·体系结构与软件技术· 文章编号：1000—3428(2013)05—0084—04 文献标识码：A 中图分类号：TP311

快速上下文切换调度算法

刘超明，钱振江，黄 皓

(南京大学计算机科学与技术系软件新技术国家重点实验室，南京 210093)

摘 要：在调度过程中，拥有相异地址空间的换入换出线程切换代价过大，以及将线程作为时间片分配的唯一主体会导致进程层面上的不公平。针对上述问题，提出一种快速上下文切换调度算法。通过创新组织就绪队列中的调度体，使属于同一进程的线程尽量靠拢，以便优先选择，同时在分配时间片时考虑进程的线程总量。实验结果证明，该算法能减少系统的切换代价，并控制进程获取的时间片总量，提高系统的执行效率和公平性。

关键词：NTOS 系统；微内核；调度算法；就绪队列；上下文切换；时间片

Fast Context Switching Schedule Algorithm

LIU Chao-ming, QIAN Zhen-jiang, HUANG Hao

(National Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

【Abstract】 There are two problems in normal schedulers, one is that threads with different contexts cost a lot while scheduling, another is that it is unfair to allocate time slice without considering thread-number of process. This paper proposes a Fast Context Switching Scheduler(FCSS) algorithm. It takes threads belonging to the same process as close as possible in the ready queue, and the total amount of threads into consideration when allocating time slice. Experimental results show that the algorithm reduces the cost of switching of the system, controls the total amount of time slice, and improves the efficiency and fairness of the system.

【Key words】 NTOS; micro-kernel; schedule algorithm; ready queue; context switching; time slice

DOI: 10.3969/j.issn.1000-3428.2013.05.017

1 概述

NTOS 是一个由 Minix 演化而来的微内核操作系统，在为其实现多线程的过程中，发现线程的调度方式对系统的性能和安全都有很大的影响。一方面，属于同一个进程的线程拥有相同的地址空间，在调度过程中，如果换入换出线程属于同一进程，将很大程度上减小性能耗损。然而，对一些常用的调度算法进行分析后发现，很多系统都将线程作为完全独立的调度体，“杂乱”地摆放在就绪队列中，完全不受所属进程的限制，这无形中减小了属于同一进程的线程在就绪队列中相邻的几率，很多时候这种杂乱是不必要的。另一方面，系统中执行体的优先性主要通过 2 个方面来体现，被授予的优先级和分配到的时间片。有些进程拥有大量的线程，而有些进程只有几个线程，如果这些线程都独立地获取时间片，那么就进程这个层面来讲是非常不公平的，另外，对于进程通过创建线程恶意的攫取系统时间这种情况，普通的调度算法是无能为力的。NTOS 系统中的调度体仍是线程，但为了提升系统的切换效率和公平性，本文在就绪队列中引入进程，以便合理有效地组织线程，达到预期目的。为了叙述方便，将属于同一进程的线程称为兄弟线程。

2 相关知识

操作系统的调度算法很大程度上决定于它的目标市场，反之亦然，了解操作系统的目标市场有利于解释其调度算法[1]。所谓调度算法即什么时候以什么样的方式选择一个执行体运行的规则[2]。常用的操作系统根据侧重点的不同会选择不同的调度策略，但有一个共同点，即对于不支持线程的系统，就绪队列中的调度体是进程，否则，调度体是线程，它们的组织方式不受所属进程的限制。快速上下文切换调度器(FCSS)主要目标是提升线程切换的效率，同时增强调度在进程层面的公平性，FCSS 的公平性主要体现

基金项目：国家“863”计划基金资助重大项目“以支撑电子商务为主的网络操作系统研制”(2011AA01A202)；江苏省“六大人才高峰”高层次人才基金资助项目(2011-DZXX-035)

作者简介：刘超明(1986－)，男，硕士研究生，主研方向：安全操作系统；钱振江，讲师、博士研究生、CCF 会员；黄 皓，教授、博士生导师

收稿日期：2012-05-09 **修回日期**：2012-06-27 **E-mail**：liuchaoming8@126.com

在时间片的控制上。一些研究着重于提高调度在用户层面的公平性[3]，在分时系统中，经常将一些用户划分一组，如果同一组中有较多的用户同时登录，可能希望响应效率的降低较多甚至是仅影响到本组，而不是不加区分地影响所有用户，不同于 FCSS，该研究将重点放在调度策略的切换上。本文将介绍 Linux 和 Windows 2 个具有代表性的系统的调度算法，以便做出对比。

2.1　Linux 调度算法

Linux 根据优先级对进程进行分类，进程的优先级也是动态变化的，系统凭借经验确定时间片的大小，尽可能保证良好的响应度。

Linux2.6 的调度算法比较复杂，它能在固定的时间内选中要运行的进程。对于多核平台，早期算法扩展性很差，所有的 CPU 共享一个就绪队列，随着负载和 CPU 数量的增加，对整个就绪队列上锁会带来很大的消耗，而后期算法则支持多个就绪队列，效率明显得到提升[4]。

runqueue 是 Linux2.6 调度程序最重要的数据结构，每个 CPU 对应一个，用于组织就绪进程。runqueue 有一个 arrays 字段，它是包含 2 个类型为 prio_array_t 的结构的数组，每个结构都包含 140 个双向链表头，用来挂载就绪进程，显然，这 2 个结构分别对应过期队列和活动队列。过期队列和活动队列并不是一成不变的，runqueue 中的 active 字段指向活动队列，expire 字段对应过期队列，当活动队列为空时，调度只需要交换 active 和 expire，则过期队列变为活动队列。

2.2　Windows 调度算法

Windows NT[5]是一个可抢占的多任务操作系统，线程的优先级、时间片可根据具体情况进行动态调整，以提高系统的吞吐量并减少响应时间。Windows XP 中的进程都由一个执行体进程块(EPROCESS)表示，进程是地址空间和线程的容器[6]。线程是实际的调度单位，由 ETHREAD 表示。

为了做出有关线程调度的各种决定，内核维护了一组数据结构，它们合起来称为分发器数据库[7]。分发器数据库跟踪记录了哪些线程正在等待执行，哪些处理器正在执行哪些线程。分发器的就绪队列包含了 32 个优先级，每个优先级都对应一个队列，链接等待被调度的线程。为了更快地选中下一个被调度的线程，Windows 维持了一个 32 位的掩码，称为就绪摘要。某一位若被设置，则表示其对应的优先级队列不为空。进程将自己的所有线程链接起来，并为它们提供了参考优先级，但线程是就绪队列中唯一的调度实体，它们的链接方式与所属进程无关。

3　快速上下文切换调度算法的设计目标

通过之前的分析可以发现，对于很多支持线程的实用操作系统来说，其调度程序有 2 个基本特点。首先，调度体是线程，就绪队列中链接着线程控制块，链接方式与所属进程无关。其次，线程的时间片要么是固定的，要么受其优先级影响，进程所拥有的线程数量完全不会影响到线程对时间片的获取。

对于前者，属于同一进程的线程拥有同样的地址空间，如果调度程序换入线程与换出的线程为这种情况，那么内核不需要重置 LDTR、CR3 并刷新硬件缓存。相反，则必然带来巨大的损耗。

对于后者，在一般情况下，工作量大的进程往往拥有更多的线程，因此会获得与线程数量呈正比的更多的执行时间。考虑一种情况，如果系统中存在 2 个进程，一个优先级较高且拥有大量的线程，另一个优先级较低且拥有较少的线程，那么优先级较低的进程往往会饥饿并严重影响系统的周转时间。当优先级较高的进程通过恶意创建线程以攫取更多的系统时间时，此情况更甚。固然可以通过动态调整线程优先级来缓解饥饿问题，但恶意进程占用大量系统时间已成为不争的事实，因此直接限制进程所拥有的线程的时间片总量是更根本的解决方法。

综上所述，快速上下文切换进程调度在吸取其他调度方法优点的同时，还要实现 2 个目标:

(1)合理组织就绪队列中的线程，尽量减少不必要的上下文切换，提升系统效率。

(2)重新设计线程时间片的分配方式，使线程获得的时间片与其所属进程的线程拥有量相关。

4　快速上下文切换调度算法的设计

为了让兄弟线程在就绪队列中尽量靠近，并须有一个媒介将它们串联起来，在适当时给兄弟线程统一分配时间片也同样如此，这个媒介就是进程。接下来就就绪队列、调度过程和时间片的分配进行详细设计。

4.1　就绪队列的设计

针对不同的应用场合，就绪队列的设计方式分别为单核相同优先级、单核相异优先级、多核相同优先级和多核相异优先级，其中，相同优先级指的是兄弟线程的优先级可以随进程变化，但必须一致，相异优先级指的是兄弟线程可以拥有不同的优先级。本文仅介绍单核相同优先级和多核相异优先级的设计，实际上，多核相异优先级就绪队列的设计模型涵盖了前三者。

4.1.1　单核相同优先级就绪队列的设计

就绪队列是一个二层链表，共分 16 级。从 0 到 15，0 是最高优先级。第一级链接的是进程控制块，第二级链表才是线程控制块，如图 1 所示。进程控制块用于管理地址空间，共享消息等公共资源，其中有 2 个重要的字段 chirldren_thread_q 和 ready_thread_q，前者链接进程所有的线程，后者与调度相关，链接进程所有的就绪线程。兄弟线程通过所属进程被组织在一起，调度程序为了减小切换代价总是可以很方便地优先调度属于同一进程的线程，而不需要遍历整个链表。后文提到的一级链表即进程所在的链，二级链表即某个进程的就绪线程链。

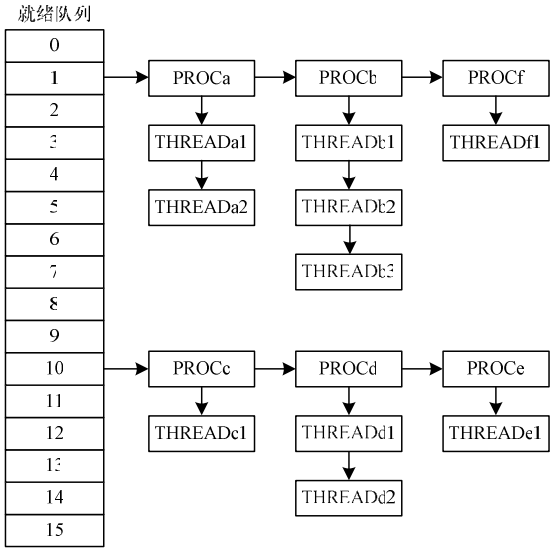

图 1 单核相同优先级就绪队列

4.1.2 多核相异优先级就绪队列的设计

多核处理器的线程调度是一个 NP-完全问题[8]，相关技术的研究尚不成熟。在多核系统运行多线程，首先要面对的是资源的共享与争用问题，为尽量减少资源争用，进一步提升系统的运行效率，必须有针对性地设计线程调度策略[9]；为减少线程间通信的开销，可以使用基于迭代的启发式算法对线程进行分组，提升多核处理器的效率[10]。这些在 FCSS 的后期设计和实现中都会有所体现，但不是本文的重点。在 FCSS 中，每个核都有自己的就绪队列，系统中每个进程对应唯一的进程控制块，每个线程对应唯一的线程控制块，线程控制块仅可能出现在某一个就绪队列中，而进程控制块则不然，因为它的多个就绪线程可能被分派到不同的处理器上。显然，进程控制块仅用指针 ready_thread_q 链接就绪线程机变得不合时宜，因为不同的就绪队列无法简单快速地定位被分派到自身的线程，所以必须改变进程控制块组织就绪线程的方式。为了支持兄弟线程的多优先级，同样需要面对这个问题。

系统中每个核对应的就绪队列与图 1 相似，指针 ready_thread_q 不再直接指向一串就绪线程，而是指向一个结构体，如图 2 所示。

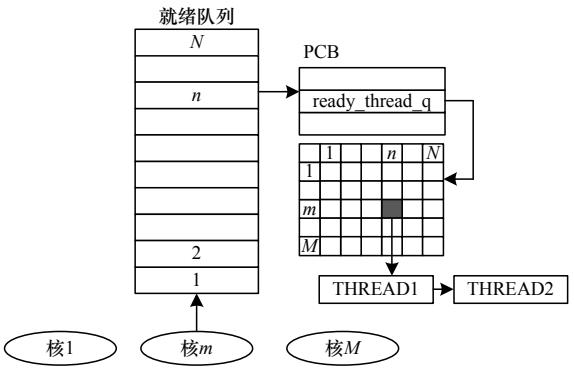

图 2 多核相异优先级就绪队列

假设系统中有 M 个核，支持 N 个优先级，那么进程控制块的 ready_thread_q 字段指向一个 $M\times N$ 的矩阵，矩阵中的每个元素(m, n)都是指针，指向进程被分派到核 m、优先级为 n 的线程。表面上看来，PCB 及指针矩阵是共享资源，但调度程序为某个核选择线程时并不需要加锁，因为每个核都对应矩阵中相异的一行，它们并不交叉。

4.2 调度过程的设计

调度过程即调度程序操作就绪队列，选择换入线程并执行的过程。调度过程分为 2 个步骤：(1)选择换入线程；(2)在从内核态返还时切换到该线程。在系统中维持 2 对变量 (CURRENT_PROC 、 CURRENT_THREAD) 和 (NEXT_PROC、NEXT_THREAD)，分别表示当前正在运行的线程及对应的进程和调度器选中的换入线程及其对应的进程。

当前线程主动放弃 CPU 或当前线程时间片到期时，调度程序会被激活，接下来要做的便是依据先高优先级后低优先级，先队首后队尾的策略选择换入线程。

系统在返回用户态前始终检查有没有必要切换线程，这时将考察 3 种情况：

(1)若 CURRENT_PROC 与 NEXT_PROC 不同，说明换出线程与换入线程分属不同进程，则需要切换地址空间，重新装载 CR3、LDTR 等寄存器，将目标线程的内核态栈地址存入 TSS 并加载保存在内核栈中的上下文。

(2)若 CURRENT_PROC 与 NEXT_PROC 相同，但 CURRENT_THREAD 与 NEXT_THREAD 不同时，说明需要切换线程，但不必切换地址空间，需要将换入线程的内核态栈地址存入 TSS，并加载保存在内核栈中的上下文。

(3)若 CURRENT_PROC 与 NEXT_PROC 相同，则 CURRENT_THREAD 与 NEXT_THREAD 也相同，说明不需要线程切换，这时仅需要加载保存在内核栈中的上下文。

4.3 时间片的分配

为使时间片的分配与线程总量相联系，必须以进程为单位在恰当的时机分配时间片，再将此时间片均分到线程。

4.3.1 时间片的分配时机

在就绪队列中，线程是实际的调度体，而进程只是链接兄弟线程的纽带。当新线程创建时，其初始时间片为 0，当就绪线程时间片用完时，并不会重新分配，它们都被直接加入到二级链表队尾。

若调度程序发现进程的二级链不为空，但队首的线程时间片为 0，则说明进程所有的就绪线程时间片都已经耗尽，则将进程从一级链表队首摘除，分配时间片并加入一级链表队尾。

4.3.2 时间片的分配算法

进程首先获得总时间片，总时间是根据就绪线程链的线程数目 N 和进程的线程总量 M 计算得来的，然后均分给就绪线程链中的线程。计算公式必须使得总时间随着 M 和 N 的增加呈趋缓增长方式。如图 3 所示，虚线部分是普通时间片分配算法进程总时间片的增长趋势，不考虑线程总量 M，与就绪线程量 N 呈正比；实线对应的是快速上下文

切换时间片分配算法。

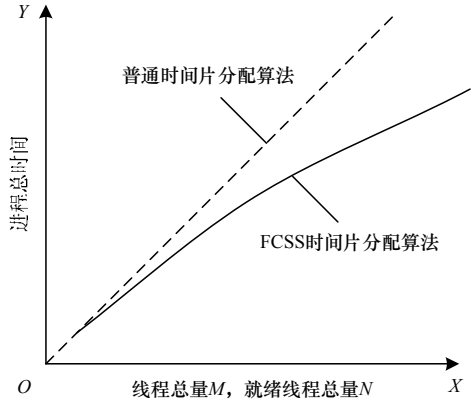

图 3　进程总时间片随 *M*、*N* 增长的趋势

假设系统的基本时间片为 s，时间片总量为 $T(M, N)$，则计算公式如下：

$$T(M,N)=s\sum_{i=0}^{i<N}\frac{M-i}{M}$$

即：

$$T(M,N)=\frac{sN(2M-N+1)}{2M})$$

可以看出，当 N 固定时，若 M 越大，则 T 越大，当 M 固定时，若 N 越大，T 也越大，但是呈趋缓增长方式。最终，每个就绪线程获得的时间片 *ThreadTime* 以下公式计算得到：

$$ThreadTime=\frac{T}{N}$$

5　快速上下文切换调度算法的实现

就绪队列是链表的链表，即有两级。第一级链接的是进程，第二级链接的是线程。显而易见，需要 2 组函数，由于实际的调度体是线程，因此一级链表的操作函数总是由两级链表的操作函数适时触发。

5.1　二级链表的操作

二级链的操作函数有 4 个，即 enqueue_t、dequeue_t、sched_t 和 pick_thread。

(1)enqueue_t 将一个线程加入其所属进程的二级链。

(2)dequeue_t 将一个线程从所属进程的二级链中删除。

(3)sched_t 判断线程链入二级链表的首或尾。

(4)pic_thread 依据预定义的模式选择就绪线程。如果函数发现队首进程二级链的队首线程时间片为 0，则为进程重新分配时间片，并加入就绪队列，然后重新选择线程执行。

5.2　一级链表的操作

一级链的操作函数有 3 个，即 enqueue、dequeue 和 sched。

(1)enqueue 确定进程在一级链中的位置。

(2)dequeue 将进程从一级链中删除。

(3)sched 根据二级链队首线程的时间片情况，判断进程应插入一级链的首或尾。

6　结束语

快速上下文切换调度算法着眼于消除不必要的切换代价，并控制进程获取的时间片总量，以达到在某些方面提升系统运行效率和安全性的目的。FCSS 的特点主要体现在就绪队列、调度过程和时间片分配的设计上。为将兄弟线程尽量组织在一起，FCSS 就绪队列被设计成 2 个层级的链表：一级链表链接 PCB，二级链表链接 TCB。调度程序可以容易地为换出线程找到换入的兄弟线程。为应对优先级和单核多核等不同要求，本文提及了 4 种就绪队列设计方案，但本质目标是一致的。针对换出换入线程的 3 种情况，调度过程做出不同的处理，系统相应地会产生不同的切换代价。为维护系统在进程层面的公平性，FCSS 的时间片分配考虑了进程的线程拥有量这个要素。实验结果证明，FCSS 的切换效率更高，同时有效遏制了恶意进程对时间片的无限攫取。FCSS 调度的公平性首先体现在进程层面，进而扩展到线程层面。进程在一级链表中排队，线程在二级链表中排队，线程的调度时机由这 2 个链表决定。根据本文提到的调度选择和时间片分配策略，一个线程必须等到所有兄弟线程时间片耗光或被阻塞，才能再次获得时间片并排队等待运行。为了提高灵活性，算法应在多种模式间切换，因此，切换时机和切换方法将是下一步的研究重点。

参考文献

[1] Aas J. Understanding the Linux 2.6.8.1 CPU Scheduler[Z]. Silicon Graphics, Inc., 2005.

[2] Bovet D P, Cesati M. Understanding The Linux Kernel[M]. 陈莉君, 张琼声, 译. 北京: 中国电力出版社, 2007.

[3] 陈　媛, 杨　武, 面向用户的进程调度策略研究与实现[J]. 计算机工程, 2008, 34(10): 78-79.

[4] Kravetz M. Enhancing Linux Scheduler Scalability[EB/OL]. (2012-05-10). http://lse.sourceforge.net.

[5] 刘　波, 李冠英. Windows NT 线程调度技术分析与应用[J]. 计算机工程, 2001, 27(6): 171-173.

[6] Probert D B. Windows Kernel Internals Thread Scheduling[EB/OL]. (2012-04-10). http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/03-ThreadScheduling/ThreadScheduling.Ppt.

[7] Russinovich M E. Microsoft Windows Internals[M]. 潘爱民, 译. 北京: 电子工业出版社, 2010.

[8] Ullman J D. NP-complete Scheduling Problems[J]. Journal of Computer and System Sciences, 1975, 10(3): 384-393.

[9] 王　晶, 樊晓桠, 张盛兵, 等. 多核多线程结构线程调度研究[J]. 计算机科学, 2007, 34(1): 256-258.

[10] 杨洪斌, 陈　伟, 吴　悦. 基于粒子群算法的多核处理器线程调度研究[J]. 计算机工程与设计, 2010, 31(5): 1045-1047.

编辑　索书志