供动态无锁数据结构使用的资源窃取型无锁内存池

刘 恒, 杨小帆
(重庆大学 计算机学院, 重庆 400044)

摘 要: 动态内存管理的问题对无锁动态数据结构的性能尤为关键,因为多线程环境下的动态内存管理涉及开销较高的同步操作。提出一种构建用于动态无锁数据结构的内存池的方法来减少动态内存使用和与之相伴的动态内存管理开销。该方法通过平衡线程的动态内存消耗来减小内存开销,利用本方法构建的内存池基于线程私有的支持节点窃取的无锁循环队列。本方法具有以下优点:a)用本方法构建的内存池是无锁的;b)能够平衡线程的堆内存消耗;c)可以方便地与动态无锁数据结构集成。实验结果显示,用该方法构造的资源窃取型内存池扩展性较强,且能够在高负载下有效降低无锁数据结构的堆内存消耗和操作执行时间;平衡算法在很大程度上决定内存消耗量,内存池在高负载下的扩展性也受到它所用的数据结构自身多线程访问性能的影响。

关键词: 资源窃取; 无锁内存池; 无锁; 动态无锁数据结构
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2012)10-3772-04
doi:10.3969/j.issn.1001-3695.2012.10.043

Resource stealing lock-free memory pool for dynamic-sized lock-free data structures

LIU Heng, YANG Xiao-fan
(*College of Computer Science*, *Chongqing University*, *Chongqing* 400044, *China*)

Abstract: Dynamic memory management in a multi-threaded environment involes expensive synchronization cost, making it a vital issue for the performance of dynamic lock-free data structures. This paper proposed a scheme for the lock-free implementation of a memory pool suited to dynamic-sized lock-free data structures to reduce the associated dynamic memory consumption and dynamic memory management cost. This scheme reduced dynamic memory consumption associated with the shared lock-free data structures by balancing threads dynamic memory consumption, it was based on thread local lock-free circular queue that supported node stealing method. This scheme possesses three outstanding advantages: a) the memory is wait-free, b) it can balance the threads' consumption of dynamic memory, c) its integration with existing dynamic-sized lock-free data strucutures is extremely easy. Experimental results show that this scheme is highly scalable and can effectively reduce the average execution time of dynamic-sized lock-free data structures' operations under heavy load. The amount of dynamic memory consumption is mostly affected by the balancing strategy and the scalability of the memory pool under high load is also affected by the underlying data structures.

Key words: resource stealing; lock-free memory pool; lock-free; dynamic-sized lock-free data structures

0 引言

对于许多 C/C++ 程序来说,动态内存管理可能是最为普遍和代价高昂的操作之一,这种情况对于在多道环境下运行的程序尤甚:从堆中分配内存或者将内存归还都需要在不同进程/线程间同步以确保正确性。研究显示,有些程序甚至会将其三分之一的执行时间花在动态内存管理上[1~3],而这一情况在无锁数据结构[4~8]上则更为严重,因为与它们相关的大部分操作都集中于插入新节点和删除旧节点——分配和释放堆内存,随着共享同一数据结构的线程增多,随之而来由于对内存使用导致的同步操作开销急剧上升,大大减少数据结构的吞吐量和操作平均执行时间。在如今这个多核处理器广为流行的时代,随着处理器核数的增多,共享同一数据结构的线程数量也在持续上升,使得动态内存管理的开销在无锁数据结构操作开销中占的比例越来越大。

本文将一种基于资源窃取的全局性资源平衡方法引入无锁数据结构内存池的设计,通过在竞争线程间均衡分布内存资源来减少与共享数据结构相关的堆内存使用和操作执行时间。

1 现有技术

供无锁数据结构使用的内存池通常基于自由链表,当线程需要向共享数据结构中插入节点时,它从自由链表中取得一个节点,只有当自由链表为空时,线程才会直接向操作系统申请新节点;相反,当线程从共享数据结构中删除一个节点后,它将删除后的节点放入自由链表以备将来之需,只有当链表中节点数量超过限额之后,线程才会将节点直接归还给操作系统。

收稿日期: 2012-03-17; **修回日期**: 2012-04-26

作者简介:刘恒(1988-),男,江苏宿迁人,硕士研究生,主要研究方向为并行与分布式计算、并发数据结构等(liuheng65@gmail.com);杨小帆(1964-),男,教授,博导,主要研究方向为并行计算、算法设计、差分方程等.

在基于公共自由链表[9]实现的内存池中,所有线程共享一个公共自由链表,这种实现方式虽然比较简单,但是缺点颇多,特别是在当竞争线程数量较大的情况下,公共自由链表成为串行瓶颈——对公共自由链表的访问涉及大量加/解锁操作会显著增加操作开销。

使用线程私有自由链表[10]的内存池可以克服上述缺点。线程访问各自私有的自由链表,从而免于对共享资源的并发访问而导致大量的同步操作。线程私有自由链表的缺点在于,尽管各个线程都是对动态数据结构执行相似甚至相同的操作序列,但由于种种原因(如操作系统调度、等待其他资源、线程同步)它们对内存的使用情况却不尽相同。一个线程用尽其自由链表中的节点时,其他线程却可能仍有富余,从而导致线程对资源使用的不均衡。

2　资源窃取型无锁内存池

使用资源窃取[4,11,12]技术,耗尽私有自由链表中的自由节点的线程可以尝试从其他线程的私有队列窃取自由节点,通过在不同私有队列间腾挪实现线程对内存消耗的平衡。基于资源窃取技术的无锁内存池(memoryPool)算法提供了两个方法:getNewNode 和 returnNode。GetNewNode 用来从内存池中返回一个新节点(bode);而 returnNode 则将一个节点返回给内存池。本文使用线程私有的支持节点窃取的无锁循环队列(nodeStealLFCircularQueue)[13,14]来存储自由节点以减小资源窃取操作的开销。内存池被实现为一个包含若干无锁循环队列的数组,共享数据结构的每个线程都有一个属于自己的私有队列,用于存储自由节点。同时,线程的私有队列也可以被其他线程访问,以窃取节点。下面给出了本文所涉及结构的简要定义。

```
类型定义　MemoryPool
STRUCT MemoryPool
  NodeStealLFCircularQueue * PrivateQueues;
  //线程私有队列数组
  int MaxTries; //最大尝试窃取次数
  int NumThreads; //线程数
END STRUCT
类型定义　NodeStealLFCircularQueue
STRUCT NodeStealLFCircularQueue
  int Bottom; //队列底
  long Top; //队列顶
  int Size; //队列大小
  Node * FreeNodes; //存放自由节点的数组
END STRUCT
类型定义　Node
STRUCT Node
  Data DataElement; //数据
  Node * Next; //下一项
END STRUCT
```

2.1　获取新节点(getNewNode)

线程调用 getNewNode 方法从内存池获取一个新节点,getNewNode 方法首先获取调用者线程的 threadID,并尝试从该线程的私有队列中获取一个自由节点,若失败,也即该线程的私有队列为空,则试着从其他线程的私有队列中窃取一个节点,若再次失败,则向操作系统申请一定数量的新节点填充到自己的私有队列中,并返回一个自由节点给当前线程。事实上,为提高性能,窃取操作可多次进行。GetNewNode 方法的具体步骤如下:

a)获取当前线程的私有队列,尝试从中获取一个自由节点,若成功则设置该节点对应域的值并将其返回,否则置当前尝试次数为零并转到步骤 b)。

b)若当前尝试次数超过尝试次数上限且仍未窃取到自由节点,则转到步骤 e),否则随机选定一个目标线程并转到步骤 c)。

c)若目标线程的线程 ID 与当前线程的 ID 相等,则转到步骤 b),否则转到步骤 d)。

d)试着从目标线程的本地队列中窃取一个自由节点,若窃取成功,则设置该节点对应域的值并将其返回,不成功则转到步骤 b)。

e)向操作系统申请一个节点,设置它对应域的值,并将其返回给调用线程。

下面给出了 getNewNode 方法的伪代码实现。

算法 1　getNewNode

```
input: MemoryPool, DataElement, CurrentTID.
output: result.
LocalQueue = MemoryPool. PrivateQueues[CurrentTID];
//获取线程私有队列
result = Dequeue(LocalQueue);//尝试获取一个自由节点
if result = = NULL
  AttemptionCount = 0;//设置尝试次数
  while result = = NULL AND AttemptionCount + + < MemoryPool. MaxTries
    VictimID = rand( ) % MemoryPool. NumThreads;
      //获取目标线程 ID
    if VictimID = = CurrentTID;
      //目标线程 ID 与当前线程 ID 相同,重试
    continue;
    else //尝试从目标线程私有队列中窃取一个节点
    VictimLocalQueue = MemoryPool. PrivateQueues[VictimID];
    result = StealNode(VictimLocalQueue);
      //调用 StealNode 方法以窃取节点
    end if
  end while
end if
if result = = NULL
    result = malloc(sizeof(Node));
      //窃取失败,向操作系统申请一个新节点
end if
result. DataElement = DataElement; //设置节点对应域的值
result. Next = NULL;
return result //返回节点
```

2.2 返还节点(returnNode)

ReturnNode 方法将一个节点返回给内存池,若调用线程的私有队列未满,则将节点放入调用线程的私有队列;若调用线程私有队列已满,则将节点直接释放,返还给操作系统,具体步骤如下:

a)若当前线程的私有队列容量已满,则将可以安全释放的节点直接释放,返还给操作系统,否则转步骤 b)。

b)将该节点放入当前线程的私有队列之中。

下面出了 returnNode 方法的伪代码实现。

算法 2 returnNode

```
input: MemoryPool, Node, CurrentTID.
output: none.
LocalQueue = MemoryPool. PrivateQueues[ CurrentTID ]
if LocalQueue. Bottom - LocalQueue. Top > = LocalQueue. Size - 1
    free( Node) ;
    //私有队列容量已满,将节点释放,返还给操作系统
else
    Enqueue( LocalQueue, Node) ;
    //尚有空间,将节点放入私有队列
end if
```

2.3 节点窃取无锁循环队列(nodeStealingLFCircularQueue)

为支持节点窃取操作(StealNode),需要对无锁循环队列作修改,以使得 Enqueue/Dequeue 和 StealNode 操作仍然无锁且足够高效,修改后的无锁循环队列如下:

a)Dequeue 方法返回一个自由节点, Dequeue 只可以被拥有该私有队列的线程调用,因而只有当队列中仅剩一个自由节点时,该方法才需要使用 CAS[4] 操作来尝试获取该节点,其他时候该方法只是简单返回队列底部(bottom)的自由节点。

b)StealNode 方法试图从队列中窃取一个自由节点,该方法使用 CAS 操作来执行窃取操作,以免与 Dequeue 方法或其他线程在该队列上的并发窃取操作产生冲突。StealNode 总是试图获取位于队列顶部(top)的节点。

c)Enqueue 方法将一个可以安全释放的自由节点存入本队列,该方法总是将新节点存入队列的底部。

下面给出了 Dequeue、StealNode、Enqueue 方法的实现。

算法 3 Dequeue

```
input: LocalQueue.
output: result.
LocalQueue. Bottom = LocalQueue. Bottom - 1;//设置新队底
oldTop = LocalQueue. Top; //获取队顶的值
leftover = LocalQueue. Bottom - oldTop;
  //获取队列中剩余自由节点数量
if leftover > 0 //若剩余数量大于零,则获取队列底部的自由节点
result = LocalQueue. FreeNodes [ LocalQueue. Bottom% LocalQueue. Size];
else if leftover = = 0
//剩余数量为零,此时队列顶部仍可能有一个自由节点
if CAS( &LocalQueue. Top, oldTop, oldTop + 1) = = TRUE
  //尝试获取队顶节点
    result = LocalQueue. FreeNodes[ LocalQueue. Bottom% LocalQueue. Size];
    else//获取队顶节点失败,令队底的值为队顶的值加 1
      result = NULL;
      LocalQueue. Bottom = oldTop + 1;
    end if
    else//剩余数量小于零,令队底的值等于队顶的值
    result = NULL;
    LocalQueue. Bottom = oldTop;
  end if
  RETURN result;
```

算法 4 StealNode

```
input: VictimQueue.
output: result.
oldTop = VictimQueue. Top; //获取队顶的值
leftover = VictimQueue. Bottom - oldTop;
    //计算队列中尚存自由节点数目
if leftover < = 0 //队列为空,窃取失败
    result = NULL;
else /* 队列非空,用 CAS 操作修改队顶,若成功,则取得当前队顶处节点 */
if CAS( &VictimQueue. Top, oldTop, oldTop + 1) = = TRUE
    result = VictimQueue. FreeNodes[ ( oldTop + 1) % VictimQueue. Size];
else//CAS 操作失败,窃取失败
    result = NULL;
end if
end if
return result;
```

算法 5 Enqueue

```
input: LocalQueue, Node.
output: none.
LocalQueue. FreeNodes [ LocalQueue. Bottom% LocalQueue. Size ] = Node;//存入节点
LocalQueue. Bottom = LocalQueue. Bottom + 1;
    //更新队底的值
```

3 实验和结果分析

3.1 实验方法

本文将资源窃取型内存池应用于使用风险指针的 Michael&Scott 无锁队列[15] 和用 IBM 自由链表算法构建的无锁栈[16],分别测量使用资源窃取型内存池和使用普通线程私有队列的内存池时,队列和栈的平均内存申请次数和操作平均执行时间。实验平台是英特尔双核(2.66 GHz 4 个硬件线程结构)I5-520 处理器,实验目的是显示资源窃取型内存池的性能和可扩展性。

实验数据采用 Herlihy 等人[17] 的方法取得,在测试程序的一次运行中,每个线程在共享队列(栈)上执行 100 万个随机操作,入队/出队(入栈/出栈)操作的总体比例均为 1:1。向操作系统申请内存的平均次数,用线程执行 100 万次随机操作过

程中调用 malloc 函数的平均次数来衡量,操作平均执行时间用线程执行 100 万次随机操作期间的入队/出队(入栈/出栈)平均执行时间来衡量。每个实验均运行 5 次,本文取 5 次结果的平均值作为最终结果。

3.2　实验结果与分析

第一个实验测量平均内存申请次数。如图 1 显示,在共享同一个队列的竞争线程数量从 1 增加到 64 的过程中,使用资源窃取型内存池的无锁队列的平均内存申请次数持续降至低于使用普通线程私有队列线程池时的 30%,类似的结果和趋势也可以从图 2 中得出。

图 1 和 2 显示,使用资源窃取型内存池时,平均内存申请次数持续降低,直至竞争线程数量增加到 64(硬件线程结构数量的 8 倍)。在多道程序道数在 2 ~ 8(8 ~ 32 个线程)期间,使用资源窃取型内存池的队列和栈的平均内存申请次数明显降低,竞争线程数量越大,平均内存申请次数越少。其原因包括:

a)竞争线程数量越大,通过 StealNode 方法成功窃取到节点的几率就越大,从而有效降低了线程的内存申请次数。

b)某线程的节点被偷窃也反过来导致该线程的下一次节点释放操作有更大的可能将可以安全释放的节点放入私有队列。

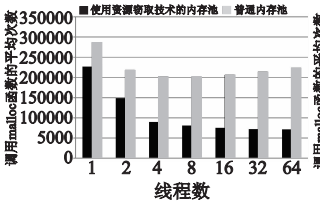

图1　使用不同内存池的队列分别执行100万次随机操作,该过程中调用malloc函数的平均次数统计

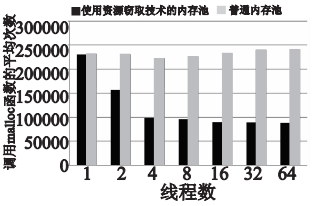

图2　使用不同内存池的栈分别执行100万次随机操作,在该过程中调用malloc函数的平均次数统计

第二个实验测量操作的平均执行时间。如图 3 所示,随着竞争线程数量从 1 增加到 64 时,使用资源窃取型内存池的队列,其入队/出队操作的平均执行时间和使用普通线程私有队列的内存池时的平均执行时间之比也随之降低到低于 50%,从图 4 也可以观察到相似的结果和趋势。其原因在于,竞争线程数量越大,使用资源窃取型内存池时的平均内存申请次数越少,而由于内存申请开销较大,减少的内存申请次数就反映为操作平均执行时间的显著降低。

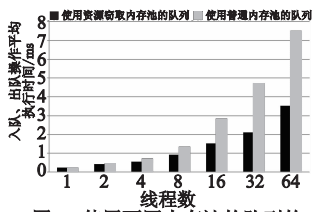

图3　使用不同内存池的队列的入队、出队操作平均执行时间统计

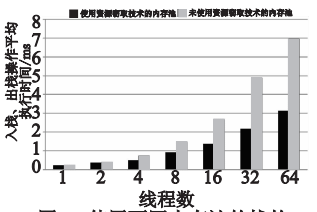

图4　使用不同内存池的栈的入栈、出栈操作平均执行时间统计

4　结束语

本文提出了构建资源窃取型无锁内存池的方法。通过使用线程私有且支持节点窃取的无锁循环队列来平衡线程和共享数据结构相关的堆内存消耗。实验结果表明,该实现方式提供了较高的性能且扩展性较强,并可在高负载下持续提供良好的扩展性。后续工作将集中于设计更高效的平衡算法和相关数据结构,以进一步降低堆内存消耗和相关无锁数据结构的操作执行时间。

参考文献:

[1] WILSON P, JOHNSTONE M, NEELY M, *et al*. Dynamic storage allocation: a survey and critical review[C]//Proc of International Workshop on Memory Management. London: Springer-Verlag, 1995: 1-116.

[2] LI Wen-tong, MOHANTY S, KAVI K. A page-based hybrid (software-hardware) dynamic memory allocator[J]. IEEE Computer Architecture Letters, 2006, 5(2): 13.

[3] ZORN B, GRUNWALD D. Empirical measurements of six allocation-intensive C programs[J]. ACM SIGPLAN Notice, 1992, 27(12): 71-80.

[4] HERLIHY M, SHAVIT N. The art of multiprocessor programming[M]. San Francisco: Morgan Kaufmann, 2008.

[5] VALOIS J. Lock-free data structures[D]. [S.l.]: Rensselaer Polytechnic Institute, 1995.

[6] KOGAN A, PETRANK E. Wait-free queues with multiple engquers and dequeuers[C]// Procc of the 16th Annual ACM Symposium of Principles and Practice of Parallel Programming. 2011: 223-233.

[7] SHALEV O, SHAVIT N. Split-ordered lists: lock-free extensible hash tables[J]. Journal of the ACM, 2006, 53(3): 379-405.

[8] ELEEN F, FATOUROU P, RUPPERT E, *et al*. Non-blocking binary search trees[C]//Procc of the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. New York: ACM Press, 2010: 131-141.

[9] WEISS M. Data structure and algorithm analysis in C[M]. [S.l.]: Pearson, 2010.

[10] HERLIHY M, LUCHANGCO V, PMARTIN P, *et al*. Nonblocking memory management support for dynamic-sized data structures[J]. ACM Trans on Computer Systems, 2005, 23(2): 146-196.

[11] BLUMOFE R D, LEISERSON C E. Scheduling multithreaded computations by work stealing[J]. Journal of the ACM, 1999, 46(5): 720-748.

[12] AGRAWAL K, LEISERSON C E, HE Yu-xiong, *et al*. Adaptive work-stealing with parallelism feedback[J]. ACM Trans on Computer Systems, 2008, 26(3): 1-32.

[13] CHASE D, LEV Y. Dynamic circular work-stealing deque[C]//Proc of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures. New York: ACM Press, 2005: 21-28.

[14] HENDLER D, SHAVIT N. Non-bolocking steal-half work queues[C]//Proc of the 21st Annual ACM Symposium on Principles of Distributed Computing. New York: ACM Press, 2002: 280-289.

[15] MICHAEL M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes[C]//Proc of the 21st Annual ACM Symposium on Principles of Distributed Computing. New York: ACM Press, 2002: 21-30.

[16] MICHAEL M. Hazard pointers: safe memory reclamation for lock-free objects[J]. IEEE Trans on Parallel and Distributed Systems, 2004, 15(6): 491-504.

[17] HERLIHY M, LEV Y, LUCHANGCO V, *et al*. A provably correct scalable concurrent skip list[C]//Proc of the 10th International Conference on Principles of Distributed Systems. 2006: 1-15.