基于新信号量策略的实时提升技术

王 波,崔 喆
(中科院成都计算机应用研究所,四川 成都 610041)

摘 要:介绍操作系统内核对实时性能的影响,结合NT技术,分析信号量机制下线程等待队列的排队策略,提出一种新排队策略,并在NT内核中实现该策略,最后对比几种策略的实验数据。

关键词:实时操作系统;信号量;FIFO;LIFO;Priority;NT内核
中图分类号:TP316.7 **文献标识码**:A **文章编号**:1674-7720(2012)08-0064-03

Promotion of real-time capability with new semaphore strategy

Wang Bo, Cui Zhe
(Chengdu Institute of Computer Applications of Chinese Academy of Science, Chengdu 610041, China)

Abstract: On the basis of introduction of real time facts of modern operating system kernel, and analysis of queuing tactics of blocked thread (s) in semaphore under traditional kernel, purposes a new tactic. At the end of this paper, detail of new semaphore strategy based on combined priority will be given, followed by its realization in NT kernel.

Key words: real time operating system; semaphore; FIFO; LIFO; Priority; NT kernel

1 操作系统对实时性能的影响

操作系统从诞生发展到现代经历了批处理系统、分时系统和实时系统等演进过程,具有多样化特征,派生出不同分支。其中,实时性是操作系统的重要特性,它要求在规定的时间窗口内逻辑正确地完成规定的任务,具有及时性、交互性、多路性、独立性等特点[1]。操作系统的实时性主要取决于I/O管理中的异步方式、内存管理中的页中断机制、线程管理中的内核代码是否可抢占、资源管理中的信号量策略以及中断延迟和时钟精度等硬件支撑结构[2]。由于多线程系统中线程对公共资源的争夺,资源的有效管理成为提升系统实时性能的重要因素,而信号量是管理公共资源的经典方式,所以,信号量设计是影响系统实时性的基础设计。本文重点论述信号量策略对实时性能的影响,并以NT内核为研究对象和实现平台,分析现有几种信号量策略的优、缺点,提出了一种新策略,在保证系统通用性前提下提升了系统实时性。

2 信号量策略对实时性能的影响

荷兰科学家设计的信号量算法为线程使用共享资源提供了有效的同步和互斥机制,NT内核中,信号量(KSEMAPHORE)通过封装DISPATCHER_HEADER结构实现计数器和等待队列,其数据结构struct _KSEMAPHORE{DISPATCHER_HEADER Header LONG Limit}在参考文献[3]中有详细描述,上述结构可简略为:

```
struct _KSEMAPHORE{LONG SignalState          //信号量计数器变量
    LIST_ENTRY WaitList}                     //线程等待队列链表
```

它的操作有创建(CreateSemaphore)、删除(CloseHandle)、请求(WaitForSingleObject)和释放(ReleaseSemaphore)信号量等。

线程使用资源前需要请求保护该资源的信号量,若信号量计数器减1后小于0,内核阻塞线程并将其排在信号量的线程等待队列中,同时启动线程调度程序将计算资源交给等待运行的线程,执行请求操作的线程没有陷入“忙等”,而是“让权等待”。若拥有信号量的线程释放资源使得计数器加1后还小于等于0,则唤醒线程等待队列中的等待线程并送线程调度队列。因此,在资源紧张情况下,请求和释放信号量会涉及资源等待队列和线程调度队列两个队列。本文讨论资源等待队列,对于资源请求,采用什么策略将线程放入队列;对于资源释放,采用什么策略把线程从队列中取出并放入调度队列。考虑放入与取出线程时同时采用策略的复杂性,固定取出策略从队列头部取出线程,请求时采取策略将线程放入队列,目前有以下三种策略[1]:

(1)后进先出 LIFO(Last In First Out),线程请求资源后,若信号量计数器小于 0,将线程排在线程等待队列的队头。该策略易于实现,线程等待队列只需一个单链表即可,这种“后来先服务”的方式还可以利用 CPU 缓存 TLB(Tanslation Lookaside Buffer)中存在的刚被挂起线程的页表数据,不必更新缓存,提高了运行速度。但是,后进先出方式让最先被挂起的线程鲜有被服务,若获得资源的线程高频率请求资源,会导致最先请求资源的线程由于长时间处在队尾得不到服务导致“饿死”(Starvation),使得一些线程频繁调度,而一些线程很少被调度。

(2)先进先出 FIFO(First In First Out),线程请求资源后,若信号量计数器小于 0,将线程排在线程等待队列的队尾。该策略克服了线程的“饿死”现象,对资源有请求的线程都能公平地占有资源,请求队列调度均衡化,从策略角度来看,所有线程都整齐划一无差别。这种先来先服务的方式没有考虑线程的其他因素,例如,对时间紧要程度的要求不同,有实时线程和一般线程之分,如果对实时线程和一般线程都采用先进先出方式,那么实时线程的实时性将大幅降低,特别在等待队列中已有很多线程的情况下,实时线程只有等待前面所有线程释放信号量后才能得到调度,造成不必要的延时。信号量的数据结构和操作也要复杂一些,需要一个队尾指针。

(3)基于优先级队列 Priority,线程请求资源后,若信号量计数器小于 0,则将线程根据其优先级排在线程等待队列的相应位置。该策略克服了先进先出的均衡化调度缺点,使优先级高的线程始终处在队列的队首,抢占优先级低的线程;线程可根据时间特性来确定它的优先级并排队,提高了线程的实时性。然而这种方式也有其不足,优先级低的线程始终得不到调度,同样会导致“饿死”。如果系统中有大量线程争抢稀有资源,排队过程还会引入队列的搜索时间。

这就需要一种策略,对于具有很强时效性的实时线程使用优先级排队,对于一般线程按照先进先出排队。由于实时线程很少,配合哈希(Hash)表分类实时线程(如图 1 虚直线上部分所示)基本不会引入搜索时间。

3 基于 Priority 和 FIFO 结合的信号量策略

针对优先级队列过分强调高优先级线程的缺点和先进先出队列过分强调平均的缺点,本文提出基于优先级和先进先出队列结合的排队策略,同时兼顾实时线程的强实时要求和一般线程的公平要求。

NT 内核将用户线程以一对一方式映射到内核中,并基于优先级调度内核线程,内核将优先级从低到高分为 32 级,0~15 级为一般线程,16~31 级为实时线程。本文将这种线程调度队列的分级方式见之于信号量的等待队列,如图 1 虚直线上部分所示,把线程等待队列构造成一个长度为 17、类型为 LIST_ENTRY 的哈希(Hash)指针数组,数组 1~16 根据优先级排列同一级别的实时线程,数组 0 根据先进先出排列一般线程。线程请求资源后,若信号量计数器小于 0,且线程优先级小于 16,则将该线程按照先进先出策略排在线程等待队列的队尾;若线程优先级大于等于 16,则按照优先级排列该线程。当线程释放资源时,若信号量计数器小于 0,内核应先从优先级队列中搜索挂起线程,再从先进先出队列中搜索挂起线程。

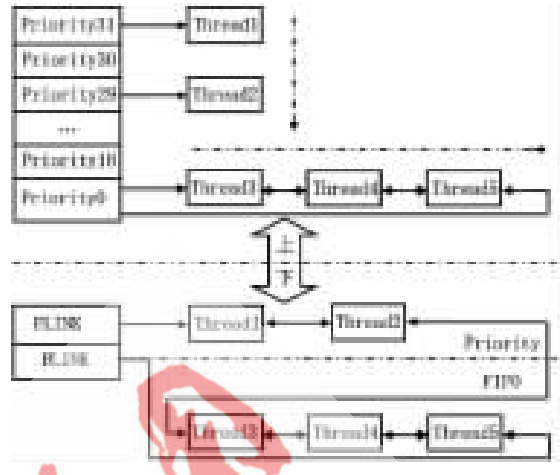

图 1 Priority and FIFO 实现

4 新信号量策略在 NT 内核中的实现及结果分析

为了兼容操作系统上层软件,本文仅修改“请求”函数的代码而不改变现有信号量的数据结构,将图 1 虚直线上部分描述的新信号量策略映射到虚直线下,把优先级队列和先进先出队列融合到一个队列中,队列的前半部分是优先级队列,由指针 FLINK 指定,后半部分为先进先出队列,由指针 BLINK 指定,这种复合型队列同时具备优先级和先进先出队列的优点,体现了“一个队列两种策略”。线程请求资源后,若信号量计数器小于 0,且线程的优先级小于 16,按照先进先出策略将线程排在 BLINK 指向的先进先出队列队尾;若线程的优先级大于等于 16,则将线程按照优先级策略在 FLINK 指向的优先级队列中搜索相应的位置,找到小于优先级队列中的线程并放在该线程之后。当线程释放资源时,若信号量计数器小于 0,由于线程已经根据策略放入恰当的位置,内核只需要从 KSEMAPHORE→WaitList→FLINK 取出第一个线程送往线程调度队列即可。为了最小化修改范围,用下述代码替换内核 \base\ntos\ke\wait.c 文件中 KeWaitForSingleObject()函数的部分代码以实现新策略:

```
if (KeQueryPriorityThread(Thread) < 16)
    {InsertTailList(&Objectx->Header.WaitListHead,
                    &WaitBlock->WaitListEntry);}
else {ListHead1 = &Objectx->Header.WaitListHead;
      WaitEntry1 = ListHead1->Flink;
      while(WaitEntry1 != ListHead1) {
      WaitBlock1 = CONTAINING_RECORD(WaitEntry1,
                    KWAIT_BLOCK, WaitListEntry);
```

```
if(KeQueryPriorityThread(Thread) >
    KeQueryPriorityThread(WaitBlock1->Thread))
                                        {break;}
WaitEntry1 = WaitEntry1->Flink;}
InsertTailList(WaitEntry1, &WaitBlock->
                                    WaitListEntry);}
```

根据C规范[4]设计一个应用程序测试内核修改后的性能指标，由于NT内核对于一个特定的进程只能有一个特定的优先级类，进程内的所有线程只能属于该优先级，程序应该第一次进程化为实时类型的主控进程，生成信号量和挂在信号量上的实时线程，第二次进程化为一般类型的客户进程，生成挂在信号量上的一般线程，主线程释放实时线程和一般线程。应用程序中有4个参量：一般线程数NrTh、实时线程数RtTh；信号量Seph及资源争夺时间RunT。实验中，固定Seph=1，RunT=10 000 ms，改变NrTh和RtTh的值，分别在表1所列的内核上运行，结果如表1所示。

从表1可以看出：1~12行的调度结果和前述分析的各种策略的优缺点一致，对于FIFO，无论不同优先级线程的比例是多少，它们被调度的次数几乎完全相同。对于LIFO，从数据可以看出，两个优先级为8、一个优先级为6和优先级为26、25、24的线程处在等待队列的前端，而且几乎每次都是这几个线程被调度。对于Priority，无论是否有实时类线程，只要优先级高，被调度的次数就多。对于新策略(Priority and FIFO)，有实时线程就按优先级调度，若只有一般线程就按照FIFO调度，既有FIFO的特性(比较第2行和第11行)也有Priority的特性(比较第1行和第4行)，而其他策略则只具有一种特性。应用程序在其他操作系统测试结果见14~22行，比较可以看出，14~22行的数据与10~12行的数据几乎完全一致，由此可以推断Windows 7操作系统的信号量等待队列也是先进先出策略。

研究发现，提升系统实时性应该具备两个条件[5]：(1)不同任务可统一，包括将中断任务和线程任务按不同特征统一映射到一个优先级队列中，内核根据这个优先级队列统一调度任务，中断线程化为上述两种任务的统一提供了可能；(2)所有资源可抢占，计算资源可抢占可行且易实现，而内存资源和I/O资源可抢占需进一步研究，一个占有共享资源(如临界区)的低优先级线程被一般优先级线程抢占计算资源，而该线程又被高优先级的线程抢占计算资源，高优先级的线程又因请求已被低优先级线程占有的资源而挂起，只有等待一般优先级线程放弃计算资源后由低优先级线程运行并释放共享资源，才能使高优先级线程得以运行，虽然通过优先级继承避免优先级反转可以提高实时性，但若高优先级线程能像抢占计算资源那样抢占线程的其他资源，实时性将大幅提升。

表1 内核使用不同策略的运行结果

Priority and FIFO	1	NrTh:0 RtTh:4	322	162	161	2								
	2	NrTh:8 RtTh:0					82	82	82	82	82	82	81	81
	3	NrTh:8 RtTh:4	322	162	161	2	2	2	2	2	2	2	2	2
Priority	4	NrTh:0 RtTh:4	322	162	162	2								
	5	NrTh:8 RtTh:0					322	162	161	2	2	2	2	2
	6	NrTh:8 RtTh:4	322	162	161	2	2	2	2	2	2	2	2	2
LIFO	7	NrTh:0 RtTh:4	2	321	162	162								
	8	NrTh:8 RtTh:0					2	2	2	2	2	215	214	215
	9	NrTh:8 RtTh:4	2	2	2	2	2	2	2	2	2	214	215	215
FIFO	10	NrTh:0 RtTh:4	162	162	162	161								
	11	NrTh:8 RtTh:0					82	82	82	82	82	82	82	81
	12	NrTh:8 RtTh:4	56	56	56	55	55	55	55	55	55	55	55	55
	Priority		31	26	25	24	15	10	9	8	8	8	8	6
Windows XP	13	NrTh:0 RtTh:4	162	162	162	162								
	14	NrTh:8 RtTh:0					82	82	82	81	82	81	81	81
	15	NrTh:8 RtTh:4	56	56	56	55	55	55	55	55	55	55	55	55
Windows 2003	16	NrTh:0 RtTh:4	162	162	162	162								
	17	NrTh:8 RtTh:0					82	82	82	82	82	82	81	81
	18	NrTh:8 RtTh:4	56	56	55	55	55	55	55	55	55	55	55	55
Windows 7	19	NrTh:0 RtTh:4	163	162	162	162								
	20	NrTh:8 RtTh:0					82	82	82	82	82	82	82	82
	21	NrTh:8 RtTh:4	55	55	55	55	55	55	55	55	55	55	55	54

参考文献

[1] 汤子瀛，哲凤屏，汤小丹.计算机操作系统[M].西安：西安电子科技大学出版社，2001.

[2] 倪远桢，阎慧娟，罗克露.计算机组成原理[M].北京：电子工业出版社，1996.

[3] SOLOMON D A，RUSSINOVICH M E.Microsoft Windows Internals Fourth Edition[M].DigitalCopy，Microsoft Press，2004.

[4] KERNIGHAN B W，RITCHIE D M.The C programming language[M].DigitalCopy，Prentice-Hall，1988.

[5] 王波，崔喆.基于新DPC机制的实时提升技术[J].计算机应用研究，2011，28(Z)：110-111.

(收稿日期:2011-02-23)

作者简介:

王波,男,1978 年生,硕士,工程师,主要研究方向:实时数据传输,NT 内核。

崔喆,男,1970 年生,博士,高级工程师,主要研究方向:计算机应用软件技术。