改进型锁无关双端队列的设计与实现①

杨东升[1]，张连法[1,2]

[1](中国科学院 沈阳计算技术研究所，沈阳 110168)
[2](中国科学院 研究生院，北京 100049)

摘 要：高性能实时系统对系统性能、确定性和容错性有着更高的要求。非阻塞同步在任务同步方面满足要求，实现方法之一就是设计锁无关数据结构。介绍了设计锁无关数据结构算法的关键技术，通过对已有算法不足的分析提出了一种改进型的锁无关双端队列算法，介绍了对该算法的实验分析和实际应用。实验结果表明，该算法提高了访问双端队列的执行速度，并避免了多任务间同步引发的死锁、优先级逆转、低容错性等缺点。

关键词：非阻塞同步；锁无关；双端队列；RTAI

Design and Realization of the Improved Lock-free Double-Ended Queue

YANG Dong-Sheng[1], ZHANG Lian-Fa[1,2]

[1](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)
[2](Graduate University, Chinese Academy of Sciences, Beijing 100039, China)

Abstract: High-performance real-time system has higher requirement for system performance, certainty and fault tolerance. Non-blocking synchronization meets the requirement in task synchronization, one of the methods of achieving non-blocking synchronization is to design lock-free data structure. This paper introduces the key technology for designing lock-free data structure algorithm, presents an improved lock-free double-ended queue algorithm by the analysis of the existing algorithm shortage, and introduces the experimental analysis and practical application of the algorithm. Experimental results show that the algorithm improves the execution speed for accessing double-ended queue, and avoids deadlocks, priority inversion and low fault tolerance caused by multitask synchronization.

Key words: non-blocking synchronization; lock-free; double-ended queue; RTAI

采用多核/多处理器硬件平台的高档数控系统需要相应的高性能实时操作系统的支持。为了满足高档数控系统对实时性与系统服务的需求，实时操作系统需要更高的硬实时性和可靠性，并避免优先级逆转、死锁的发生。基于相互排斥的阻塞同步存在任务阻塞、死锁、优先级逆转、低容错性等缺点，不符合实时系统的要求，因此阻塞同步引发的缺点成为亟需解决的问题。

非阻塞同步利用处理器提供的特殊指令实现并保证访问共享数据的正确性。非阻塞同步方法可以避免在多个任务共享同一资源时发生死锁和优先级逆转，提高系统的可靠性与实时性，从而避免了由阻塞同步引发的诸多缺点。

现在国外的主要研究方向为非阻塞同步数据结构算法设计[1-4]、算法正确性理论证明[5]、算法性能测试[2]、非阻塞同步数据结构集成的软件库[6,7]等。将非阻塞同步数据结构应用到多核多处理器环境中，以实现系统的高性能、可扩张性、稳定性。

设计锁无关同步数据结构是实现非阻塞同步的方法之一。锁无关同步保证了整个系统会在有限步骤内完成对同一共享数据的某些操作[7]。虽然存在任务重复执行某一操作无限次的可能，但保证在有限的步骤内至少有一个任务完成它的操作，即相互同步的多个任务在整体上是不断前进的。

① 基金项目:国家科技重大专项(2011ZX04016-071)
收稿时间:2011-07-12;收到修改稿时间:2011-08-22

本文介绍了锁无关算法实现的关键技术，对文献[1]中提出的锁无关双端队列算法进行改进后提出了一种改进型的锁无关双端队列算法，修正了原有算法中的错误，对该算法进行了实验测试，表明该算法的性能有明显提高。

1 锁无关算法实现关键技术

1.1 锁无关算法基本结构

锁无关算法实现了多个任务不采用锁机制就可以同时操作的基本数据结构，如栈、队列。每种数据结构包括多个处理数据的操作，如队列包括入队、出队两个基本操作。每个操作包括三个主要部分：预处理部分、修改共享数据部分，后续处理部分。预处理部分包括增加访问任务计数、申请资源等。修改共享数据部分有一个潜在的无限循环，循环中获取共享数据状态并试图修改共享数据。如果修改成功则跳出循环，否则重新执行。在共享数据没有访问冲突的情况下，循环只需一次；当多个任务并行访问共享数据时，只有一个任务执行成功，其他任务需重新执行。后续处理部分包括减少访问任务计数、释放资源等。

1.2 原子指令

设计锁无关数据结构采用的主要原子指令包括CAS(Compare-and-Swap)、LL/SC 指令，原子加减 1 指令。大多数体系结构实现了此类指令，因此设计锁无关双端队列中采用了 CAS 指令和原子加减 1 指令。

CAS 指令主要形式为 CAS(&val，old，new)，val 等于 old 时赋值 val 为 new，否则失败。CAS 指令可以在一条指令中完成对内存数据的测试和更新，但必须解决“ABA 问题”。一个任务在改变共享变量之前先读取它的值为 A，之后另一个任务修改为 B 后又改回 A，此时 CAS 指令正确执行，却违背了对共享变量的读与写之间不能发生其他任务写操作的原则[2]。最简单有效的方法是使用修改计数记录共享变量的修改次数，CAS 指令判定变量和修改计数都改变时才能更新。这种方法并未完全避免“ABA 问题”，但将其发生的概率降低到极低的水平，可以忽略不计。

1.3 优化屏障问题

编译器编译源代码时会对源代码进行优化，将源代码的指令和数据进行重排序，以适合于 CPU 的并行执行。同步相关的代码必须避免指令和数据的重新排序，否则将产生严重的错误。优化屏障避免编译器的重排序优化操作，保证编译程序时在优化屏障之前的指令不会在优化屏障之后执行。在设计锁无关数据结构的过程中需采用优化屏障避免编译器的优化问题。

2 改进型锁无关双端队列

2.1 原有算法的不足

文献[1]中介绍了采用 CAS64 原子指令实现的基于双链表的双端队列算法，CAS64 可以操作 8 个连续字节的 CAS 指令。双端队列结构包括左指针、右指针和状态位，共占用 8 个字节，可由 CAS64 原子指令在一条指令内修改。数据节点结构包括前指针、后指针及相应的修改计数和数据域。受制于 CAS64 操作数据位数的限制，双端队列结构中并没有处理“ABA 问题”的修改计数，因此该算法存在导致队列混乱的风险。

图 1 是该算法产生错误的一个实例，具体算法参考文献[1]。

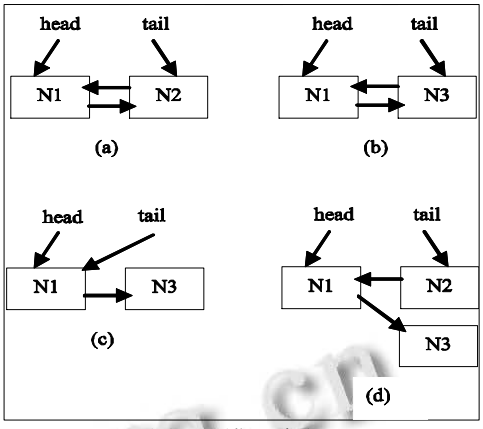

图 1　错误实例

双端队列结构中的状态有三种情况：稳定态，即所有节点都符合后节点的前节点和前节点的后节点是本身；左（右）入队状态，在双端队列左（右）侧刚插入一个结点，还未做稳定化处理。每一个任务在试图修改双端队列时，如果处于非稳定状态就需先进行稳定化处理。

根据图 1 分析产生错误的步骤如下：

(1) 在图 1(a)中，一个任务完成稳定化处理并修改状态位，另一个任务也完成稳定化处理并将要修改状态位为稳定状态。

(2) 结点 N2 右侧出队，结点 N3 右侧入队，处于稳定状态。

(3) 结点 N3 出队，处于稳定状态。

(4) 结点 N2 入队，处于右侧入队状态。此时(1)中第二个任务设置状态位，因为双端队列结构的值此

时与(1)时的值完全相同，CAS64 指令成功执行，致使图 1(d)的状态称为稳定状态，显然是错误的。

操作系统中的内存管理程序或为处理锁无关数据结构而设计的内存管理程序都倾向于将刚释放的内存重新分配，因此上述错误产生的可能性非常大。

2.2 改进型算法的解决方法

解决上述错误需在双端队列结构中添加修改计数，并满足 CAS64 指令只能同时修改 8 个连续字节和内存中字节对齐的限制。原算法中不管那一侧的操作都必须同时修改两侧指针，实际上除队列空时入队一个结点和一个结点时出队两种情况外只有一个指针改变，对另一侧的修改是多余的。因此每一端的操作只需要修改相应侧的指针和状态位即可。为解决两种特殊情况，加入了空状态(EMPTY)和正添加第一个结点(FIRST)两个状态并使队列中始终存在一个结点，由状态位表明只有一个结点的情况下节点中的数据是否有效。

主要数据类型：

```
Define statusType {STABLE,RPUSH,LPUSH,EMPTY,FIRST}
Deques{ Node *left;unsigned int:29 count;unsigned int:3 status ;Node *right;};
Node{Node*prev; unsigned int pcount;Node*next; unsigned int ncount;ValueType value;};
```

双端队列结构中有 12 字节组成，两侧指针共用一个修改计数，操作左侧时修改前 8 个字节，操作右侧时修改后 8 个字节。

2.3 改进型算法

双端队列包括左入队、左出队、右入队和右出队等四种操作。以右入队和右出队简要说明算法的实现。

初始化双端队列为一个结点，状态位为空。如下为右入队操作。

```
int deques_pushright(Deques_t*Q, ValueType value){
访问共享数据结构任务数加 1
while(1){
<l,c,s,r> = *Q //读取双端队列值
if s==EMPTY { //状态为空
if CAS64(&(Q->count),<c,s,r>, <c+1,FIRST,r>){
第一个结点赋值
c=Q->count
CAS64(&(Q->count)),<c,FIRST,r>,<c+1,STABLE,r>)
if node!=NULL
存储 node 结点
操作成功，跳出循环
}}
else if s==FIRST //其他任务正插入第一结点
continue
else if s==STABLE{ //稳定状态
if node==NULL
获得结点并赋值
node->prev=r
if CAS64(&(Q->count),<c,s,r>,(c+1,RPUSH,no de){
StabilizeRight(Q,l,c+1,node); //稳定化处理
操作成功，跳出循环
}}
else Stabilize(Q,l,c,s,r)//稳定化处理
}
访问共享数据结构任务数减 1,为 0 时启动内存回收，返回。
}
```

首先读取双端队列值，根据双端队列的状态进行不同的处理：1) 状态为空时，修改状态为正添加第一个结点，为第一个结点赋值，修改状态为稳定；2)状态为正添加第一个结点时，循环等待；3)状态为稳定时，申请新结点并赋值，插入到队列后进行稳定化处理；4)状态为左添加或右添加状态时，先进行稳定化处理。

如下为右出队操作。

```
int Deques_popright(Deques*Q, ValueType *value){
访问共享数据结构任务数加 1
while(1){
<l,c,s,r> = *Q //读取双端队列值
if s==EMPTY||s==FIRST
队列空，跳出循环
else if(l==r){
读取第一个结点的数据
if CAS64(&(Q->count),<c,s,r>,<c+1, EMPTY, r>)
操作成功，跳出循环
}
else if s==STABLE{
prev=r->prev;
if CAS64(&(Q->count),<c,s,r>,<c+1,s,prev>) {
读取删除结点中的数据，并存储结点
操作成功，跳出循环
}}
```

else Stabilize(Q,l,c,s,r) //稳定化处理

}

访问共享数据结构任务数减 1,为 0 时启动内存回收，返回。

}

首先读取双端队列值，根据双端队列的状态进行不同的处理：1) 状态为空或正添加第一个结点时，返回队列为空；2)状态为非空且只有一个结点时，读取数据值并设状态为空；3)状态为稳定时，删除一个结点，读取数据值并存储空闲结点；4)状态为左添加或右添加状态时，先进行稳定化处理。

每个操作根据双端队列的状态选择不同的处理，操作成功时则跳出循环，否则重新执行。算法中的原子指令保证对共享数据进行不可中断地修改从而避免并发访问共享数据出现的数据不一致风险。

2.4 正确性证明

锁无关算法的正确性包括安全性、可线性化、锁无关性。

2.4.1 安全性

安全性是指保证算法符合具体数据结构的操作要求，如队列出队必须对队头进行操作。通过分析算法中改变双端队列状态的语句是否符合双端队列的操作要求得到算法的安全性。证明过程如下：1) 定义双端队列的操作要求；2) 分析算法中改变双端队列状态的语句是否符合操作要求。按照此方法分析，本算法具有安全性。

2.4.2 可线性化

可线性化是指算法中每一个操作都会在一个特殊点生效[2]，实现不同操作的有序化。每个操作中跳出循环前的原子操作语句即可看做相应操作的生效点。利用 CAS 指令完成对双端队列状态的修改，保证多任务访问共享数据时数据的一致性。因此本算法是可线性化的。

2.4.3 锁无关性

锁无关性是锁无关算法的本质特征，即保证整个系统会在有限的步骤内完成对同一共享数据的某些操作。为证明算法的锁无关性可将算法看做状态转换系统，利用状态转换图表示算法的执行语句和中间状态。通过状态转换图的转换过程，分析在有限次状态转换中是否存在一个操作完成。通过对本算法进行建模和分析得到算法是锁无关的。

2.5 内存管理

数据域的内存管理有任务处理，结点占用内存必须有内存管理程序处理。在多任务环境和不使用锁的情况下，即使使用引用计数方法也无法确定一个结点能否安全释放。判定引用计数是否为 0 与释放结点之间引用计数不受保护，其他任务可增加引用计数，从而引发错误的释放。

原算法中只是假设具有完善的内存管理程序，而没有进行详细的描述。本文中实现了一种良好的内存管理程序，主要流程图如图 2。

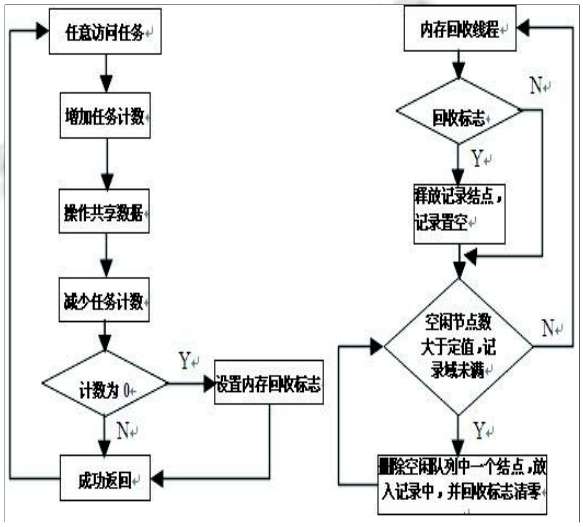

图 2　内存管理相关流程图

在每个操作过程中包括结点的申请和释放，与内存管理程序紧密相连。空闲结点队列保存所有空闲结点，也可通过锁无关数据结构实现。申请结点时，空闲队列不为空时返回一个空闲结点并减少空闲结点计数，否则直接向操作系统申请新结点。释放结点时，直接将空闲结点添加到空闲队列并增加空闲结点计数。

当访问双端队列的任务计数为 0 时，空闲队列中的结点不会被任何任务引用，可以安全释放。内存回收线程根据任务计数和空闲结点计数确定空闲结点释放的时机和数量。

3　性能分析与应用

3.1 性能分析

实验中采用的硬件测试平台为：CPU 为 Pentium (R) Dual-Core CPU E5300,2.6GHz；内存为 2G；操作系统为 Linux Ubuntu10.04。

实验中对改进型的锁无关双端队列和基于锁的双端队列进行性能对比测试，以相同情况下的执行速度快慢为优劣标准。测试过程中重要的两个方面为：1）

共享数据同步与任务访问共享数据的并行度有关，因此需在不同并行度的情况下进行测试；2）为有效测量执行时间并减少误差，应在对共享数据采用大量访问的情况下记录执行时间。

实验中对初始化为空的双端队列进行一系列的左入队、左出队、右入队和右出队操作[2]。设操作总数为 N，任务数为 P，则每个任务的操作次数为 N/P，每一种操作的数量相同为 N/4P。工作量一定的情况下，任务数量越大则并行度越高，执行时间越少算法越优异。

实验中取 N=1000000，P 分别取 1、2、4、8、16、32。改进型的锁无关双端队列和基于锁的双端队列的执行时间如图 3 所示。

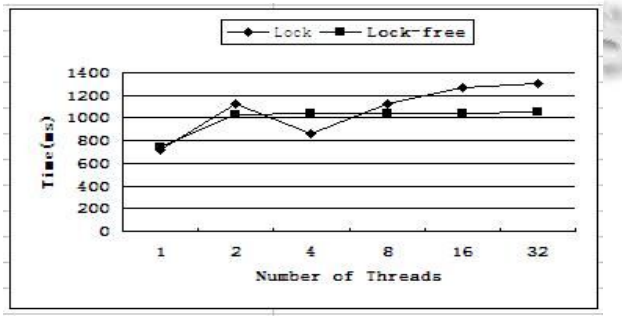

图 3 Lock、Lock-free 双端队列的时间开销

由图 3 可得，锁无关双端队列比基于锁的双端队列具有更高的执行速度。当任务数大于 4 时，前者比后者提高 8.5%~24.4%。随着并行度的增大，锁无关算法的执行速度越快，优势越明显。

锁无关算法对任务进行同步时不采用锁机制，避免了对共享数据访问的相互排斥、请求和保持等必要条件。锁无关算法中不存在死锁、优先级逆转和持有锁的任务崩溃导致其他任务永久阻塞而产生的低容错性等缺点发生的必要条件，从而得到避免。

综合上述，锁无关双端队列算法具有更高执行速度，避免了使用锁而引起的死锁、优先级逆转、低容错性等缺点，有利于提高了实时系统的性能和实时性。

3.2 应用实例

RTAI 是一款具有实时扩展的 Linux 内核，广泛应用于各种实时系统。实时 FIFO 队列是 RTAI 中重要的任务间通信工具。

为尽可能地保持接口的一致性，锁无关的实时 FIFO 队列采用实时 FIFO 队列相同的实现方式。锁无关实时 FIFO 队列包括创建、销毁、控制、读和写接口，以内核模块形式实现。锁无关实时 FIFO 队列在内核空间可直接被调用，在用户空间被看作普通的字符设备，因此其接口分别由内核调用函数和字符设备驱动函数实现。系统中共设置了 32 个锁无关实时 FIFO 队列，并相应生成 32 个字符设备文件。

锁无关实时 FIFO 队列中的共享数据由双端队列存储，其中采用右入队和左出队操作即可实现 FIFO 队列。通过对锁无关双端队列各种操作的封装实现锁无关实时 FIFO 队列的接口。

锁无关实时 FIFO 队列具有的诸多优点将有利于提高实时操作系统 RTAI 中任务间通信的性能和实时性，取得良好效果。

4 总结

本文提出了一种改进型的锁无关双端队列算法，对比并改进了原有算法的不足，通过实验验证了算法相比于基于锁的双端队列具有更高的执行速度，并避免死锁、优先级逆转、低容错性等缺点。最后将该算法应用到对 RTAI 下实时 FIFO 队列的改造中，有利于提高实时系统的整体性能。

参考文献

1 Michael MM. CAS-Based Lock-Free Algorithm for Shared Deques. The Ninth Euro-Par Conference on Parallel Processing, 2003. 651–660.

2 Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Proc. of the Annual ACM Symposium on Principles of Distributed Computing. 1996. 267–275.

3 Sundell KH. Efficient and Practical Non-Blocking Data Structures. Gothenburg, Sweden: Gothenburg University, 2004.

4 Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing, 2010,70(1):1–12.

5 Colvin R, Dongol B. A general technique for proving lock-freedom. Science of Computer Programming, 2009,74 (3):143–165.

6 Sundell KH. NOBLE Professional Edition Application Programmers Interface (API). 2008.5.

7 Hansson H. A network for real-time research and graduate education in Sweden. Uppsala, Sweden: Uppsala University, 2006:161–179.