的不断发展,Linux 调度算法会进一步完善,新的调度算法更令人期待。

参考文献:

[1] 张桂兰,王飞超. Linux 内核完全公平调度器的分析及模拟[J]. 中国信息科技,2009,4:134-137.
[2] 高博. Linux 内核调度器分析及模拟[D]. 杭州:浙江大学,2008.
[3] Daniel P. Bovet, Marco Cesati. 深入理解 Linux 内核[M]. 南京:东南大学出版社,2006.
[4] Wolfgang Mauerer . Professional Linux Kernel Architecture[M]. Wiley Publishing Inc Indianapolis, Indiana, 2008.
[5] 刘谦. 基于 Linux 的调度机制及其实时性研究[D]. 成都:西南交通大学,2008.
[6] 苏新,毛万胜. 基于 Linux 2.6 内核进程调度策略分析[J]. 福建电脑,2007,(12).

An Analysis of CFS Scheduling Algorithm

ZHU Xu,　YANG Bin,　LIU Hai-tao
(School of Information Science & Technology, Southwest Jiaotong University, Chengdu 610031, China)

Abstract: With the improvement of scheduling algorithm, Linux scheduler shows excellent performance, especially the latest CFS scheduling algorithm. The aim of CFS scheduling algorithm is to share the processor more fairly. This paper illustrates the workflow and main characteristics of CFS based on the Linux 2.6.28 kernel source code. Finally, the performance comparison of O(1) and CFS are carried on from both the algorithm and the Hackbech test.

Key words: computer science and technology; computer application; CFS scheduling algorithm; fairness