Linux 调度器免锁优化方法研究

张 旭[1,2,3],顾乃杰[1,2,3],苏俊杰[1,2,3]

[1](中国科学技术大学 计算机科学技术学院,合肥 230027)
[2](中国科学技术大学 中科院沈阳计算技术研究所 网络与通信联合实验室,合肥 230027)
[3](中国科学技术大学 先进技术研究院,合肥 230027)
E-mail: jessexz@mail.ustc.edu.cn

摘 要: Linux 操作系统被广泛用于各领域,多核环境下 Linux 调度器依靠自旋锁保证其正确运行,这给调度器带来了严重的锁竞争. 在分析 Linux 调度器的基础上,对其提出三个层次的免锁优化方法: 基础优化、调度行为优化和基于上层应用特征的参数调优. 基础优化尝试从代码层面直观地缩小程序的锁冲突域; 调度行为优化针对进程创建过程中的唤醒操作提出了一种新进程延迟唤醒方法,有效地减少了进程创建过程中的锁竞争; 基于上层应用特征的参数调优可以在对内核修改很小的情况下完成调度器性能提升.

关 键 词: Linux 内核; 调度器; 自旋锁; 锁竞争
中图分类号: TP311 **文献标识码:** A **文 章 编 号:** 1000-1220(2017)04-0690-06

Research on Decreasing Lock Contention for Linux Scheduler

ZHANG Xu[1,2,3], GU Nai-jie[1,2,3], SU Jun-jie[1,2,3]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](USTC &SICT Network and Communication Joint Laboratory, Hefei 230027, China)
[3](Institute of Advanced Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Linux operating system is widely used in different areas. Linux scheduler employs spinlocks to guarantee its correctness on multicore processors, but spinlocks lead to severe lock contention. This paper analyses the characteristics of spinlock operations in Linux scheduler and proposes three different optimization strategies on three different levels to decrease lock contention. The first strategy is to shrink the critical sections from code level; the second strategy changes the behavior of scheduler, and introduces a new method, which delays the wakeup of new processes and considerably decreases the lock contention during process creation; the third strategy is tuning the key parameters of Linux scheduler according to the process features of top-level applications and it achieves reasonable performance promotion with minor modifications to the kernel.

Key words: linux kernel; scheduler; spin lock; lock contention

1 引 言

随着多核处理器的普及,Linux 操作系统在原来的调度器基础上增加了对多核处理器的支持,提出了调度域、负载均衡[1]等概念,用以提升系统在多核处理器上的性能. Linux 使用 CFS(Completely Fair Scheduler)调度器[2],每一个处理器核都有属于自己的调度信息和一个进程运行队列. 在多核处理器上,调度并不仅仅在单个处理器核中进行,而是在多个核中异步地推进. 如果每个处理器只能访问自己的调度信息和运行队列,而不知道全局的负载情况,难免会造成全局负载不平衡,进而导致处理器计算资源的浪费. 内核中负载均衡模块就是用于平衡不同处理器核的工作负载的,但是这样需要破坏调度信息的局部性,允许不同的处理器核访问其他处理器核的资源. 这种情况下,调度器中关键资源的访问冲突问题变得异常突出,而最核心的竞争资源就是进程运行队列. 目前 Linux 内核主要通过使用自旋锁(spinlock)[2]来控制运行队列的并发访问. 自旋锁语义直观,且使用方法简单,有利于程序开发人员使用. 虽然自旋锁能够很好地对临界资源进行保护,但是 CPU 处于自旋状态时实际上是在做"无用功",所以大量的锁竞争将严重影响调度器的执行效率和系统性能.

免锁优化指在不改变程序正确性的情况下,通过代码调整或算法改进减少程序中出现的锁竞争. 目前针对 Linux 调度器的研究主要是对调度算法或者负载均衡策略的改进[1,3,4],近年来则侧重于针对特定应用场景的调度算法的设计或改进,例如,旨在提升嵌入式系统电源使用效率的基于能耗的公平队列[5],以及用以实现自动化散热管理的基于热量自感知的调度算法[6]. 针对调度器的免锁优化研究还处于起步阶段,内核中常见的做法是通过细化锁粒度来提升系统的可伸缩行,即对于一个数据结构,需要标记其中各个独立的部分,使用多个锁来保护结构成员[7]. 这种方法可以一定程度

收稿日期: 2016-01-29 收修改稿日期: 2016-03-16 基金项目: 安徽大学青年科研基金项目(KJQN1118)资助. 作者简介: 张 旭,男,1990 年生,博士研究生,研究方向为操作系统、操作系统虚拟化、计算机网络; 顾乃杰(通信作者),男,1961 年生,博士,教授,博士生导师,研究方向为并行分布式算法、并行体系结构、并行分布式计算中的通讯问题等; 苏俊杰,男,1991 年生,硕士研究生,研究方向为操作系统、操作系统虚拟化.

提升系统性能,但是锁个数的增加造成锁操作开销增大,且多个锁的获取和释放顺序需要保证. 与此同时,相关研究人员还提出了一些对自旋锁的改进方案,以适用不同的应用场景,如顺序锁[7]、RCU(Read-Copy-Update)机制[8]等. 顺序锁为写者赋予了较高的优先级,即在有读者访问资源的时候依然允许写者继续进行. 与顺序锁相似,RCU 机制也是应用于读多写少的情况. RCU 技术的核心是写操作分为写和更新两步,允许读操作在任何时候无阻碍的运行. 考虑到原子操作的高效性[9],基于 CAS(Compare and Swap)原语[9]的无锁编程技术[10]也是时下的研究热点,不过无锁编程技术需要解决垃圾回收[11]和 ABA 问题[9],为系统带来额外开销,而且基于 CAS 设计的无锁数据结构通常应用场景较小,使用较为困难.

本文在分析 Linux 调度器的基础上,对其提出三种不同层次的免锁优化方法: 基础优化、调度行为优化、基于上层应用特征的参数调优. 其中基础优化是代码级的、以直观地缩小锁冲突域为目标的优化,不影响调度器本身的行为; 调度行为优化针对进程创建过程中的唤醒操作,提出了一种新进程延迟唤醒方法,该方法将不能被立即唤醒的新进程放入延迟队列中,并为延迟队列创造"一个生产者一个消费者"的访问环境,并在此基础上通过循环数组和链表实现队列的免锁,避免了不必要的锁竞争; 基于上层应用特征的参数调优以 Web 服务器为研究对象,根据 Web 服务器中服务进程的生命周期特征,调整调度器核心参数,达到减少锁操作的目的. 实验表明,三种层次的优化都能不同程度地提升 Linux 调度器性能,其中调度行为优化效果较为明显,而基于上层应用特征的参数调优可以在对内核修改很小的情况下,提升应用程序的服务质量.

本文剩余部分组织如下: 第 2 节对 Linux 调度器中的锁函数和锁竞争进行介绍和分析; 第 3 节介绍基础优化方法; 第 4 节阐述对进程创建过程中唤醒操作的改进方法; 第 5 节讨论基于上层应用程序特征的免锁优化; 第 6 节通过实验分析各个层次免锁优化的实际效果; 第 7 节总结全文. 需要说明: 本文余下部分提到的锁若非特别指明,均指自旋锁; 处理器核均指逻辑处理器核.

2　内核调度器锁竞争分析

Linux 内核版本更新较快,但调度器和负载均衡模块变化较小. 本节以两个长期支持版的内核 Linux 2.6.32 和 Linux 3.4 为基础分析调度器中使用的锁函数和锁操作特征.

2.1　锁函数

Linux 中以自旋锁为基础针对进程运行队列实现了各种锁函数,完成不同的功能需求. 函数接口和功能描述如表 1 所示.

2.2　调度器锁操作特征

Linux 内核引入负载均衡模块打破了调度信息和运行队列的局部性,使不同的处理器核可以访问其他处理器核的信息. 实际上,调度器中的锁操作也正是集中在与负载均衡相关的函数中. 在内核中负载均衡模块有四个执行时机:

a) 每个时钟节拍到来时,内核会执行时钟中断处理函数,该函数会判断是否已经到了负载均衡的执行时间点. 如果到了,则通过唤醒软中断的方式,使周期性负载均衡入口函数 load_balance() 得以执行.

表 1　锁函数接口及描述

Table 1　Descriptions of spinlock interfaces

函数接口	功能描述
void spin_lock(spinlock_t * lock)	自旋锁基础函数. 获取自旋锁; 在调度器中该函数的参数通常是运行队列结构体 struct rq 的成员变量 lock 的指针.
struct rq * task_rq_lock(struct task_struct * p, unsigned long * flags)	禁止本地中断,对进程 p 所在处理器核的运行队列加锁,返回该运行队列,并且通过指针返回中断状态字.
void double_rq_lock(struct rq * rq1, struct rq * rq2)	加双锁函数. 按地址从小到大的顺序对 rq1 和 rq2 加锁. 该锁函数用于进程迁移,使用条件为本地中断已被禁止.
int double_lock_balance(struct rq * this_rq, struct rq * busiest)	与 double_rq_lock() 类似. 使用条件为本地中断已被禁止且 this_rq 已经被加锁而 busiest 没有加锁. 最终结果为两个队列都被加锁.

b) 当本地运行队列空时,调度器会从最繁忙的处理器核上迁移进程到本地. 具体地说,调度器主函数 schedule() 会调用 idle_balance() 函数判断本地处理器核是否空闲,进而调用 load_balance() 或 load_balance_newidle() 函数完成相应迁移操作.

表 2　调度器核心函数中锁操作描述

Table 2　Descriptions of lock operations in scheduler

函　数	锁操作描述
schedule()	调度器主函数. 在函数开始时加锁. 如果未发生进程切换,则由原进程解锁; 否则,由切换后运行的进程解锁.
load_balance() / load_balance_newidle()	周期性负载均衡入口函数和空闲时负载均衡函数. 在完成进程迁移操作时调用 double_rq_lock() 对本地队列和目标队列同时加锁. (load_balance_newidle() 在 Linux 3.4 中被移除,功能由 load_balance() 实现)
try_to_wake_up()	用于唤醒睡眠进程,将进程加入新的运行队列上时对该队列加锁.
wake_up_new_task()	用于唤醒新进程,将新进程插入运行队列,函数开始和结尾分别对目标队列加锁和解锁.

c) 唤醒进程时,调度器调用 try_to_wake_up() 函数选择合适的处理器核,并将该进程插入该处理器核的运行队列中.

d) 创建进程时,调度器会选择负载最轻的处理器核,然后调用 wake_up_new_task() 函数将新进程插入其运行队列中.

相应地,内核调度器的加锁操作也主要集中在 schedule()、load_balance()、try_to_wake_up()、wake_up_new_task() 等函数中,其锁操作特征如表 2 所示.

此外,调度器为其他模块提供的接口函数,如用于计算进

程运行时间的 task_sched_runtime( ) 函数等,也会对进程运行队列加锁. 然而,这些函数获得和释放自旋锁的时间间隔很短,对系统的性能影响较小,不在本文的考虑范围内.

3 基础优化

减小自旋锁对程序性能的影响,最直观的基础方法就是缩小锁冲突域. 冲突域是获得锁和释放锁之间进程完成的操作,也称锁区域或临界区. 冲突域越小,进程持有一把锁的时间就越小,可能发生的锁竞争数量也就越少,其他进程因为锁冲突而等待的时间就越小. 缩小冲突域最直接的方法就是把原本放在冲突域内的代码移动到冲突域之外. 但对于操作系统这样的事务复杂的系统,这样简单的修改并不容易,需要对冲突域中的代码行为、逻辑关系及隐含的并行信息有充分的理解.

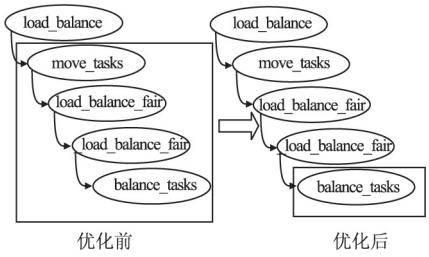

图 1 load_balance( ) 锁冲突域缩小示意图

Fig. 1 Shrink critical section for load_balance( )

如 2.2 节所述,在 Linux 2.6.32 中,load_balance( ) 函数和 load_balance_newidle( ) 函数在进程迁移时有一段加双锁的代码区域,即在调用 move_tasks( ) 函数时对本地队列和目标队列同时加锁,可以使用缩小冲突域的方法对这两个函数进行优化. 其中对 load_balance( ) 的优化如图 1 所示,箭头表示调用关系,方框中显示的是冲突域. 由图可知,move_tasks( ) 函数的函数调用链较长,这使得该处的锁冲突域较大. 经过分析,在 move_tasks( ) 的调用路径上,只有 balance_tasks( ) 函数对两个队列进行实际的操作. 但在 move_tasks( ) 函数的调用路径上还会调用其他相关函数完成对运行队列负载的更新,但是考虑到微小的负载误差不会给 CFS 调度算法的准确性带来太大的影响,所以可以只对函数 balance_tasks( ) 加锁. 针对 load_balance_newidle( ) 函数的优化与之类似,在此不做赘述.

可以看出,这种基础优化方法在本质上要权衡缩小冲突域给调度器的准确性和并行度带来的改变. 该方法虽然能够一定程度提升系统的并行性,但会影响系统调度的准确性. 此外,经过内核多个版本的优化后,Linux 内核对自旋锁的使用已经比较成熟,这种直观的方法并不能大量使用.

4 调度行为优化

进程被创建后,调度器会选择负载最轻的处理器核,即目标处理器核,然后调用 wake_up_new_task( ) 函数将该进程插入到该处理器核的运行队列中,完成进程的唤醒操作. 该函数会请求目标队列的自旋锁,获得锁后将进程插入该队列,完成后释放锁. 这里很容易出现锁竞争,即除已获得锁的处理器核外,其他请求该队列锁的处理器核都要忙等. 对于进程创建较多的应用来说,这样的锁竞争将造成巨大的处理器时间消耗,影响系统性能. 本文提出一种新进程延迟唤醒方法,对进程创建过程进行免锁. 该方法的核心思路是使用“尝试加锁”的方式请求目标处理器核的队列锁:

1) 如果尝试成功则按原有方式完成进程的唤醒操作;

2) 如果尝试失败则将新进程插入到本地处理器的一个独立的延迟队列中,由目标处理器核在下一次调用 schedule( ) 函数完成进程切换之前,将延迟队列中的进程插入到它自己的运行队列中唤醒,进而保证进程的正常调度运行.

算法 1. 对进程 p 所在的运行队列尝试加锁

```
1   task_rq_trylock( p)
2   {
3     for ( ; ; ) {
4       rq←task_rq( p)
5       if( raw_spin_trylock( rq^. lock) )
6         if( rq = task_rq( p) )
7           return SUCCESS
8         else raw_spin_unlock( rqq^. lock) )
9       else return FAIL
10    }
11  }
```

图 2 算法 1: 对进程 p 所在的运行队列尝试加锁

Fig. 2 Algorithm 1: trylock the runqueue of process p

对运行队列“尝试加锁”的语义如图 2 算法 1 所示. 其中,raw_spin_trylock( ) 尝试获取锁,若成功则返回 1,若失败则直接返回 0,而不忙等; raw_spin_unlock( ) 释放锁; task_rq(p) 用于读取进程 p 的目标处理器核的运行队列.

4.1 延迟队列

新进程延迟唤醒方法的核心数据结构是延迟队列,为此本文在为每个处理器核的管理单元中添加了三个成员变量:

a) 延迟队列数组 dtqueue [NR_CPUS],NR_CPUS 为处理器核总数,即每个处理器核上都有 NR_CPUS 个延迟队列,用于存放将向不同处理器核的运行队列中插入的延迟进程.

b) 位映射 dtmap,有 NR_CPUS 个比特位,用于标记哪些处理器核的延迟队列中有本应放到本地处理器核上的延迟进程.

c) 自旋锁 dtlock,用于控制 dtmap 的并发写.

下页图 3 以四核处理器为例介绍延迟队列的使用. 如图所示,每个处理器都有四个延迟队列,从上到下标号 0 至 3; 四个比特位,从右至左 0 至 3. CPU0 的 1 号延迟队列非空,即 CPU0 有目标为 CPU1 的延迟进程; 相应地,CPU1 的第 0 位被置位. CPU1 的 2 号延迟队列非空,即 CPU1 有目标为 CPU2 的延迟进程; 相应地,CPU2 的第 1 位被置位. CPU2 的 1 号延迟队列非空,即 CPU2 有目标为 CPU1 的延迟进程; 相应地,CPU1 的第 2 位被置位. CPU3 的 0 号延迟队列非空,即 CPU3 有目标为 CPU0 的延迟队列; 相应地,CPU0 的第 3 位被置位.

函数 schedule( ) 和函数 wake_up_new_task( ) 都是在禁止本地中断的情况下执行的,这种情况下函数不会被重入且内

核抢占不会出现. 即这两个函数的执行不会被打断,因此该算法中每一个的延迟队列都有且只有一个生产者,即本地处理器核,和一个消费者,即目标处理器核. 延迟队列都在“一个生产者一个消费者”的访问环境中,这种情况下使用循环数组实现延迟队列,可以在不加锁的情况下实现数据的并发访问.

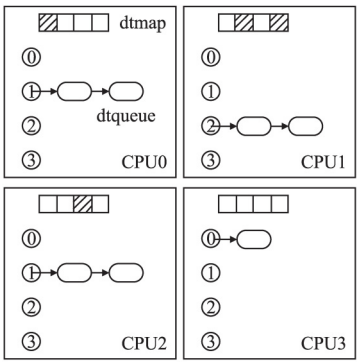

图 3　四核处理器延迟队列示意图

Fig. 3　Queues of delayed tasks on a quad core processor

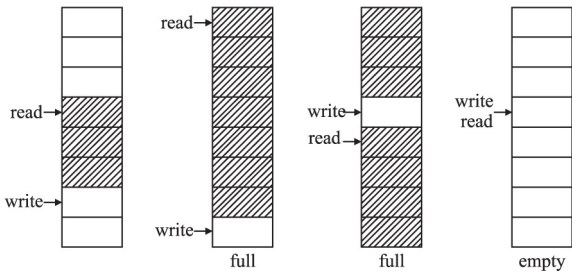

图 4　免锁循环数组示意图

Fig. 4　Ring buffer without lock

循环数组的操作示意图如图 4 所示,算法如图 5 算法 2 所示,其中 LENGTH 表示数组长度,在实验中被设置为 32. 生产者只需要操作 write,而消费者只需要操作 read,即可完成对循环数组的安全访问. 循环数组操作简单,然而数组难以扩展,如果遇到被延迟新进程规模较大的情况则会影响调度器性能. 文本使用循环数组结合链表的方式实现延迟队列,以提升方法的扩展性. 在数组未满时,优先插入数组,否则插入链表. 然而链表涉及指针,使用常规的链表需要通过加锁来避免空指针. 生产者通过 tail 指针向链表结尾添加节点时,需要对链表加锁,保证添加的同时消费者没有将链表清空而使 tail 成为空指针.

算法 2. 免锁循环数组

```
1   BufferDequeue( )
2       if ( IsBufferEmpty( ) )
3           return NULL
4       tmp←buffer [read]
5       read←( read + 1) % LENGTH
6       return tmp
7   BufferEnqueue( x)
8       if ( ! IsBufferFull( ) )
9           buffer [write] ← x
10          write ← ( write + 1) % LENGTH
11  IsBufferFull( )
12      return ( read = ( write + 1) % LENGTH)
13  IsBufferEmpty( )
14      return ( read = write)
15  BufferInit( )
16      read←write←0
```

图 5　算法 2: 免锁循环数组

Fig. 5　Algorithm 2: ring buffer without lock

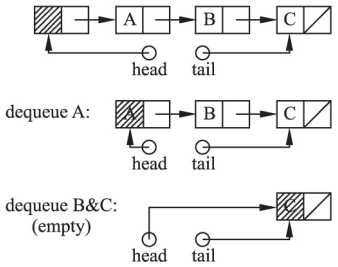

图 6　免锁链表示意图

Fig. 6　Linked list without lock

本文对链表的组织结构进行了改进,在“一个生产者一个消费者”的访问环境中,保证指针操作在无锁条件下的安全性. 改进后的链表结构如图 6 所示,具体的算法设计见下页图 7 算法 3. 在这个链表中,head 不是指向链表的第一个节点而是指向一个伪节点,该节点是最后一个出队但未被删除的节点; 而每一次出队操作都会删除上一次出队的节点(ListDequeue( )) . 在链表初始化时会分配一个节点作为第一个伪节点(ListInit( )) . head 和 tail 相同时,即同时指向一个伪节点时,链表为空(IsListEmpty( )) . 也就是说链表中至少会存在一个伪节点,这样就可以避免生产者插入节点时 tail 指向的节点被消费者删除而出现的空指针情况.

4.2　新进程延迟唤醒方法

本节介绍新进程延迟唤醒方法的设计和流程. 首先介绍新进程进入延迟队列的过程,该过程在调用 wake_up_new_task( ) 函数唤醒新进程时被执行,改进后的函数流程如下:

A.1　禁止本地中断.

A.2　调用负载均衡模块的接口函数计算并设置新进程 p 的目标处理器核 target_cpu.

A.3　调用 task_rq_trylock( ) 函数对 target_cpu 的运行队列尝试加锁; 如果获取失败则跳到 A.5 步.

A.4　将进程 p 插入 target_cpu 的运行队列,释放 target_cpu 的队列锁,跳到 A.9 步.

A.5　如果本地处理器核 local_cpu 的 dtqueue [target_cpu] 中的循环数组未满,则将 p 放入 dtqueue [target_cpu] 的循环数组中; 否则,将 p 插入 dtqueue [target_cpu] 的链表中.

A.6　获取 target_cpu 的自旋锁 dtlock.

A.7　如果 local_cpu 的 dtqueue [target_cpu] 中的循环数组或链表不为空,则将 target_cpu 的 dtmap 中的第 local_cpu 位置位.

A.8　释放 target_cpu 的自旋锁 dtlock.

A.9　允许本地中断.

目标处理器核在执行 schedule( ) 函数进行调度时,会将

目标为自己的延迟进程插入到自己的运行队列中,保证进程的正常运行和响应,具体流程如下:

B.1 获取本地处理器核 local_cpu 的 dtmap,并存于临时变量 tmpmap 中.

B.2 如果 tmpmap 为零跳到 B.9 步.

B.3 获取 tmpmap 中第一个被置位的位,记为 source_cpu.

算法 3. 免锁链表

```
1  ListDequeue( )
2      if ( IsListEmpty( ) )
3          return NULL
4      p←head
5      head←p^. next
6      tmp←p^. next^. value
7      delete p
8      return tmp
9  ListEnqueue( x)
10     q← new node
11     q^. value ←x
12     q^. next← NULL
13     tail^. next←q
14     tail←q
15 IsListEmpty( )
16     return ( head = tail)
17 ListInit( )
18     n←new node
19     head←tail←n
```

图 7 算法 3: 免锁链表

Fig. 7 Algorithm 3: linked list without lock

B.4 将 source_cpu 的 dtquque [local_cpu] 中的循环数组和链表中的延迟进程全部移出并插入本地运行队列中.

B.5 将 tmpmap 的第 source_cpu 位清零.

B.6 获取 local_cpu 的自旋锁 dtlock.

B.7 如果 source_cpu 的 dtqueue [local_cpu] 中的循环数组或链表都为空,则将 local_cpu 的 dtmap 中的第 source_cpu 位清零.

B.8 释放 local_cpu 的自旋锁 dtlock,跳到 B.2 步.

B.9 继续执行 schedule( ) 函数其余部分.

在对 dtmap 进行置位和清零时,本文仍然使用自旋锁来保证"判断和置位"过程的原子性,即该方法并不能做到严格意义上的无锁. 但此处的锁冲突域很小,对系统性能影响不大.

5 基于上层应用特征的参数调优

Linux 操作系统常用作服务器系统,上层应用程序通常较为单一,相应的进程生命特征通常较为明显. 从进程的生命特征入手对 Linux 调度器进行免锁优化是很好的选择. 当一个新进程或睡眠进程被唤醒时,内核根据一定的策略给该进程抢占当前运行的进程执行的机会,这种抢占称之为唤醒抢占,其目的是使被唤醒进程有更好的响应速率. 一次抢占意味着一次进程调度,一次对调度主函数的调用. 而如 2.2 节所述,调度主函数 shedule( ) 在函数的起始和结尾处加解锁,存在较大的锁冲突域. 这意味着,唤醒抢占的次数越多,锁冲突的可能性就越大.

Linux 调度器通过参数 sysctl_sched_wakeup_granularity,即唤醒抢占粒度,来控制一个进程最少执行多长时间才能被抢占. 对于 Web 服务器来说,服务进程较多但生命期类似,生命长度通常固定在较小的范围内. 可以考虑调节唤醒抢占粒度来控制调度频率,使得进程尽量在被新进程抢占之前执行完毕. 这样既可以减少一次链接请求的完成时间,又能降低调度代码对内核执行效率的影响.

这种优化只需要对调度器的关键参数进行设置,对内核的修改较小,但是效果却较为明显. 然而,参数的设置和处理器核性能及应用进程的生命特征有直接关系,不能套用,需要仔细验证,第 6 节将通过实验进一步分析这一优化方法的特征.

6 实验和分析

本文使用测试基准 hackbench[1] 对优化方法进行评测,通过模拟 C/S 模式下的客户端和服务器的通信来测试 Linux 进程调度器的性能、开销和可扩展性. hackbench 有两种模式: 线程模式和进程模式,由于两种模式的测试结果相似,本文仅列出进程模式下的测试结果. 实验中,每个进程组包括 40 个进程(20 个 client 和 20 个 server) ,每个 server 向每个 client 发送 1000 条消息,通过最终完成通信的时间来评价调度器的性能. 针对 Web 服务器的优化,本文使用 phoronix-test-suite[2] 的 pts/apache 测试基准对优化方法进行进一步评测. 实验的硬件环境: 处理器为 Intel Core i7-3770,4 物理核 8 逻辑核,主频为 3.4GHz,内存为 3GB.

首先对基础优化方法进行评测. 对 Linux 2.6.32 中的 load_balance( ) 和 load_balance_newidle( ) 进行优化,实验表明这种优化方法有一定效果,性能提升在 0% 至 0.3% 的范围内. 由于篇幅关系,本文没有列出具体数据.

本文使用新进程延迟唤醒方法对 Linux 3.4 进行优化,下页表 3 显示的是在启动 4 个和 8 个逻辑核时使用 hackbench 对该方法的评测结果. 可以看出,随着测试规模的增加,两种核数下性能提升分别稳定在 8% 和 6% 左右. 4 核系统上取得的性能提升一定程度上优于 8 核系统,这是因为在相同进程规模下 4 核系统上对同一运行队列的锁竞争比 8 核系统更为严重. 此外,在启动 8 个逻辑核时,本文使用 pts/apache 在相同网络条件下对使用该优化方法前后的系统性能进行评测,每秒处理的请求个数如下页表 4 所示,性能提升十分明显.

基于 Web 服务器的进程特征,本文通过调节唤醒抢占粒度来达到免锁的目的. Linux 3.4 对该参数的设置为 1 微秒,通过大量实验,本文将该参数改为 2 微秒. 下页表 5 显示了在启动 8 个逻辑核的情况下唤醒抢占粒度从 1 微秒到 4 微秒时 pts/apache每秒完成的请求数. 可以看出抢占粒度需要结合处

[1] http: //devresources. linuxfoundation. org/craiger/hackbench

[2] http: //www. phoronix-test-suite. com

理器性能进行仔细评测,如果抢占粒度设置过大,进程的响应速度也随之减小,造成性能的衰减. 表 6 列出了将抢占粒度设置为 2 微秒前后 hackbench 对调度器性能的测试. 由于唤醒抢

表 3　新进程延迟唤醒方法获得的性能提升

Table 3　Performance promotion by using delayed wakeup

组数	4 核			8 核		
	优化前(秒)	优化后(秒)	提升(%)	优化前(秒)	优化后(秒)	提升(%)
20	1.815	1.778	2.039	1.621	1.575	2.838
40	3.753	3.516	6.315	3.289	3.096	5.868
60	5.803	5.395	7.031	5.015	4.738	5.523
80	7.957	7.357	7.541	6.650	6.420	3.459
100	10.095	9.225	8.618	8.409	8.042	4.364
120	12.132	11.017	9.191	10.224	9.460	7.473
140	14.272	12.902	9.599	11.950	11.159	6.619
160	16.353	15.101	7.656	13.677	12.733	6.902
180	18.549	17.052	8.071	15.298	14.388	5.948
200	20.429	18.853	7.715	16.992	15.900	6.427

表 4　使用新进程延迟唤醒方法前后 pts/apche 的测试结果

Table 4　Results of pts/apache by using delayed wakeup

	优化前	优化后
请求数/秒	13851	14665

表 5　四种不同唤醒粒度下 pts/apche 的测试结果

Table 5　Results of pts/apache with 4 wakeup granularity

粒度(微秒)	1	2	3	4
请求数/秒	13851	14508	13648	12790

表 6　唤醒粒度调优获得的性能提升

Table 6　Performance promotion by tuning wakeup granularity

组数	4 核			8 核		
	优化前(秒)	优化后(秒)	提升(%)	优化前(秒)	优化后(秒)	提升(%)
20	1.815	1.803	0.661	1.621	1.569	3.208
40	3.753	3.691	1.652	3.289	3.170	3.618
60	5.803	5.606	3.395	5.015	4.909	2.114
80	7.957	7.712	3.079	6.650	6.382	4.030
100	10.095	9.785	3.071	8.409	8.093	3.758
120	12.132	11.431	5.778	10.224	9.812	4.030
140	14.272	13.618	4.582	11.950	11.551	3.339
160	16.353	15.801	3.376	13.677	13.275	2.939
180	18.549	17.618	5.019	15.298	14.881	2.726
200	20.429	19.268	5.683	16.992	16.329	3.902

占粒度适度增大,由调度主函数 schedule( ) 被调用引发的锁竞争随之减小. 在启动 4 个和 8 个逻辑核的情况下,调度器分别取得了 0.66% 至 5.68% 和 2.11% 至 4.03% 的性能提升.

7　结　论

针对多核环境下 Linux 调度器中锁竞争对系统性能的影响,本文提出了三个层次的优化方法: 基础优化、调度行为优化、基于上层应用特征的参数调优. 基础优化从代码层面缩小锁冲突域,有一定的优化效果; 针对进程创建过程中的唤醒操作,本文提出的新进程延迟唤醒方法有效地减少了进程插入运行队列时的锁竞争,提升了进程创建性能和调度器执行效率; 针对 Web 服务器系统,本文使用参数调优的方式在对内核修改尽量小的情况下有效地减小了调度器的锁操作.

References:

[1] Correa M, Zorzo A, Scheer R. Operating system multilevel load balancing [C]. Proceedings of the ACM Symposium on Applied Computing, New York, USA: ACM Press, 2006: 1467-1471.

[2] Bovet D P, Cesati M. Understanding the linux kernel (3rd edition) [M]. Sebastopol, USA: O'REILLY, 2005.

[3] Boneti C, Gioiosa R, Cazorla F, et al. A dynamic scheduler for balancing HPC applications [C]. Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Piscataway, USA: IEEE Press, 2008: 1-12.

[4] Hofmeyr S, Iancu C, Blagojevic F. Load balancing on speed [C]. Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, USA: ACM Press, 2010: 147-157.

[5] Wei J, Juarez E, Garrido M, et al. A Linux implementation of the energy-based fair queuing algorithm on an ARM-based embedded system [C]. Proceedings of 2014 IEEE International Conference on Consumer Electronics, Las Vegas, USA: IEEE Press, 2014: 546-547.

[6] Salami B, Baharani M, Noori H, et al. Physical-aware task migration algorithm for dynamic thermal management of SMT multi-core processors [C]. Proceedings of 2014 19th Asia and South Pacific Design Automation Conference, Singapore, IEEE Press, 2014: 292-297.

[7] Mauerer W. Professional linux kernel architecture [M]. Birmingham, England, Wrox Press, 2008.

[8] Fraser K, Harris T. Concurrent programming without Locks [J]. ACM Transactions on Computer Systems, 2007, 25(2): 1-59.

[9] IBM. System/370 principles of operation [M]. Indiana, USA: IBM Press, 1970.

[10] Peng Jian-zhang, Gu Nai-jie, Zhang Xu, et al. Fast epoch: a fast memory reclamation algorithm for lock-free programming [J]. Journal of Chinese Computer Systems, 2013, 34(12): 2691-2695.

[11] Zhou Wei-ming. Multi-core computing and programming [M]. Wuhan: Huazhong University of Science and Technology Press, 2009.

附中文参考文献:

[10] 彭建章,顾乃杰,张　旭,等. 快速时代回收: 一种针对无锁编程的快速垃圾回收算法 [J]. 小型微型计算机, 2013, 34(12): 2691-2695.

[11] 周伟明. 多核计算与程序设计 [M]. 武汉: 华中科技大学出版社, 2009.