第39卷第12期 计算机研究与发展 Vol. 39, No. 12
2002年12月 JOURNAL OF COMPUTER RESEARCH AND DEVELOPMENT Dec. 2002

一个基于多线程的优先级继承协议锁的算法研究

郭长国 周明辉 王怀民 许 勇
(国防科学技术大学计算机学院网络与信息安全研究所 长沙 410073)
(rtcorba@sohu.com)

摘 要 实时线程库对构造实时中间件和开发具有良好可移植性、有实时要求的分布式应用具有重要意义.防止优先级翻转的线程互斥和同步机制是实现实时线程库的核心,目前多数的线程库都缺乏这种机制.基于优先级继承协议,提出了一个防止优先级反转的互斥算法.算法能够保证操作的原子性,可以避免发生死锁,且能够有效地防优先级翻转.在Windows和Solaris平台上对性能进行了分析,并将算法应用到了实时CORBA工程实践之中.

关键词 实时,固定优先级调度,线程库,优先级继承,互斥,实时CORBA

中图法分类号 TP311.52

A MULTI-THREADED MUTEX ALGORITHM BASED ON PRIORITY INHERITANCE PROTOCOL

GUO Chang-Guo, ZHOU Ming-Hui, WANG Huai-Min, and XU Yong
(*Institute of Network Technology & Information Security, School of Computer Science, National University of Defense Techonology, Changsha* 410073)

Abstract Real-time thread library is very important for building real time middleware. It is helpful for the portability of distributed application which possess time-critical aspects. Mutex and synchronization mechanism to protect priority inversion is the key for real-time thread library. Based on priority inheritance protocol, a muti-thread mutex algorithm is presented, which can prevent deadlock and priority inversion. All the properties of the algorithm are proved correct including mutex, deadlock and priority inversion. In Windows and Solaris, the performance is analyzed. At the same time, this algorithm has been adapted in real-time CORBA application successfully.

Key words real time, fixed priority scheduling, thread library, priority inheritance, mutex, real-time CORBA

1 引 言

随着应用集成度的提高,出现了许多"混合型"(mixed-mode)[1]的应用:这些应用一方面有较强的时间要求,另一方面又要求很强的网络功能和桌面处理能力,比如C⁴ISR系统、视频点播、航天控制、医疗监控系统等.专用的实时操作系统和传统的分

原稿收到日期:2001-09-24;修改稿收到日期:2002-01-24
本课题得到国家"八六三"高技术研究发展计划基金(2001AA113020)、国家自然科学基金(90104020)、国家重点基础研究发展规划基金(G1999032703)资助

布软件开发平台都无法满足这些应用的需求.使用具有实时功能的中间件来开发这类应用就成了当前一个重要的研究课题[2].

在构造这类系统和实时中间件时,存在如下的问题:①中间件虽然屏蔽了平台的差异,但是这些系统过多地依赖于特定平台上的并发机制,如 Win32 和 Solaris 就使用完全不同的线程库[3],这极大影响了系统的移植性,无法充分发挥中间件的可移植优势;②通用操作系统虽然提供了实时扩展,使构造这类系统成为可能,但是它们的调度模型各不相同,使得实时中间件的实现非常困难.这些问题的存在迫切要求一个能够屏蔽平台差异的,具有统一接口(最好是面向对象的)的实时线程库来支持这类系统和实时中间件的开发.

在实时线程库和实时中间件中,最关键的技术在于避免并发活动发生优先级翻转(priority inversion)[4],因此必须提供具有防止优先级翻转的互斥和同步机制.

本文提出了一个面向对象的基于多线程的优先级继承协议锁算法,并从多方面分析了其特性;实现了一个具有固定优先级调度能力的实时线程库,并依此实现了符合实时 CORBA1.0 规范[5]的,具有固定优先级调度能力的实时 ORB.

2　研究现状

作为使用最广泛的两个操作系统,Windows 和 Solaris 都在其新版本中不断增强线程库的功能.比如 Windows2000 中就增加了线程优先级级别,但是其依然没有提供线程调度中的防优先级反转机制.Solaris 各个不同版本的线程库中,都有不同程度的改进.但是也只有在最新的 Solaris8 中才提供了防优先级反转的措施.虽然实时 Java 的规范仍然在制定之中,但目前的 Java 线程库都不提供防优先级反转的机制.实时操作系统中,也有一些并没有实现优先级的防反转机制,比如 μC/OS-II[6].

使用操作系统线程库构造能够满足特殊需求的线程库是很多系统采用的手段.这种线程库的开发有两个主要的方向:一是标准化,如 Pthreads Win32[7],致力于为 Win32 提供一个符合 Pthread 标准的线程库;二是面向对象封装,如 JTC[8](Java-like thread for C++),致力于为C++提供一个面向对象的 Java 风格的线程库.但是这些线程库没有考虑实时调度的需求,甚至使操作系统原本具有的某些实时功能在线程库中也无法体现.本文研究的线程库要提供面向对象的编程接口,提供严格的固定优先级调度,并且要避免调度过程中优先级反转的发生.

对固定优先级调度线程库来讲,关键是提供避免优先级翻转的同步和互斥机制.避免优先级翻转可以使用优先级封顶协议(priority celling)或者使用优先级继承协议[4](priority inheritance),本文使用优先级继承协议.

3　对象定义

本文将线程库中使用优先级继承协议来防止优先级翻转的互斥对象记为 RTMutex,其定义如下:

```
class RTMutex {
  ...
  MUTEX EntryMutex;
  BOOL IsLocked; //加锁标志
  int NormalPri; //锁所属线程的正常优先级
  int TempPri;
     //锁所属线程的当前临时优先级
  THREAD_HANDLE Owner;
    //锁的所属线程
  MUTEX StateMutex;
    //修改锁内部状态的互斥量
  public:
    void lock();
    void unlock();
  ...
};
```

因为要实现优先级继承协议,所以任何加锁动作都可能导致修改锁所属线程的优先级,所以锁必须记录所属的线程(Owner),并记录其优先级(NormalPri 和 TempPri).为了维护锁内部状态的一致性,使用 StateMutex 作为访问锁内部状态的互斥量.EntryMutex 和 StateMutex 可以是线程库提供的原始锁,如 Win32 下的 CRITICAL_SECTION,Pthread 线程库的 pthread_mutex_t,或者是根据线程库的原始锁构造的具有其它特性的锁,如根据原始锁构造的 FIFO 锁.RTMutex 提供加锁(lock)和解锁(unlock)两个操作.

4　加锁算法

优先级继承协议锁的加锁算法如算法 1 所示,

为方便描述,算法标记了序号.在以下的算法描述中,使用了如下的函数假设:

GetCurrentThread():获得当前线程标识(句柄);

GetThreadPriority (ThreadHandle):获得ThreadHandle表示的线程的优先级;

SetThreadPriority (ThreadHandle, Priority):将ThreadHandle线程的优先级设置为Priority.

算法 1. 优先级继承协议加锁算法

```
L1  while (TRUE){
L2    StateMutex.lock(); //保证查看和修改锁的内容是互斥的
L3    THREAD_HANDLE CurrentThrHandle=GetCurrentThread(); //获得当前线程句柄
L4    if (!IsLocked){ //当前没有线程加锁,可以获得锁
L5      IsLocked=TRUE; //设置加锁标记
L6      Owner=CurrentThrHandle; //记录锁的拥有者
L7      NormalPri=GetThreadPriority(Owner);
        //记录拥有锁的线程的正常优先级
L8      TempPri=NormalPri; //记录拥有锁的线程的当前优先级
L9      EntryMutex.lock(); //获得锁
L10     StateMutex.unlock(); //锁的内部状态修改完毕,其它线程可以查看和修改锁状态
L11     break; //加锁成功
L12   }
L13   else { //锁已经被其它线程获得
L14       int TmPri=GetThreadPriority(CurrentThrHandle); //获得当前线程优先级
L15       if (TmPri>TempPri){ //需要临时提升拥有锁的线程的优先级
L16       TempPri=TmPri; //记录拥有锁的线程的临时优先级
L17       SetThreadPriority(Owner,TempPri); //临时提升拥有锁的线程的优先级
L18       }
L19     StateMutex.unlock(); //锁的内部状态修改完毕,其它线程可以查看和修改锁状态
L20     EntryMutex.lock(); //等待锁被释放
L21     EntryMutex.unlock(); //重新开始尝试
L22   }
L23 }
```

4.1 加锁算法的正确性分析

从互斥、死锁、状态一致性,避免优先级翻转和公平性以及忙等待来分析算法的正确性.

4.1.1 互斥分析

在加锁算法中,是利用操作系统提供的锁来实现的(L_9),所以不会出现多个线程都加锁成功的情况.根据算法的流程,任何线程要能够加锁成功,必须首先获得内部锁StateMutex(L_2),内部锁也是操作系统提供的,显然不可能出现多个线程都获得内部锁的情况.

4.1.2 死锁分析

因为算法中只涉及两个锁:EntryMutex和StateMutex,如果出现死锁,下面两个条件必须同时出现:

条件 1. 某线程T_1拥有EntryMutex,而且要求获得StateMutex;

条件 2. 某线程T_2拥有StateMutex,而且要求获得EntryMutex.

可以证明条件1不会出现:

根据算法,线程要拥有EntryMutex是执行了L_9或者L_{20}.如果线程通过L_9获得EntryMutex,根据L_{10},它会立刻释放StateMutex,条件1不成立;而且在执行L_9时,T_1必然已经获得StateMutex,所以这时也不会出现满足条件2的线程,因而通过L_9获得EntryMutex不会导致死锁.

如果线程通过L_{20}获得EntryMutex,根据算法的L_{19},线程已经释放了StateMutex,条件1不会出现.

综上所述,加锁算法保证不会出现死锁.

4.1.3 状态一致性

优先级继承协议锁具有一系列的内部状态,算法必须保证锁内部状态的一致性.根据算法,一个线程在操作(读和写)锁状态的时候必须拥有内部锁StateMutex,这样就阻止了任何其它线程对锁状态的读和写.所以锁的内部状态不会出现不一致的情况,而且线程也不会对锁内部状态判断失误,因为对内部锁状态的读也是互斥的.

4.1.4 防优先级翻转

根据L_{14}~L_{18},如果线程加锁不成功,并且优先级比拥有锁的线程的优先级高,则会临时提升拥有锁的线程的优先级,满足优先级继承协议,可以防止优先级翻转.

加锁算法的防优先级翻转具有以下性质:

假设线程T获得了锁,T的优先级记为P,在T拥有锁的过程中,假设有n个线程依次要求获得锁,这些线程记为$T_1,T_2,\cdots,T_n$,优先级记为$P_1,P_2,\cdots,P_n$,则:

命题 1. 连续提升性质:

记有序序列$S=P_1,P_2,\cdots,P_n$,$C(S)$表示序列

中元素的个数，对 S 执行以下筛选：

① k 从 1 到 $C(S)$：如果 $P_k \leqslant P$，则从 S 中删除 P_k；

② 如果 S 不空，则 k 从 1 到 $C(S)-1$：如果 $P_k \geqslant P_{k-1}$，则从 S 中删除 P_{k-1}.

则线程 T 被提升的次数为 $C(S)$次.

命题 2. 最大提升性质：

记 $P_m=\max(P_1,P_2,\cdots,P_n)$，则线程 T 最终的优先级将是 P_m.

这两个性质的证明从略，这两个性质在对固定优先级系统进行静态行为分析时，非常有意义.

4.1.5　忙等待分析

当线程无法立即获得锁的时候，算法 L_{20}和 L_{21}使得线程在 EntryMutex 锁上等待，而不是循环测试，这使得算法的效率比较高. 尽管线程能否获得锁，依然要进行再一次的测试.

4.1.6　公平性分析

算法的公平性依赖于 EntryMutex 锁的特性. 当有线程释放锁的时候，在 EntryMutex 上等待的线程将有一个被唤醒，被唤醒的线程如果有机会执行，则它又会立刻释放锁，导致唤醒在 EntryMutex 上等待的另一个线程. 这使得在 EntryMutex 上等待的线程有可能被全部唤醒，重新开始竞争锁. 如果某个被唤醒的线程没有获得执行，则该线程以后的在 EntryMutex 上等待的线程将没有机会参与竞争锁.

实际上，可以在线程库的基础上，构造满足自己需求的 EntryMutex 锁. 在固定优先级调度系统的实现中，EntryMutex 锁使用优先级队列：即唤醒等待线程中优先级最高的线程，这是因为系统强调按照固定优先级调度线程，在这种系统中，低优先级的线程可能被饿死.

5　解锁算法

算法 2. 优先级继承协议锁解锁算法

```
U1   StateMutex.lock();
       //保证查看和修改锁的内容是互斥的
U2   THREAD_HANDLE currentThrHandle=
     GetCurrentThread(); //获得当前线程句柄
U3   IsLocked=FALSE; //设置解锁标记
U4   EntryMutex.unlock(); //解锁
U5   if (NormalPri != TempPri){
       //优先级被临时提升了
U6     THREAD_HANDLE PreviousOwner=Owner;
U7     int PreviousPri=NormalPri;
U8     StateMutex.unlock(); //释放内部锁
U9     SetThreadPriority(PreviousOwner,
         PreviousPri); //恢复优先级
U10  }
U11  else //优先级没有提升
U12    StateMutex.unlock().
```

5.1　解锁算法的正确性分析

假设只有拥有锁的线程才可以解锁，即线程在执行 unlock 之前，必须 lock 成功. 根据加锁算法的互斥性，假设保证了同时只有一个线程在解锁. 这个假设是合理的.

5.1.1　死锁分析

由于解锁的线程必须是锁的拥有者，所以解锁线程一定获得了 EntryMutex，所以解锁线程只可能满足条件 1，可以证明这时加锁的线程不可能满足死锁条件 2：

加锁线程要满足死锁条件 2，它必须执行加锁算法中的 L_9 或者 L_{20}. 但是执行 L_{20}时，它已经释放了 StateMutex，所以这时不会死锁. 如果加锁线程获得了 StateMutex，由于对 IsLocked 的访问是互斥的，解锁线程一定在 StateMutex 上等待，无法修改 IsLocked 的值，所以加锁线程不会执行到 L_9（加锁 L_4 的判断不会成功），所以也不可能死锁.

5.1.2　防优先级翻转

解锁时，线程如果发现优先级被临时提升了，它会将优先级再恢复到正常的值. 在优先级的恢复上，算法必须考虑到各种情况，如下的算法看起来更好，因为它减少了两次拷贝：

算法 3. 解锁算法（先恢复优先级，后释放锁）

```
…//U1～U4
U5   if (NormalPri != TempPri){
U6     SetThreadPriority(Owner, NormalPri);
U7     StateMutex.unlock();
U8   }
…//U11～U12
```

算法 4. 解锁算法（先释放锁，后调整优先级）

```
…//U1～U4
U5   if (NormalPri != TempPri){
U6     StateMutex.unlock();
U7     SetThreadPriority(Owner, NormalPri);
U8   }
…//U11～U12
```

算法 3 和算法 2 惟一的区别在于算法 2 先释放

锁,后调整优先级,算法 3 先调整优先级后释放锁.在固定优先级调度系统中,算法 3 会导致重新发生优先级翻转.如图 1,4 个线程 T_1,T_2,T_3,T_4,其优先级分别为 $P_1<P_2<P_3<P_4$,其中 T_1 和 T_4 有共享资源,T_1 先得到锁.启动顺序为 T_1,T_2,T_3 和 T_4.如果没有优先级继承,则会发生优先级翻转,执行顺序为 T_3,T_2,T_1 和 T_4.由于采用优先级继承协议锁,T_1 的优先级被 T_4 提升,所以它最先完成.当 T_1 解锁时,它先将自己的优先级调整为 P_1,然后准备释放 StateMutex,但是这时 T_1 的优先级最低,它马上被 T_3 剥夺,没有机会释放 StateMutex,使得 T_4 无法执行,第 2 次发生优先级翻转,导致执行顺序为 T_1,T_3,T_2 和 T_4,而不是正确的 T_1,T_4,T_3 和 T_2.图 1 是根据 4 个线程执行过程中记录的数据得到的执行示意图.

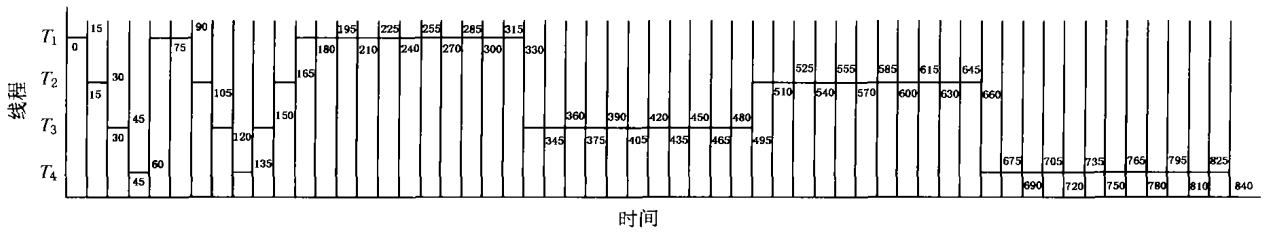

图 1 算法 3 下的线程执行示意图

解锁时必须在释放锁以后再调整优先级.但是如果先释放 StateMutex,再设置优先级,会发生互斥的问题,如算法 4 所示.U_7 对 Owner 和 NormalPri 的访问不是互斥的.考虑两个线程 T_1 和 T_2,T_1 按照算法 4 执行解锁,T_2 按照算法 1 执行加锁,当 T_1 执行完算法 4 的 U_6 以后,假设发生线程切换,这时 T_2 开始执行加锁,并且它加锁成功,这时 Owner 和 NormalPri 都会发生变化,而当 T_1 继续执行时,就会非法修改 T_2 的优先级,而无法正确恢复 T_1 的优先级!为此,执行两个拷贝,将 Owner 和 NormalPri 拷贝到线程自己的栈中,然后释放 StateMutex,尽管恢复优先级的时候,这两个变量可能发生了变化,但是解锁线程使用的是栈里的变量,所以不会访问锁变量,保证互斥.图 2 是 4 个线程的执行示意图.

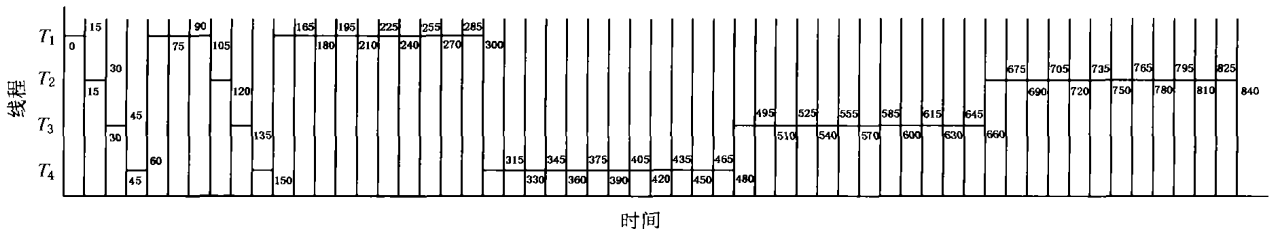

图 2 算法 2 下的线程执行示意图

5.1.3 互斥分析

和加锁算法的互斥性分析相同,并结合第 5.1.2 节分析,可知解锁算法满足互斥要求.

5.1.4 状态一致性

和加锁算法的状态一致性分析相同,并结合第 5.1.2 节分析,可知解锁算法满足状态一致性要求.

5.1.5 公平性

公平性依赖于 EntryMutex 锁的特性,在固定优先级调度系统中,使用按照优先级高低顺序解锁是比较合适的,它可以保证整个系统具有一致的调度特性,在其它要求的系统中,可以使用其它特性的锁,比如 FIFO 锁,或者随机锁等等.

6 优先级协议锁的性能

优先级继承协议锁采用操作系统或者线程库的锁来构造,必然带来性能上的损失,表 1 是在 Windows2000 和 Solaris8 操作系统上的测试比较.测试项目包括一般锁、优先级继承协议锁和按优先级解锁的优先级继承协议锁(Win32 下使用 QueryPerformanceCounter 方法获取时间,否则精度难以满足).表中时间是分别执行 1000,10000 和

100000 次加锁和解锁的时间.

表 1　优先级协议锁的性能比较　　ms

次数	一般锁 (Win32/Solaris)	优先级继承协议锁 (Win32/Solaris)	使用按优先级解锁的优先级继承协议锁(Win32/Solaris)
1000	0.027/0.930	3.232/5.726	8.945/10.849
10000	0.260/9.014	28.886/66.520	87.698/117.868
100000	2.584/95.626	287.725/574.443	873.074/1114.914

可以看到使用优先级继承协议锁，会带来较大的性能下降，所以在系统不要求按照固定优先级调度时，最好使用一般锁.

7　应用和进一步的工作

本文使用互斥对象 RTMutex，构造了防止优先级翻转的信号量和管程(moniter). 基于本文的算法，在 Win32 平台和多种 Unix 平台(基于 Pthread 线程库)上，实现了一个具有固定优先级调度能力的线程库，使用该线程库实现了符合实时 CORBA1.0 规范的实时 ORB，规范中定义的 Mutex 接口，可以使用本文的 RTMutex 方便地实现. 目前，业界对实时 CORBA 的研究居多，但符合规范的实时 ORB 很少.

进一步的工作包括扩展线程库功能，使其支持其它的调度策略，并设计对实时系统行为的静态和动态分析方法，以及实现具有动态调度能力的实时 ORB.

参　考　文　献

1　S Khanna *et al*. Realtime Scheduling in SunOS 5.0. The USENIX Winter Conf, San Francisco, CA, 1992

2　Object Management Group. Realtime CORBA--A White Paper-Issue 1.0. OMG Realtime PSIG, Tech Rep: ORBOS/96-09-01, 1996

3　骆斌，费翔林. 多线程技术的研究与应用. 计算机研究与发展，2000，37(4)：407～412 (Luo Bin, Fei Xianglin. Study and application of multithread technology. Journal of Computer Research and Development (in Chinese), 2000, 37(4): 407～412)

4　Lui Sha, Ragunathan Rajkumar, John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans on Computers, 1990, 39(9): 1175～1185

5　Object Management Group. The common object request broker: Architecture and specification, 2.4.1 edition. 2000. http://www.omg.org

6　Jean J Labrosse. μC/OS-II: The real time kernel. R & D Books. Lawrence, Kansas, USA, 2000

7　POSIX Threads for Win32, Open Source. http://sources.redhat.com/pthreads-win32/index.html

8　Java-Like Threads for C++, IONA Company http://www.iona.com/devcenter/orbacus/jtc.html

郭长国　男，1973 年生，博士研究生，主要研究方向为分布实时计算.

周明辉　女，1974 年生，博士研究生，主要研究方向为分布容错计算.

王怀民　男，1962 年生，教授，博士生导师，主要研究方向为智能软件、分布计算和网络信息安全技术.

许　勇　男，1974 年生，硕士研究生，主要研究方向为线程库和线程调度.