Linux内核完全公平调度器改进的研究

朱永华[1],沈 熠[2],刘 玲[1]
ZHU Yonghua[1], SHEN Yi[2], LIU Ling[1]

1.上海大学 计算中心,上海 200444
2.上海大学 计算机工程与科学学院,上海 200444
1.Computing Center, Shanghai University, Shanghai 200444, China
2.School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China

ZHU Yonghua, SHEN Yi, LIU Ling. Research on improving Linux completely fair scheduler. Computer Engineering and Applications, 2014, 50(21):59-62.

Abstract: Fairness issue of the Completely Fair Scheduler (CFS) used in Linux kernel comes up due to the fact that programs with higher number of threads are favored by the scheduler, which are based on the number of thread in the system. A novel algorithm as well as its implementation through optimized procedure is proposed as a solution to achieve better fairness by punishing greedy-threaded programs. Several tests are conducted to illustrate fairness issue and to examine the effect of the proposed algorithm.

Key words: Linux kernel; process scheduling; complete fair scheduler

摘 要:针对现有Linux内核使用的完全公平调度器无法有效解决贪婪线程问题,提出一种改进的调度算法和该算法的高效实现,该算法通过惩罚贪婪线程的方法提升调度器的公平性。实验结果证实,贪婪线程问题存在;改进后的调度算法有效减少了存在贪婪线程问题的程序对降低系统整体性能的影响。

关键词: Linux内核;任务调度;完全公平调度
文献标志码:A **中图分类号:**TP312 doi:10.3778/j.issn.1002-8331.1211-0036

1 引言

随着Linux内核[1]的不断改进,功能的不断完善,其性能越来越强。其中,任务调度算法经历了许多版本的改进:Linux 2.4内核的 $O(n)$ 算法和Linux 2.6早期版本的 $O(1)$ 算法都是基于Linux早期版本的调度思想,但存在着代码结构复杂、进程饥饿、大量经验公式等问题[2-4];自Linux 2.6.23内核使用了新的完全公平调度器(Completely Fair Scheduler,CFS)[5-6],该调度器以公平思想为调度原则。

2 Linux调度器现状

2.1 CFS

CFS由Ingo Molnar提出,它从Con Kolivas的SD(Staircase Scheduler)与其改进版本RSDL(Rotating Staircase Deadline Scheduler)中吸取了完全公平的思想,将所有进程都统一对待,不再区别对待交互进程与普通进程。“80%的CFS算法的设计可以总结为一句话:CFS在真实硬件上模拟了一个‘理想的、精准的多任务CPU’”[7]。该调度器的思想是,两个性质、优先级、运算量相同的进程同时运行,其运行结束时间应该是近乎一致的。

Linux的非实时调度器是以优先级为计算基础的,设置了 *static_prio*,*normal_prio*,*prio* 三个优先级参数。由于CFS的对象是普通非实时的调度实体(sched_entity),故三者的值相等,均为 *static_prio* 。该参数由 *nice* 值给出, *nice* 值的值域[-20,19]映射到优先级数值区间

基金项目:国家高技术研究发展计划(863)重点项目(No.2009AA012201)。
作者简介:朱永华(1967—),男,博士,副教授,主要研究领域为高性能计算、通信与信息工程;沈熠(1989—),男,硕士研究生,主要研究领域为高性能计算与系统算法;刘玲(1977—),女,助理实验师。E-mail:shenyi0828@gmail.com
收稿日期:2012-11-05 **修回日期:**2013-01-25 **文章编号:**1002-8331(2014)21-0059-04
CNKI**网络优先出版:**2013-03-26, http://www.cnki.net/kcms/detail/11.2127.TP.20130326.1040.005.html

[100,139]。调度实体的重要性不仅由优先级指定,还需考虑该调度实体在就绪队列(CFS中使用红黑树[8])中的权重。调度实体的优先级影响其权重,其权重影响了每次累计的 *vruntime*,其在红黑树中位置由虚拟运行时间(*vruntime*)决定。通过sched.c中定义的prio_to_weight[]数组维护优先级至权重的转换关系。调度实体每提升一个优先级,则多获得10%的CPU时间。CFS通过平衡红黑树中每个调度实体的 *vruntime* 达到公平调度的目的。不同优先级实际运行时间与虚拟运行时间的比值是不同的,通过以下公式累积 *vruntime*:

$$vruntime += delta\_exec \times \frac{NICE\_0\_LOAD}{load.weight} \quad (1)$$

vruntime 越小,就越靠近红黑树的左侧,也就会被更早调度。 *nice* 值为0的虚拟运行时间与实际运行时间是相同的。

2.2 CFS的问题

虽然完全公平的调度策略思想先进,且在大部分的实际情况下达到了公平调度的目的,但存在一些额外情况。如用户可以通过fork()调用创建多个子进程,这样可以使自己任务得到更多的CPU时间已达到更快处理的目的,CFS为了解决该问题,引入了组调度。

假设用户A有2个进程,用户B有8个进程。若此时调度粒度为进程,那么调度结果会对用户A不公平。组调度的目的就是让用户A与B各自得到50%的CPU资源。通过调用Sched_autogroup.c中定义的系统调用,将多个进程打包为组,达到公平的目的。

类似的问题会出现在多线程的程序调度中,用户仍可以通过创建多个线程获得更多的CPU时间。假设某情形(例1):有三个进程A,B,C,分别拥有1,2,2个线程,此时CPU的资源分配情况如图1所示。其中 *period* 为调度延迟值,即每个可运行的调度实体至少运行一次的时间。如图1所示,似乎CFS公平地为三个进程分配CPU资源,然而换一种情形(例2):同样有三个进程A,B,C,分别拥有1,1,8个线程,此时CPU的资源分配情况如图2所示。

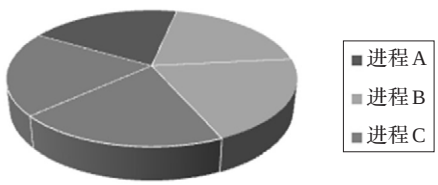

图1 例1中CPU资源分配图

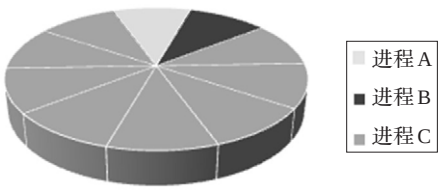

图2 例2中CPU资源分配图

如图2所示,进程A,B的运行时间大大被压缩,系统大部分计算资源被分配给进程C,这样就违背了公平原则。

3 改进算法研究

3.1 现有改进算法及其不足

已有提出并实现了一种解决方案PFS(Process Fair Scheduler)[9],该方案借用组调度的思想,将调度粒度提升至进程,即对于之前的例子将如下分配CPU时间。其权重计算公式为:

$$se \rightarrow load.weight' = \frac{se \rightarrow load.weight}{\alpha} \quad (2)$$

其中 α 为进程的线程数。根据新的权重计算方式,在例2描述的情形中CPU的资源分配情况如图3所示。

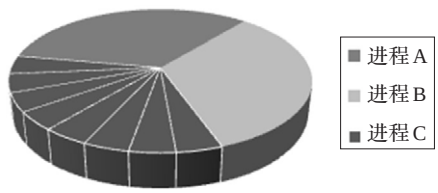

图3 例2中使用PFS调度策略后CPU资源分配图

如图3所示,进程A,B,C之间被“公平”地对待,各自享有均等的计算资源。但上述算法也并非完全合理,有以下两个问题:

(1)经过合理多线程优化(well-threaded)的程序并没有得到合理的对待,它与其单线程版本程序相比并没有得到应有的优待,相反会因为线程切换带来的性能开销而造成其运行效率反不如单线程版本程序。

(2)若不友好(greedy-threaded)的进程尝试创建多个线程以获得CPU运行时间,那么其切换频率会非常频繁,造成大量CPU和I/O开销,最终导致调度器选择下一个调度实体的次数增多,严重影响整体性能。

3.2 改进策略分析

现有Linux内核CFS的调度粒度为线程,PFS的调度粒度为进程。所要寻找的调度算法其粒度应设定为两者之间,通过对拥有不同线程数的程序的线程权重调节,达到算法改进的目的。改进后的算法需要满足以下几点:

(1)避免复杂计算。调度器的运行频率为毫秒级,若其本身就需要占有系统很大一部分的计算资源,那将本末倒置,系统整体性能也将下降。CFS很好地避免了检测交互式进程与“惩罚”非交互式进程所需要的大量的启发式计算(由__normal_prio()完成),不应引入过于复杂的公式计算。

(2)优待多线程程序。通过将串行算法改变为并行算法以达到提升性能的程序应优先得到计算资源。在文献[2]中提出的PFS并没有给多线程应用应有的优待,

然而应给予并行算法优待。但需要在一定线程数限制下,提高其运行权重。

(3)顾全大局。在多线程程序优先获得计算资源的同时,不能因为过于优先以至于抢占了其他程序应获得的计算资源。对于贪婪地创建线程的程序,应对其进行"惩罚",减少运行权重。

3.3 DW-CFS(Dynamic Weight Complete Fair Scheduler)

为了描述该算法,需要对以下概念进行定义:

定义1(在线核心数) 当前工作的处理器核心数,即可用逻辑计算核心数(用 N_o 表示)。一般情况下系统的计算核心数从内核启动后不发生变化;但若开启热插拔后,在线核心数(online_cpu)可以发生变化。

定义2(程序线程数) 当前调度实体共享内存空间的线程数(用 N_t 表示)。一个程序的线程数是设定线程初始权重的依据,创建一定数据量的线程会提升程序的总优先级,但过度数量的线程则会被"惩罚"。每当新的线程被创建时,通过对copy_signal()函数中signal结构体的count累加来计数,并更新nr_thread的计数,实现记录线程数的目的。

为了实现改进,现提出DW-CFS算法。该改进算法改变了调度实体的权重计算方式,公式如下:

$$\begin{cases} se \rightarrow load.weight' = \alpha \times se \rightarrow load.weight \\ \alpha = f\left(N_t, N_o, \beta\right) \end{cases} \tag{3}$$

其中权重调节因子 β 是一个常数,用以表示可获得额外权重的最大值,改进后的权重 W' 与三个参数需要满足一下条件:

(1)当 $N_t \leqslant N_o$ 时,若 N_t 越大,则 W' 越大,反之越小。

(2)当 $N_t > N_o$ 时,若 N_t 越大,则 W' 越小,反之越大。

(3)当 $N_t = 1$ 时,$W' = W$ 。

在DW-CFS算法中,原有累计 *vruntime* 的公式(1)变为如下公式:

$$vruntime += delta\_exec \times \frac{NICE\_0\_LOAD}{\alpha \times load.weight} \tag{4}$$

满足以上条件的 α 函数有很多,本文提出一个计算量小,与实际需求拟合度较高的计算公式:

$$f\left(N_t, N_0, 1.1\right) = -\left|\frac{1.2 \times \left[1 + \left(11 \times N_o - 12\right)\right]}{N_t + \left(11 \times N_o - 12\right)} - 1.1\right| + 1.1 \tag{5}$$

其中 β 已用一个确定的值替代,其值表示当 $N_t = N_o$ 时,该线程可获得10%权重提升。该函数以 N_o 为分界,当 $N_t \leqslant N_o$ 时,函数单调递增;当 $N_t > N_o$ 时,函数单调递减,且函数值收敛于0。当 $N_o = 4$ 时该函数曲线如图4所示。

为了优化公式计算,以减少调度器在计算权重时的开销,该公式的实现算法可由如下伪代码表示。

```
#define INV_SIGN_BIT 0x 7ffffff
var a=(((n0>>1)+n0)>>2) – n0 – 12;
var down=nt+(++a);
var up=a*1.2;
var before_abs=up/down – 1.1;
var after_abs=before_abs & INV_SIGN_BIT;
return 1.1 – after_abs;
```

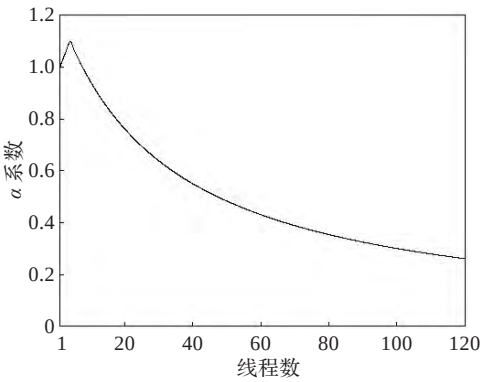

图4　权重调节函数 ($N_o = 4, \beta = 1.1$)

4　实验结果及分析

4.1　实验环境

实验平台说明如表1所示。

表1　实验平台配置

Processor	Intel® Core™ i5-520M@2.40 GHz
Memory size	2×2 GB DDR3@1066 7-7-7-20
OS/kernel	Ubuntu x86_64/Linux 2.6.38.11

4.2　实验方法

相比于使用仿真软件[10],真实平台所能反映的结果更准确。本实验使用圆周率Π的计算[11]为测试程序,其算法复杂度为 $O(n)$, n 为精度。实验中同时启动两个进程,进程A为单线程版本的算法实现;进程B为多线程版本的算法实现,线程数为 m, 其中 $m \in \{1, 2, 3, 4, 5, 6, 7, 8, 16, 32\}$ 。根据公平原则,两者的结束时间应接近一致。其中单线程版本程序A与多线程版本程序B同时运行的流程图如图5所示。

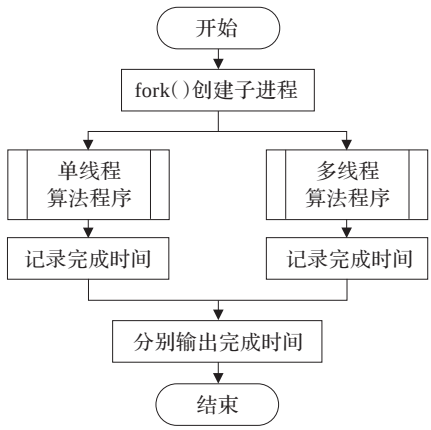

图5　测试程序流程图

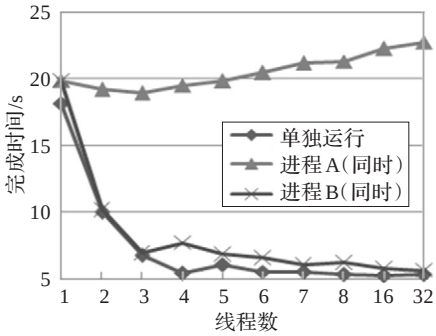

图6(a) 改进前测试结果

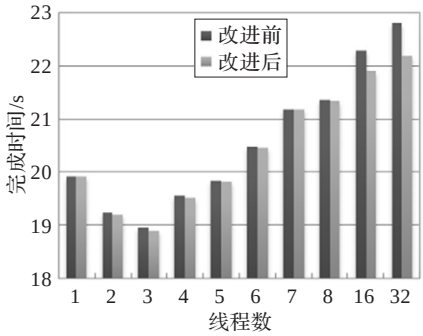

图6(b) 进程A(单线程)改进前后对比

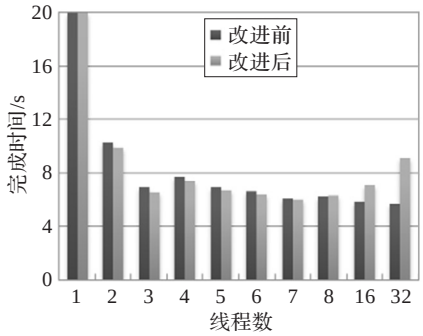

图6(c) 进程B(多线程)改进前后对比

4.3 实验结果及分析

图6(a)与图6(b)描述了改进前后单线程与多线程算法在不同线程数下的运行情况。

根据图6所示实验结果可得出以下结论:

(1)由图6(a)中进程B的运行结果得知,多线程任务确实可以减少运行时间。由于测试使用平台为双核四线程,当程序达到四线程后,进程B运行时间减少效果开始不明显,但运行时间仍然继续减少,其原因为该进程由于线程数量的优势增大了该进程被调度的整体几率,即抢占了系统其他程序运行机会,证实了CFS存在调度不公平的问题。

(2)由图6(a)中进程A运行结果得知,单线程版本的程序的运行时间随着同时运行的多线程版本程序的线程数的增加而增加,表明过多的线程抢占了系统其他程序的运行机会从而使单线程版本的程序运行时间增加,进一步证实了CFS存在调度不公平的问题;进程A在与其同时运行的进程B达到四线程前仍能减少运行时间,是因为CPU使用了Turbo Boost特性,可以提升单任务的执行效率。

(3)由图6(b)与图6(c)中运行结果(线程数[1,3]区间)得知,在进程A与进程B线程数总和达到最优线程数(实验中 $N_o=4$)之前,进程B因为得到更高的权重,其 *vruntime* 累计得更少,更靠近红黑树的左侧,更快被调度,最终比CFS改进前更快完成。同时进程A由于进程B的更快完成,受益于CPU的Turbo Boost特性,也降低了运行时间。

(4)由图6(b)与图6(c)中运行结果(线程数[4,7]区间)得知,由于进程A与进程B线程数总和超出最优线程数,需要更多的调度,使其运行时间均有增长。在此区间范围内,进程B的"优待"减少($N_t=8$ 时, $\alpha=0.99$,开始"惩罚"该进程),获得相对于其在 $N_t=4$ 时较少的调度机会。如图6(c)所示,进程B在线程数超过 N_o 值后的提速效果较区间[1,3]中的效果小。

(5)由图6(b)与图6(c)中运行结果(线程数[8,32]区间)得知,由于对进程B过多线程数的"惩罚"力度不断增大,其执行时间比CFS改进前不断增大;同时进程A由于能够被更快地调度,其运行时间也相比CFS改进前有所减少。DW-CFS改进效果得以说明。但是因为线程切换开销和无法使用Turbo Boost特性,改进效果被一定程度地抵消。

5 结束语

本文从Linux内核的CFS研究出发,讨论了完全公平调度策略与该调度器的不足,介绍了研究改进的现状。在此基础上提出了DW-CFS,该算法基于对程序线程数的检测,通过程序线程数与在线核心数的比较,调整其权重,改变调度结果,使得良好优化的多线程程序得到调度的优先;同时避免了程序利用CFS在贪婪线程问题上的处理不足,抢占系统大量资源。实验表明DW-CFS有效减少贪婪线程对降低系统整体性能影响。

参考文献:

[1] The Linux kernel archives[EB/OL].[2012-03-15].http://www.kernel.org/.

[2] Mauerer W.Professional Linux kernel architecture[M].[S.l.]:Wiley Publishing,Inc,2008.

[3] Daniel P B,Marco C.Understanding the Linux kernel[M].[S.l.]:O'Reilly Press,2005.

[4] Love R.Linux kernel development[M].[S.l.]:Noval Press,2010.

[5] Molnar I.Modular scheduler core and completely fair scheduler[EB/OL].(2007-05-11).http://lwn.net/Articles/230501/.

[6] Molnar I.CFS updates[EB/OL].[2012-04-10].http://kernel-trap.org/Linux/CFS_Updates/.

[7] Andrew J.Interview:Ingo Molnar[EB/OL].[2012-04-17].http://kerneltrap.org/?q=node/517.

[8] Thomas H C.Introduction to algorithms[M].[S.l.]:The MIT Press,2009.

[9] Chee S W,Ian T,Rosalind D K,et al.Towards achieving fairness in the Linux scheduler[J].Operating Systems Review(ACM),2008,42(5):34-43.

[10] John M C,Dan P B,Tong L,et al.LinSched:the Linux scheduler simulator[C]//ISCA PDCCS,2008:171-176.

[11] Pi[EB/OL].[2012-04-20].http://en.wikipedia.org/wiki/Pi.