并发数据结构及其在动态内存管理中的应用

重庆大学硕士学位论文

（学术学位）

学生姓名：刘　恒

导师姓名：杨小帆　教　授

专　　业：计算机软件与理论

学科门类：工　学

重庆大学计算机学院

二ＯＯ一三年五月

Concurrent Data Structures with Applications in Dynamic Memory Management

A Thesis Submitted to Chongqing University

in Partial Fulfillment of the Requirement for the

Master's Degree of Engineering

By

Heng Liu

Supervised by Prof. Xiaofan Yang

Specialty: Computer Software and Theory

College of Computer Science of Chongqing

University, Chongqing, China

May 2013

摘　　要

随着多核处理器越来越广泛的应用，软件如何改变传统的串行设计理念以适应并发的挑战正变得越来越紧迫。作为应用程序基础的数据结构首当其冲，如何在保证数据结构正确性的同时提供更高的并发度和扩展性成为研究人员关注的重点问题。在这种情况下，无锁数据结构应运而生。无锁数据结构摆脱了锁的限制，能够避免优先级反转问题并承受线程的随机故障退出，提供了更高的并发度和更好的扩展性。但是，无锁数据结构所要应对的是由多核处理器引入的更多的并发线程，以及随之而来的更大数量的动态内存消耗和对运行时动态内存分配器的更加频繁的访问，容易在高并发情况下形成瓶颈。

本文从负载均衡的角度考察无锁数据结构相关的动态内存管理问题，将其抽象成“逆向”负载均衡问题，基于负载均衡的思想提出了两种并发队列并将这两种队列用作线程本地缓冲区构建了两种供无锁数据结构使用的内存池，来解决无锁数据结构的动态内存管理问题，具体工作如下：

① 分析了并发数据结构领域的历史和现状，对该领域的各种常用技术和结构做了研究，指出了各自相对的优缺点和在多核时代面临的挑战。

② 对共享内存多核处理器环境下的并发内存管理做了总结，分析了该领域的几种通用技术，指出了研究人员对于无锁数据结构动态内存管理问题特殊性的忽略，提出了用内存池解决无锁数据结构的动态内存管理问题。

③ 介绍了负载均衡思想在计算机系统任务调度领域的应用，分析了任务窃取和任务分担这两种常用的调度策略，研究了几种任务窃取相关数据结构的原理、设计和实现，并用负载均衡的思想将无锁数据结构的动态内存管理问题抽象为“逆向”均衡问题，提出用负载均衡的方法解决该问题。

④ 提出了两种支持窃取和/或归还操作的并发队列，并在这两种队列的基础上分别构建了两种供无锁数据结构使用的内存池。这两种内存池可以在运行时通过在线程本地缓冲区之间执行窃取和/或归还操作以实现内存资源在各线程间的均匀分布，以改善无锁数据结构的性能。

⑤ 将本文提出的两种内存池和普通内存池分别应用于无锁数据结构，并对它们的性能作了详细的测量和对比，证明本文提出的两种内存池的各个指标在广泛的并发水平下均明显优于普通内存池，同时，还通过实验测量并证明了窃取和归还这两个均衡操作的有效性。

关键词：并发，窃取，资源均衡，内存池，无锁

ABSTRACT

With the far spreading utilization of modern multicore processors, software industry finds itself been posed with quickly growing pressure to transform its usual sequential methodology so as to meet with this new concurrency challenge and a new era of computing. In order to achieve better speed up, we need high performance concurrent data structures so that concurrent threads' execution time spent on interacting through shared objects (data structure) can be reduced. Unfortunately, concurrent data structures are difficult to design. There is a kind of tension between correctness and performance: the more one tries to improve performance, the more difficult it becomes to reason about the resulting algorithm as being correct. Also, it's hard to strike a balance between single performance under single thread and scalability with the growing number of threads enabled by more and more available processor cores.

One pressing issue that is borthering high performance concurrent data structues, especially lock-free ones, is dynamic memory management---the larger amount of contending threads enabled by the multicore processor induce a huge amount of rise in threads consumption of dynamic memory related to the shared lock-free data structures, and as a result, more frequent interaction with the runtime dynamic memory allocator, which will become a bottle neck under high level of contention.

This paper tries to look at lock-free data structures' dynamic memory management from a load balancing point of view and treats it as "backward" load balancing, so that we can use the technicals in the field of load balancing to solve lock-free data structures' dynamic memory management problem. Based on load balancing, we propose two concurrent queues and upon them, two memory pools that can distribute dynamic memory more equally between threads' local buffers.

Firstly, we analysed the history and current status of researches in the field of concurrent data structures, we investigated the common used technical and constructs in this field and pointed out their relative adavatages and shortenings to each other, and the challenge they are facing in the multicore age.

Secondly, we investigated dynamic memory management with shared memory multicore processors, studied typical concurrent dynamic memory allocators, pointed out that researchers ignored the special cases related to lock-free data structures'

dynamic memory management, and proposed memory pools as a supplement.

Thirdly, we looked into the problem of load balancing, particularly task scheduling in modern computing systems. We studied the two major technicals—work stealing and work dealing, and modeled lock-free data structures' dynamic memory management as "backward" load balancing.

Then two concurrent queues that supports steal and/or return operations were proposed along with two memory pools built upon them which can achieve more equally distribution of dynamic memory among threads' local buffer by performing steal and/or return operations during runtime, so that lock-free data structures' dynamic memory consumption can be reduced.

We also performed detailed experiments to investigate the performance of the two memory pools we proposed compared to regularly used conventional memory pools. Experiments show that our implementations out performed the conventional memory pools under a wide range of contention level, they achieve much less dynamic memory consumption and as a result, considerably less average operation execution time of the target lock-free data structures. Our experiments also showed that the two proposed balancing operations i.e. steal and return did help to achieve far better balancing results compared to the conventional memory pool.

Keywords: concurrent, steal, resource balancing, memory pool, lock free

目　　录

1 绪 论

在很多程序中，一个或多个竞争线程需要依赖同类资源的实例以保持进度，这些实例可以是待申请或释放的动态内存、需要响应的用户请求、待处理的订单以及待解码的网络数据包。绝大部分情况下，这些线程竞争访问的资源都是重要系统资源或事件且受到系统的严格控制[1, 2]，线程对这些资源的访问需要被细致地协调以防访问冲突或不当共享产生的的错误导致系统崩溃[1, 2]。线程所需资源的分布因程序而异，或位于各线程皆能访问的全局存储区，或位于只有拥有者线程才能访问的线程私有存储区（Thread Local Storage，TLS）。应用程序的总体性能是由该程序所产生的所有工作线程决定的，而线程的进度又在很大程度上取决于它们对其执行连续性至关重要的资源获取的顺利与否，线程对这些资源的访问通常体现为从存储区中获取一个待处理单元或者将数据处理的结果放置到存储区。

本文提出了两个可扩展并发队列，通过提供均衡操作平衡竞争线程对资源的占有和获取以在竞争线程之间协调资源分布。同时，基于这两个可扩展并发队列，本文提出了两个为无锁动态数据结构[1, 2]构造高效可扩展内存池的方法，应用本文的方法构建内存池，不仅可以有效降低目标无锁数据结构操作的动态内存消耗量，还可以进一步地降低目标无锁数据结构操作的平均执行时间。

1.1 概述

研究者们提出了各种不同的数据结构和/或算法来协调线程对共享资源的访问。Rudolph 等人[3]提出了一个简洁且可扩展的负载均衡方法用以解决多处理器系统中的处理器调度问题，该方法将任务分布到各个处理器的本地工作堆上，各个处理器以一个与其本地工作堆中任务数量成反比的概率执行均衡操作：①以一个一致的概率随机选取一个处理器并检查其工作堆中任务数量；②跟该处理器交换任务以使两个处理器工作堆中的任务数量相等。Blumofe 和 Leiserson [4]提出了一个随机任务窃取方法以在使用多处理器的完全受限任务[4]处理系统中解决处理器调度问题，他们证明了用该方法实现的任务窃取处理器调度器的时间界。Arora 等人[5]提出了第一个非阻塞[6,7]任务窃取算法并给出了实现，该算法为每个进程维持一个无锁本地队列并将到达的任务分布到各个进程的本地队列中，当一个进程的本地队列为空时它将利用无锁队列提供的窃取方法尝试从其它进程的本地队列中窃取一个任务。算法的精髓在于其非阻塞地从大小固定的队列中窃取单一元素的模式，在该模式下，进程通过仅从本地队列的一端获取任务而让窃取操作在队列的另一端进行，大大减少了开销高昂的原子（Atomic）[8, 9]操作的使用。Handlers 和

Shavit [10]为共享内存多处理器系统中的高效任务窃取提出了一个非阻塞任务窃取队列，该队列允许线程通过一个窃取操作无锁地窃取目标队列中一半的任务，有效促进了平衡条件的达成。Chase 和 Lev [11]提出了第一个适用于共享内存多核处理器的实用无锁任务窃取队列，他们使用循环队列解决了界线问题。

在动态内存管理领域，研究主要集中于：①设计和实现高效（并行）内存分配器以在运行时提供本地性以及随处理器可用核数增多的扩展性；②通过运行时和/或编译时分析、采样和统计为应用程序提供专用内存池以提高对特定尺寸的内存申请和/或释放操作的处理效率并减少程序的动态内存消耗。然而，研究者却鲜有涉及无锁数据结构的动态内存管理问题，特别是供动态无锁数据结构使用的高效可扩展内存池[12, 13]。随着共享内存多处理器系统越来越在市场上占据主流，无锁数据结构也越来越走向使用，对无锁数据结构的动态内存管理问题的研究也显得越来越重要和迫切。

1.2 研究内容及主要贡献

本文提出的第一个并发队列[12]与 Chase 和 Lev [11]为多线程间高效任务窃取设计的无锁循环任务窃取队列相似。任务窃取是基于以下情况的观察而提出的： 在任务动态到达的在线系统中，尽管各处理单元（线程/进程/处理器）在初始时拥有相同的负载，但在系统的长期运行过程中由于动态到达的任务自身对执行时间的要求、系统调度以及缺乏全局试图，各线程在负载量上会产生极大的差异，导致有的工作单元无负载处理而空闲与此同时其它处理单元却依然繁忙。任务窃取的核心是一个低开销且可扩展的任务窃取模式，在该模式下负载低的线程执行窃取操作以使各线程在负载量上尽量均衡：①选取目标处理单元，②获取目标处理单元的负载量，③当两个处理单元的负载量差值超过某个阈值时从目标单元窃负载。通过让负载较低的线程执行任务窃取，提交的任务将获得较低的响应时间，应用程序的总体性能也将得到提高。

基于前述资源窃取队列，本文构建了供动态无锁数据结构使用的资源窃取型内存池[12]，该内存池为每个共享目标数据结构的线程维持一个本地缓冲区，并用资源窃取队列实现线程本地缓冲区。由于应用目标是动态无锁数据结构，线程对它的操作涉及的内存分配和释放总是以“节点”为单位。资源窃取内存池事先给每个线程预分配一部分节点并存储在各线程的本地缓冲区中，在应用程序执行期间，用完本地缓冲区中节点的线程将尝试从其它竞争线程的本地缓冲区窃取一个节点而不是直接通过运行时内存分配器向系统申请，为公平性和平衡条件的考虑，线程对窃取目标的选择是随机的。通过提供窃取操作，资源窃取内存池能够达到以下两方面的效果：①线程对共享无锁数据结构操作涉及的内存申请被提交到多个

地点——本地缓冲区以及其它线程的本地缓冲区（执行窃取操作时），减少了线程对热点位置——运行时内存分配器的访问，减少了在该处的并发访问涉及的同步开销；②减少了无锁数据结构操作涉及的动态内存使用量及操作在动态内存管理上花费的时间。事实上，资源窃取内存池对无锁数据结构动态内存消耗量的减少的附带效应相当明显，使得无锁数据结构操作的平均执行时间显著下降——动态内存管理花费时间的减少超过了窃取的同步操作的时间开销。

本文将资源窃取内存池[12]应用于几种常用动态无锁数据结构，实验显示各个并发水平下资源窃取内存池在动态内存消耗（平均减少 50%）和目标无锁数据结构操作平均执行时间（平均降低 10%）这两个指标上都超越了传统内存池。然而，在达到一定的并发水平后，资源窃取内存池带来的性能增益便无法随并发水平的提高而继续扩展，究其原因，在于它仅仅基于窃取这一个操作来实现内存资源在各个线程本地缓冲区间的均衡分布，而这是远远不够的。更进一步地，资源窃取内存池忽略了目标数据结构操作所涉及的动态内存释放操作使其直接到达运行时内存分配器，于是，当并发水平升高时，这些被忽略的动态内存释放操作在运行时内存分配器处产生的同步操作开销就会逐渐升高直至抵消甚至超过窃取操作所消除的那部分同步开销，增加了目标数据结构操作的内存消耗和操作平均执行时间。

本文的下一个实现，也是本文的核心，是一个同时支持并发窃取和归还操作的队列——资源窃取和归还队列，以及基于该队列构建的资源均衡内存池，该内存池可以高效地将内存资源（节点）在共享目标无锁数据结构的线程的本地缓冲区之间均衡地分布以降低无锁数据结构相关的动态内存消耗和操作平均执行时间。

资源窃取和归还队列用以达到比仅支持窃取操作的资源窃取队列更好的平衡条件，以更均衡地将内存资源在线程间分布。窃取和归还操作的实质在于均衡操作的执行不应该仅仅是可用资源较少线程的特权，在资源拥有量上有盈余的线程也应该执行均衡操作，如此一来它们可以将富余的资源提供给资源拥有量较少的线程，从而资源在线程间的分布更为均衡。归还操作可以按如下方式执行：①在资源拥有量上有盈余的线程选定一个目标线程，②若其资源量较少则向其转移一部分资源。通过让资源较少的线程执行窃取操作同时让资源较多的线程执行归还操作，应用程序可以期望更少的资源消耗和更高性能。

与资源窃取内存池相似，支持资源窃取和归还的资源均衡内存池也为每个共享目标无锁数据结构的线程维持一个本地缓冲区，但该缓冲区用资源窃取和归还队列实现。此外，资源均衡内存池使得拥有额外内存（节点）的线程尝试将待释放的节点放置到其它线程的本地缓冲区而不是直接通过内存分配器释放给系统。

被选择接收待释放节点的线程也是随机选取得到的，尝试执行归还操作的线程会遇到目标线程本地缓冲区已满的情况，因而归还操作会被执行多次直到归还成功或达到最大尝试次数。由于将动态内存通过运行时内存分配器释放给操作系统同样涉及堆的同步，归还操作减少了内存释放操作的数量及其涉及的堆同步开销，减轻了高并发情况下在运行时内存分配器上的瓶颈效应，更进一步地减少了目标无锁数据结构操作相关的动态内存消耗量和操作平均执行时间。

实验显示，基于资源窃取和归还队列的资源均衡内存池在各种并发水平下的性能都超过了基于资源窃取队列的资源窃取内存池：①减少目标无锁数据结构动态内存消耗量（平均减少 68%），②减少操作操作平均执行时间（平均减少 15%）。此外，由于增加的归还操作和窃取操作共同实现了更好的平衡条件，资源均衡内存池的扩展性也大大增强，并能随并发水平的提高持续地提升性能。

1.3 论文组织结构

论文共分 6 章，组织结构如下：

第一章　首先对多线程环境下动态内存管理的重要性及负载均衡作了简要的介绍；其次介绍了本文研究的内容，对本文设计和实现的两种内存池的性能和优缺点做了阐述和比较；最后归纳了各章节的内容安排。

第二章　在对并发数据结构及多核时代、共享内存多核处理器环境下的动态内存管理、负载均衡及资源均衡的动机这三个方面进行研究和总结的基础上，提出用内存池解决无锁数据结构的动态内存管理问题，并用资源均衡技术构建内存池。

第三章　提出并实现了无锁资源窃取队列，在其基础上构建了无锁资源窃取内存池，并给出了队列和内存池操作的正确性和无锁性证明。

第四章　提出并实现了资源窃取和归还队列，在其基础上构建了资源均衡内存池，并给出了队列和内存池的正确性证明。

第五章　将本文提出的两种内存池分别应用于无锁队列和无锁栈这两种常用的无锁数据结构，实验表明本文提出的两种内存池在内存消耗量、操作平均执行时间、线程私有缓冲区中节点数方差这三个指标上均显著优于普通内存池。

第六章　对本文的研究工作进行了总结，并在其基础上给出了未来工作的展望。

2 研究背景

随着多核处理器的应用越来越广泛并逐渐成为微处理器市场的统治性力量，桌面处理器，移动处理器，嵌入式处理器，甚至连主流手机处理器大多拥有至少两个核心。这个快速扩散而又不可抗拒的趋势正给软件施加越来越大的压力：如何更好地利用伴随快速增长的处理器核心而来的几近无穷的计算能力。对于许多运行在拥有多核处理器系统之上的软件而言，主要的挑战在于更多的处理器核心带来了更多的可用线程，而更多的可用线程就意味着在确保多线程操作正确性和可扩展性的前提下实现性能提升所带来的额外同步开销。此外，多线程程序在设计、实现和调试方面额外的复杂性也限制了程序的扩展性和正确性证明。受此影响最大的便是那些应用程序广泛使用的基础数据结构：队列[13]，栈[14]，链表[15]，二叉树[16]以及散列表[17]，多线程访问对这些数据结构的并发度、效率和扩展性提出了更加严格的要求，产生了对这些数据结构相应无锁[6,7]实现的迫切需要。

对动态（无锁）数据结构而言，研究者对其性能的关注主要集中在四个方面：

① 多线程对数据结构并发访问产生的所谓“热点(Hot Spot)”区域，在这些“热点”区域会出现并发操作的串行执行现象：数据结构中的某些区域要求排他性访问，竞争线程对数据结构方法的并发调用在这些区域形成一个等待队列并逐次通过，这在相当程度上降低了对多核处理器核心的利用率，大大浪费了系统的计算资源并给系统带来了性能惩罚。

② 开销高昂的同步原语：线程同步所依赖的基本手段——锁，互斥体，信号量，管程以及最近得到广泛使用的“比较并交换”（Compare And Swap，CAS）[19, 20, 21]操作。这些同步手段或由操作系统提供——在运行期引起程序在核心态和用户态之间切换因而代价高昂，或是由处理器直接支持的硬件指令——需要锁住系统总线以确保排他性因而限制了其他程序的线程对无关内存位置的访问。

③ 对动态内存管理的压力：更多可用处理器核心意味着更多的可用线程，带来对共享数据结构更多、更频繁的访问，导致更多的动态内存消耗和对运行时内存分配器的更频繁的访问。由于内存是关键系统资源，向系统申请内存或者将内存释放以归还给系统都会导致应用程序堆被锁定以确保多线程访问下的安全性，多个线程并发地尝试锁定堆带来大量同步操作会大大降低系统性能并引起诸多问题——伪共享，数据本地性，缓存“乒乓”。

④ 对负载均衡的压力：应用程序的总体进度取决于它产生的每个进程的进度，而这又在很大程度上取决于各个线程的负载量。多核处理器核心数的稳步提升带来更多的可用线程，从而引入了更多的动态性，使得负载均衡更加难以实

现，同时，线程数量越多各个线程之间负载量的差异给程序性能造成的拖累就越大。

诸如锁，互斥体，信号量，管程等均为操作系统为确保多进程/线程交互安全性而提供的高开销原语，且在使用过程中均会调用所谓的“系统调用（System Call[22,23]”，这会引起程序在核心态和用户态之间的来回切换：导致高达数万处理器周期[24]的额外开销。此外，上述同步机制的提出时间相对久远（早在 20 世纪 70 年代），当时对于同步机制的要求主要集中在多进程交互的安全性而非效率。即使是晚些时候伴随多核处理器而出现的 CAS 操作，尽管其由处理器硬件指令直接实现，仍然会消耗相当多的处理器周期，况且该操作本身就相对受限：现有多核处理器仅支持同时对一个内存字的 CAS 操作且互斥性的实现需要对总线加锁，导致其它线程对无关内存位置的访问受阻，降低了系统的性能。

对于多线程环境下的动态内存管理而言，线程共享一个由所属进程拥有的公共堆。在一个进程上下文中运行的所有并发线程的动态内存管理操作都会操作共享堆，在堆上产生巨大的同步开销，而随着处理器核数的不断增长带来的越来越多的可用线程，这种同步开销只会越来越大，越来越难以接受。同时，动态内存管理本身就在许多应用程序的总运行时间中占据相当的比例[25, 26, 27]，多核处理器的广泛应用便使得动态内存管理的重要性凸显。

目前，负载均衡领域的研究主要集中在分布式系统中的进程/处理器的高效调度，以及互联网上的流量分担，忽略了对较低层次的动态系统中负载均衡的研究。事实上，通过将抽象层次提高，包括多线程环境下动态内存管理等诸多问题都可以看作负载均衡问题，并用负载均衡的思想来解决。各个线程对共享数据结构的访问表现为向数据结构插入或者从数据结构中删除节点，也意味着通过运行时内存分配器向系统申请或释放内存。尽管线程对共享数据结构的访问调用的方法类似，但其访问模式各异（由线程本身的动态性以及操作系统的处理器调度算法导致），因而对动态内存的消耗模式就更是不同，就在各个线程对动态内存的消耗上产生了相当程度的不均衡。

本文提出的资源窃取队列、资源窃取和归还队列以及以它们为基础构建的内存池正可以解决上述问题：

① 降低热点位置的同步开销：队列有两个访问点——头部和尾部，不同类型的方法调用可以经由这两端执行，减少了访问冲突和同步开销，以及由此产生的缓存失效带来的总线内存传输，

② 最小化同步原语相关开销：线程本地缓冲区的引入减少了同步原语的使用，同时用开销相对较低的 CAS 操作代替了传统的涉及代价高昂的系统调用的同步原语，

③ 更少的动态内存管理相关同步开销：线程的内存需求尽可能地经由内存池通过线程本地缓冲区或其他线程本地缓冲区解决，对堆的访问仅在必要时执行——带来更少的堆访问级相关同步开销，

④ 各线程对动态内存更均衡的消耗及由此带来的更少的总体动态内存消耗量：内存池通过平衡操作在各线程的私有缓冲区之间腾挪以实现内存资源在各个线程间的均衡分布，减少了单个线程因为私有缓冲区为空而通过运行时内存分配器向系统申请内存以及私有缓冲区满时通过内存分配器向系统释放内存的几率，带来更少的动态内存消耗，减少的动态内存分配/释放操作减少了数据结构操作的平均执行时间：应用程序在动态内存管理上花费相当比例的时间，而更少的动态内存管理操作便意味着更少的（同步）开销，带来数据结构操作平均执行时间的减少。

2.1 并发数据结构及多核时代

多核处理器正在革命性地改变设计和使用数据结构的方式，因而那些能够提供高效多线程访问并且能在高负载下持续提供扩展能力的数据结构就变得越来越重要。根据阿姆达定律[28]（见图 2.1），加速比取决于程序中必须被串行执行的部分所占的比例。在大部分程序中，这些必须被串行执行的部分涉及线程间通信和协调以及动态内存使用（申请和释放），在多处理器/多核机器上，这些串行部分通常表现为对共享数据结构的并发访问。阿姆达定律表明，尽可能地将程序中的串行部分并行化是非常值得而且十分必要的，其关键就在于使用高并行度的并发数据结构。

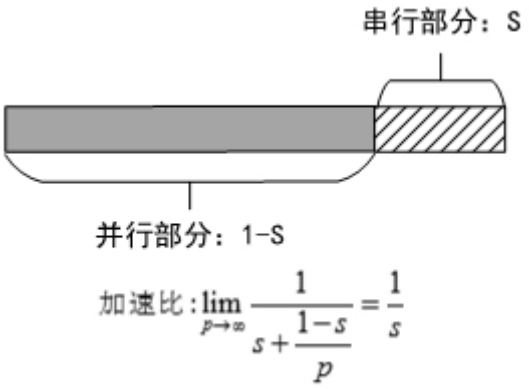

图 2.1 阿姆达定律

Fig.2.1 Amdahl’s Law

早期的并发数据结构主要关注多线程访问的安全性[29, 30, 31]，而非多线程访问下的性能，使用诸如信号量[32, 33, 34, 40]、互斥体[33, 34, 35]、管程[33, 34, 36]、自旋锁[33, 34, 37, 38, 39]来对多线程的并发访问进行同步。这一阶段的并发数据结构实质上是非并发数据结构与同步结构的简单结合，将数据结构的修改操作抽象成临界区并用同步结构加以保护（见图 2.2）。

图 2.2 临界区模型的抽象结构

Fig.2.2 Abstraction of Critical Section

信号量最早由 Dijkstra[41]提出，它本质上是一个计数器和一个等待队列（见图 2.3）。线程/进程通过调用信号量的 P 和 V 两个方法来实现同步：P 方法用于减少信号量的计数值，将计数器减一，若计数器值为正则进入临界区，计数器值为负则进入等待队列等待唤醒；V 方法用于增加信号量的值，若计数器为负则将等待队列中的一个线程/进程释放。应用程序通过使用不同个数的信号量，并给信号量的计数器设定不同的初值来实现不同类型的同步，应对不同的同步问题模型（生产者/消费者问题，睡眠理发师问题，哲学家问题，荷兰银行家问题）[33, 34, 42, 43, 44, 45, 46]。

```
struct semaphore {
        int counter;
        list wait_list;
}
P (semaphore* sem)
{
        sem->counter--;
        if (sem->counter<0)
                block and wait on sem->wait_list
}

V (semaphore* sem)
{
        sem->counter++;
        if (sem->counter<=0)
                unblock one thread from sem->wait_list
}
```

图 2.3 信号量的一般形式

Fig. 2.3 General Form of Semaphore

互斥体可以看作是二值信号量，由一个 0/1 计数器（True/False 显示器）和一个等待队列（见图 2.4）。线程/进程通过调用互斥体的 Acquire 和 Release 方法来实现同步：线程/进程调用 Acquire 方法尝试将计数器从 1 变为 0，若成功则进入临界区执行操作，否则进入等待队列等待唤醒；Release 方法用于释放在互斥体上等待的线程/进程，若等待队列为空，则将计数器从 0 变为 1。

```
struct mutex {
        bool occupied;
        list wait_list;
}
Accquire (mutex* mut)
{
        if (mut->occupied)
                block and wait on mut->wait_list;
        else
                mut->occupied=true;
}

Release (mutex* mutm)
{
        mut->occupied=false;
        if (mut->wait_list is not empty)
                unblock one thread from mut->wait_list
}
```

图 2.4 互斥体的伪代码形式

Fig. 2.4 Pseudo Code for Mutex

管程由 Brinch-Hansen 和 Hoare 提出并发展，它由信号量实现的条件变量和定义在管程内部的必须互斥执行的函数构成（见图 2.5）。管程中定义的函数必须通过管程实例来访问，且必须互斥执行：①对于一个管程函数，任何时刻最多只能有一个线程/进程在执行；②若一个线程/进程在某个管程函数上执行过程中阻塞挂起，管程必须调度其它在该函数上等待的线程/进程进入执行。

```
Monitor stack
{
        Elem *buf;
        int top, capacity;
        Condition nonempty, nonfull;
        public void push(Elem x){
                if(top==capacity-1)
                        nonfull.wait;
                buf[top++]=x;
                nonempty.signal
        }
        Public Elem pop(){
                if(top<0)
                        nonempty.wait;
                Elem result=buf[top--];
                nonfull.signal;
                return result;
        }
}
```

图 2.5 管程

Fig. 2.5 Abstraction of Monitor

相对于阻塞锁（互斥量），使用自旋锁的线程/进程不会在获取锁失败后进入等待队列休眠等待唤醒，而是不停地测试锁的状态，第一个观察到锁的状态发生改变并成功锁定的线程/进程可以进入临界区。自旋锁实质上是一个提供互斥操作的 0/1 变量（见图 2.6）：线程/进程调用 Lock 方法以试图进入临界区，若失败则继续测试直到成功为止或者超时退出；Unlock 方法用于重置锁的状态。自旋锁的实现机制需要操作系统对软中断的支持和 CPU 时钟中断的配合，适用于使用时间片轮转调度算法[47, 48]并注重用户交互的系统[49, 50,]和实时系统[51, 52]。

```
struct spin_lock {
        bool locked;
}
Lock (spin_lock* lock)
{
        while(lock->locked)
                spin;
        lock->locked=true;
}

Unlock (spin_lock* lock)
{
        lock->locked=false
}
```

图 2.6 自旋锁示意图

Fig. 2.6 Sementic Demenstration of Spin Lock

上述同步结构及相关操作可以确保多线程环境下并发数据结构的安全性，但是它们都需要操作系统的干预，在执行同步操作时大多会导致程序在用户态和内核态[53, 54]之间的切换（见图 2.7），带来大量系统开销——这对于简单的并发数据结构如计数器而言就显得尤为昂贵。而且，这种数据结构+同步结构=并发数据结构的简单组合方式常常导致较高的并发粒度[55, 56, 57]，使系统中一部分处理器不必要地闲置，浪费了多处理器计算资源。

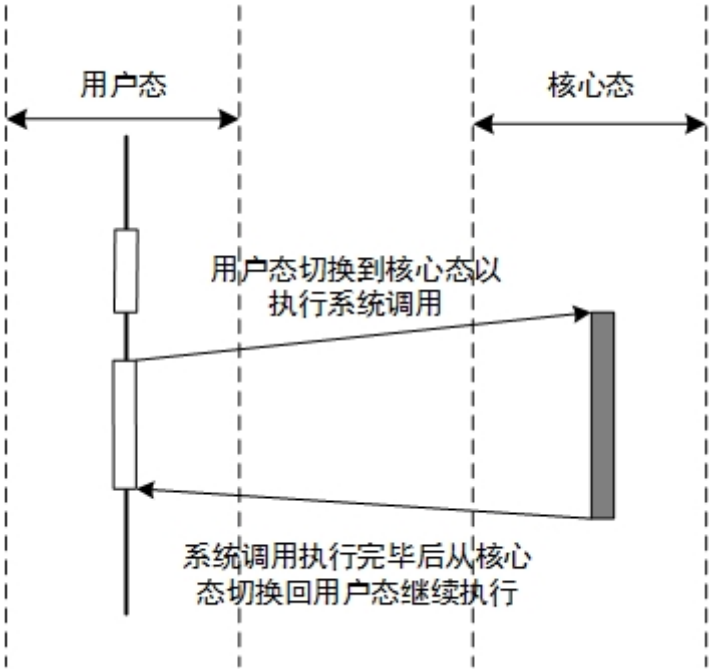

图 2.7 状态切换示意图

Fig. 2.7 Sementic Demenstration of Context Switching

此外，就性能而言，共享内存多核处理器及其快速发展所带来的可用处理器核数的增多，及随之而来的可用线程/进程数的增多，给并发数据结构带来了许多额外的挑战，这表现为：

① （线性）瓶颈对数据结构可扩展性的制约：数据结构本身的正确性定义使得在多线程操作下对临界区访问必须互斥，这导致在任意时刻最多只能有一个线程访问共享数据结构。瓶颈会严重制约数据结构的扩展性，根据阿姆达定律，若程序的 10%涉及（线性）瓶颈，则该程序在一颗 10 核处理器上能实现的最大加速比仅为 5.3，该处理器一半的核都处于空闲状态。因而，减少程序中线性部分的数量和长度对于性能而言就显得至关重要。

② 访存冲突：多个线程并发地试图访问共享内存区，从而在总线上造成额外的总线数据传输，导致系统性能的下降。在实现缓存一致性的多核机器上，包含锁的缓存行会被一而再地在处理器之间传输，这导致线程为访问共享区而付出更长的等待时间。而且，这还会由于多线程获取锁的失败带来的额外内存传输而加重。

③ 阻塞：当一个拥有锁的线程产生延迟，其它所有试图获取该锁的线程都将被迫停下来等待。这在当今的共享内存多核处理器上表现的更为明显，在绝大部分时间里，每个核上都有超过一个线程在运行，操作系统可能在任何时刻抢占一个正持有锁的线程。

Wing 与 Herlihy [58]在顺序一致性（sequential consistency）[59]及可串行化性（serializability）[60]概念之上发展出可线性化性（linearizability [58, 61]）来抽象和论证并发数据结构的正确性和进度条件（progress）。可线性化性要求两点：1）数据结构满足顺序一致性；2）令数据结构达成顺序一致性的操作排序不破坏操作之间的实时顺序。这使得可以在操作的执行期间找到一个线性化点（linearization point）[58, 61]，将操作按照各自的线性化点排序得到的结果就满足相应的顺序语义。

目前，并发数据结构的研究多集中无阻塞(non-blocking）[61, 62]数据结构（无等待（wait-free）[7, 61, 62]和无锁（lock-free）[61, 62, 63]）、高性能锁和(软件)事务式内存[64, 65, 66, 67, 68]。这些新型并发数据结构、并发同步结构广泛使用现代处理器提供的更为高效、轻量且工作在用户态的 Compare-and-Swap（CAS）指令实现同步结构和操作（见图 2.8），而不是相对昂贵且工作在核心态的操作系统同步原语。

```
bool CAS(destination, expected, new)
{
        bool result=false;
        atomically{
                if(*destination==expected)
                {
                        *destination=new;
                        result=true;
                }
        }
        return result;
}
```

图 2.8 CAS 操作的语义

Fig. 2.8 Sementic Demenstration of the CAS Operation

简单自旋锁的问题在于多核处理器带来更多的线程产生对锁的频繁、过度竞争导致的性能损失。指数退避（exponential back-off）[69]技术让获取锁失败的线程随机退避之后再重新尝试以减少锁的竞争，及失败的加锁尝试带来的内存传输，但存在所有线程退让时间太长以致锁闲置的问题。 Anderson [70]、Graunke 和 Thakkar [71]提出的基于数组的队列锁(queue locks)解决了指数退避技术导致的锁闲置问题，该方案使等待中的线程形成一个队列，并让每个释放锁的线程将锁的所有权传递给排在其后的线程（见图 2.9），既解决了锁闲置问题又极大消解了多线程对锁的竞争的激烈程度。在此基础上，Crummy 和 Scott [72]提出的基于链表的 MCS 锁、Craig [73]、Magnussen 和 Landin 以及 Hagersten [74]提出的基于链表的 CLH 锁又对队列锁的性能作了进一步的改进。通过 CLH 锁进行同步的每个线程拥有一个节点，每个节点包含一个标记，这些节点构成一个虚链表。当一个线程的前驱节点中的标记为 true 时，该线程获得锁。锁对象本身保留一个 tail 域，为获取锁，线程需要创建一个新节点，置其标记为 false，并用 CAS 操作将新节点放在链表尾部，然后在前驱节点的标记上旋转。在使用缓存的系统中，由于各个线程在内存的不同位置处旋转，拥有锁的线程释放锁时只影响在其节点的标记上旋转的等待线程，其它线程的旋转标记并未受影响，因而这些线程继续在其本地缓存行上旋转不产生额外的内存传送——极大减少了竞争并提升了系统的扩展性。然而，若用于非一致的 NUMA 机器上，线程将会在远程内存上旋转，再次造成额外的内存传输。MCS 锁通过让线程在自己的节点而非前驱的节点的标记上旋转，如此一来节点可以被分配在本地内存中，从而解决了 CLH 锁的问题。此外，研究者们还提出了不同类型的可退出锁（abortable locks）[75, 76], 组合锁[77]和读写锁（readers/writers locks）[78, 79]。

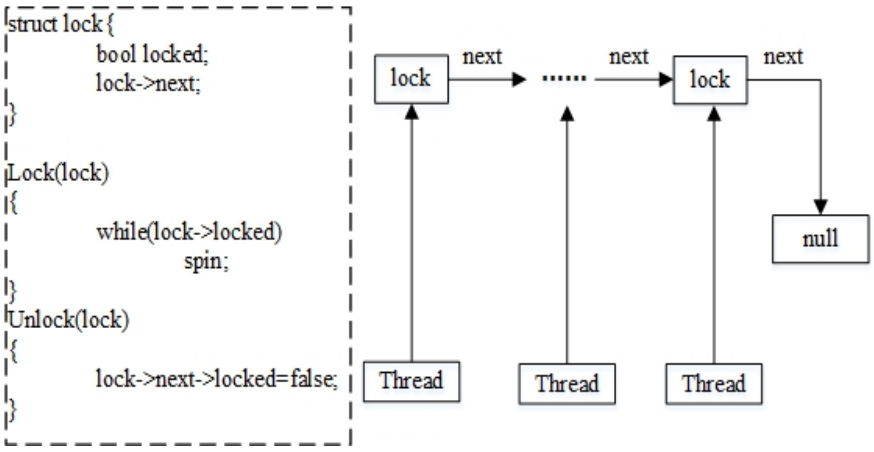

图 2.9 队列锁示意图

Fig. 2.9 Sementic Demenstration of Queue Lock

作为并发数据结构操作的进度条件，无锁（lock-free）操作可以确保在自身的有限步骤之内，有其它操作执行完毕，无等待（wait-free）是比无锁更强的进度条件，无等待操作可以确保在自身的有限步骤之内执行完毕，无需考虑其它操作的表现。无等待条件相比无锁虽然更强，但它更难实现，所以无锁数据结构是当前研究的主要关注点。

通过给每个数据节点附加一个冗余节点，Valois [80]实现了第一个无锁链表。Harris [81]实现了一个需配合垃圾收集器使用的无锁链表, 该链表给每个节点附加一个可以通过节点指针原子地访问的“deleted”域来标记节点是否已被（逻辑）删除。Michael [82]改进了 Harris 的实现，使其可以和内存回收方法[83, 84]配合使用，以摆脱对垃圾回收机制的依赖。

Treiber [85]提出并实现了第一个无锁栈，该实现使用有头指针的单链表来表示栈，并用 CAS 操作来实现对头指针的原子修改。后续研究[86]表明，Treiber 的无锁栈实现拥有比其它实现更好的性能，但头指针在竞争线程数增长后成为线性瓶颈，因而扩展性欠佳。通过使用使用消解（elimination）技术[87]，Handler 等人[88]提出了可扩展可线性化的无锁栈，解决了 Treiber 的问题。

Herlihy 和 Wing [58]提出了一个基于数组的无锁队列，但它必须要求数组无界。Michael 和 Scott [89, 90]提出了第一个可扩展可线性化无锁队列，该队列允许同时在队列两端的并行/并发操作。该实现维持队头和队尾指针，并使用帮助技术（helping technique）[61]来确保对队头/队尾施行的 CAS 操作的一致性。

基于递归分裂排序技术（recursive split ordering），Shalev 和 Shavit [91]提出了第一个无锁可扩展散列表（hash table）。与传统基于锁的散列表不同，该实现将元素放在单一无锁链表中而不是多个分离的桶里，并使用可扩展的提示数组来确保对元素的常数访问时间，操作通过提示在链表中找到靠近目标位置的点，然后跟随

链表指针到达目标位置。递归分裂排序技术则确保提示被简单、高效、递增地随着元素的插入而添加。

手工打造的（hand-crafted）无锁数据结构具有较高的吞吐量、并发度和扩展性，但也存在一些难以忽略的弱点：

① 复杂度较高，难以设计和实现：无锁性的要求与并发数据结构语义的严格程度是相互冲突的，设计需要融合这两个矛盾而又必须同时兼顾的方面。

② 难以进行正确性证明：可线性化性急无锁要求在每个操作中识别出可线性化点，而某些复杂操作则可能拥有多个可线性化点。此外，并发操作的正确性证明和无锁性证明复合后也大大增加了证明的难度。

③ 缓存、伪共享问题以及 ABA 问题：在共享内存多核体系结构下，程序的性能对缓存有较大的依赖性，而多线程共享无锁数据结构容易产生由伪共享导致的不必要的缓存失效和随之而来的总线存储传输。而且，无锁数据结构广泛使用的 CAS 本身一次只能比较和修改一个内存字的限制又容易产生 ABA 问题。

④ 耦合度大，难以复合[92]：无锁数据结构的设计复杂度使得操作内部耦合度大增，尽管无锁性可以叠加，但将多个无锁数据结构之间的操作又必须另加额外的并发控制机制（见图 2.10）。

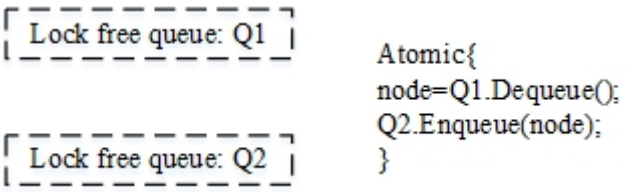

图 2.10 两个无锁队列互操作，需要额外的并发控制机制
以确保两个无锁队列间出队和入队操作作为一个整体的原子性

Fig. 2.10 Using Extra Concurrency Control to Ensure Atomicity
When Two Lock Free Queues Are Used

为此，研究者们提出了软件事务式内存来解决这些问题。 软件事务式内存借鉴数据库管理系统的事务处理，将并发线程对数据结构的访问抽象成内存访问并用事务的方式组织，将应用于数据结构操作涉及到的一系列内存访问组织成一个或多个事务，事务内的所有内存访问必须同时生效或丢弃，用冲突检测和仲裁机制来解决多线程访问的冲突问题（见图 2.11）。尽管软件事务式内存使用阻塞算法，受制于操作系统的调度算法，但通过使用高效的加锁/解锁模式、新型的轻量级自旋锁、可适配的冲突仲裁机制，软件事务式内存在提高了吞吐量和并发度的（低于手工打造的无锁数据结构）同时，极大减轻了应用层次的复杂度和负担。

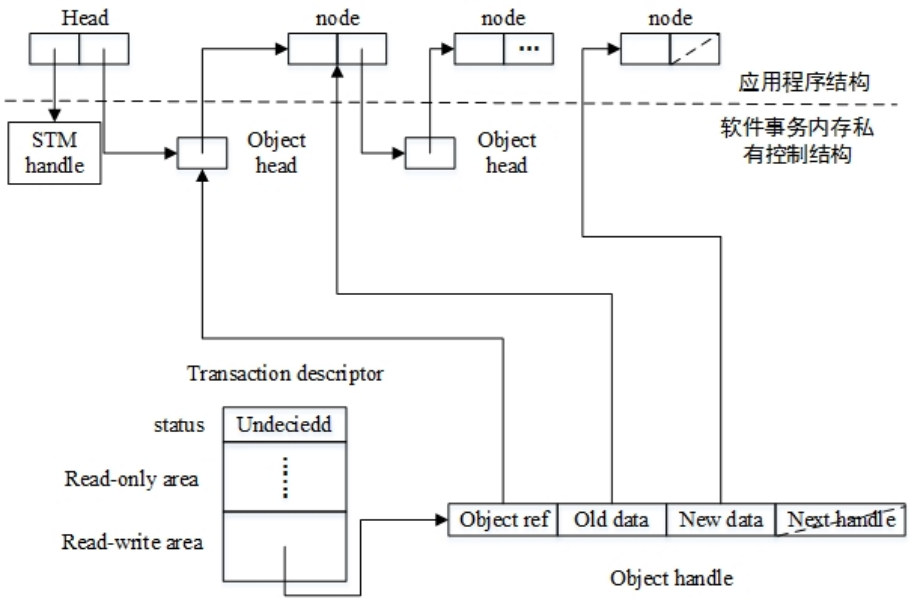

图 2.11 软件事务式内存

Fig. 2.11 Software Transactional Memory

2.2 共享内存多核处理器环境下的动态内存管理

就大多数 C/C++程序而言，动态内存管理可以算得上是最为普遍且代价高昂的操作之一了。作为关键性系统资源，内存由操作系统分配给程序运行使用，程序运行结束后由系统收回。由于现代操作系统均为多道系统，系统中同时驻留数十个甚至数百个程序运行，操作系统使用基于分页和/或分段的虚拟内存和请求调页技术来满足多个程序对内存的需求，这使得内存先得尤为珍贵。同时，为保证进程独立性和安全性，程序必须运行在独立地址空间中，给内存管理带来了额外的复杂性（见图 2.12）。研究显示，许多程序的动态内存管理会花费超过三分之一的运行时间[93, 94, 95]。

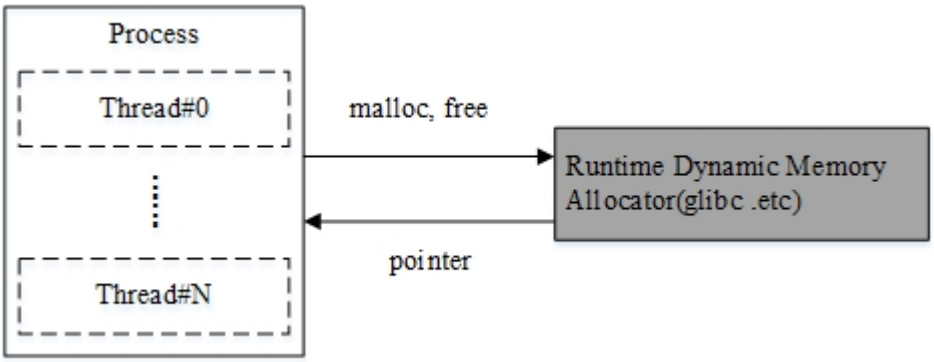

图 2.12 动态内存管理典型示例

Fig. 2.12 Typical Dynamic Memory Managment

动态内存管理的开销对于运行在多线程环境下的程序而言就更为突出：从应用程序堆中分配内存和将使用完毕的内存归还到堆上时都需要对多线程的操作进行同步，以确保正确性——内存错误极易导致程序出错导致用户关键数据或计算丢失，更严重的情况下会导致系统崩溃，多核处理器使得应用程序能够也倾向于使用更多的线程，带来更大的内存需求和更频繁的动态内存管理操作，堆的多线程访问涉及的同步开销也更加昂贵，使得内存分配器成为多核处理器系统的瓶颈。此外，多核多线程还引入了伪共享问题[96, 97, 98]：当多个处理器拥有位于同一缓存行中的不同数据却并非实际共享它们时，就产生伪共享问题（见图 2.13），当一个处理器更新本地数据时，其他处理器位于同一缓存行中得数据都将失效，在并行程序中，伪共享会导致程序性能低下。当内存分配器使堆上的数个由不同处理器写的对象处于同一个缓存行中或应用程序将数个不同对象传递给不同线程时都会产生伪共享问题。

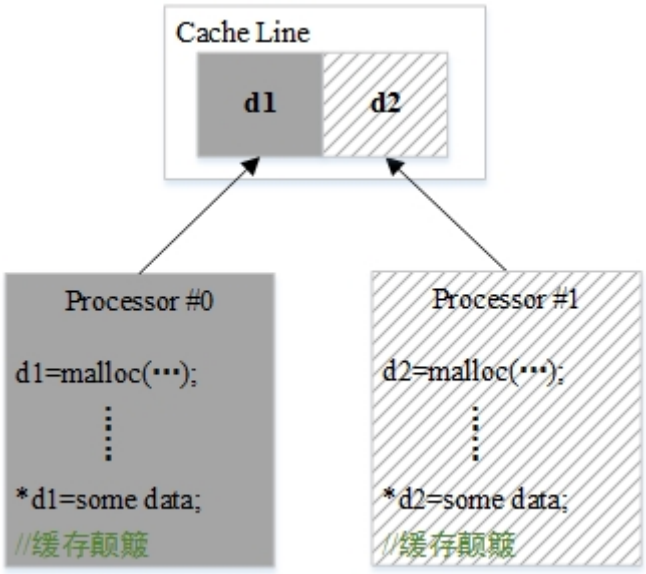

图 2.13 伪共享问题示例图

Fig. 2.13 False Sharing

动态内存管理领域的研究主要集中于设计和实现能够降低动态内存管理开销、增加应用程序堆上对象的引用本地性以改进内存管理操作执行时间、避免伪共享问题的并行内存分配器，以及通过编译时或运行时行为分析给程序对特殊尺寸的内存需求提供自定义的动态内存管理策略。

基于 Doug Lea 的非并行内存分配器[99]，Gloger[100]提出了一个用于 linux 系统的并行内存分配器——Ptmalloc。该分配器使用多个内存场地（arena）以减少多线程条件下的竞争， 每个内存场地由一个锁保护。若执行 malloc 的线程发现目标内

存场地已经被锁定，它会尝试锁定下一个，若所有内存场地均已被锁定，则向系统申请一个新的内存场地，从该处获取内存并将该内存场地加入到内存场地列表中。为增强本地性及减少伪共享，每个线程需要在其线程私有存储区中记载最近一次 malloc 函数调用使用的内存场地，在下一次调用 malloc 函数时将首先尝试锁定该内存场地并从中分配内存。释放内存时，线程会把内存归还给该内存所属的场地。

Berger[101, 102]实现了第一个适用于共享内存多核处理器系统的实用并行内存分配器——Hoard。对位于同一地址空间中的线程，Hoard 维持每个线程一个私有的仅所有者线程可以访问的本地堆和一个线程共享的全局堆。Hoard 以块为单位向系统申请内存分配到各个堆上，并记录每个堆的全部内存大小和已分配给线程的内存数量。每个块包括若干同样大小的区以减少内部碎片，块内还包含标记块内自由区的自由链表，自由链表按后进先出（LIFO）方式访问以增强本地性。当一个线程私有堆中的空闲块数量超过阈值时，Hoard 会从该堆中取走一部分块放入全局堆中，若私有堆中的空闲块数不能满足需求，Hoard 将从全局堆中分配一定数量的块给有需求的私有堆（见图 2.14）。由于基于锁，Hoard 不具有异步信号安全（async-signal-safe）性，无法被信号处理器等程序使用：若持有内存分配器锁的线程接收到信号，而信号处理器又需要申请内存，此时就会出现循环等待，产生死锁。

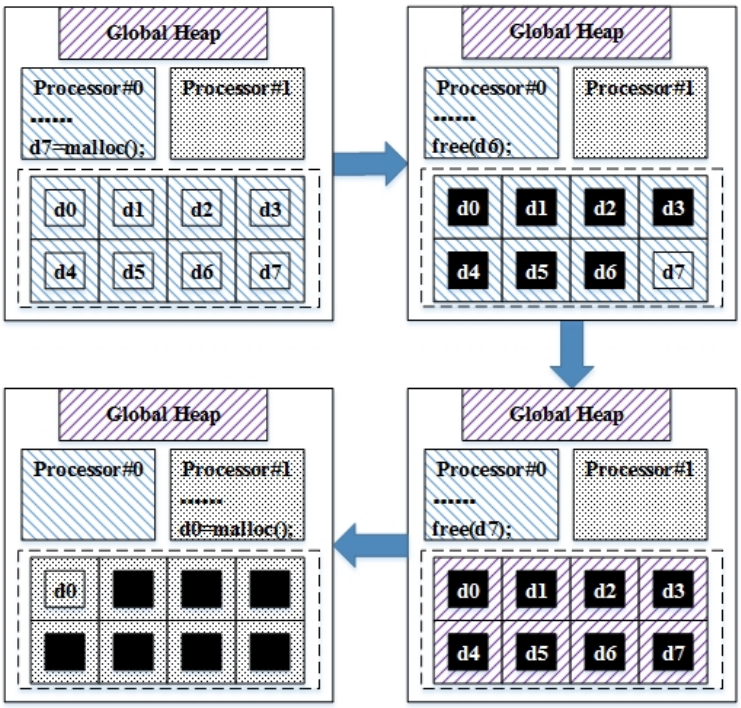

图 2.14 Hoard 内存分配/释放示意图

Fig. 2.14 Dynamic Memory Allocation/De-allocation with Hoard

Michael[103]改进了 Berger 的工作, 并实现了第一个适合共享内存多核处理器系统使用的无锁并行内存分配器。与 Hoard 不同，该分配器使用处理器堆而非线程堆。处理器堆中的内存被组织成若干具有描述符的超级块，每个超级块由若干个大小相同的单位块组成，根据内部单位块的大小，超级块被归为不同的类别。每个处理器堆最多拥有一个活动超级块，线程可以通过处理器堆的头部找到改处理器堆的活动超级块（如果有的话）。malloc 函数根据调用线程要求的内存大小和线程标识找到目标处理器堆，读取该处理器堆的活动超级块的描述符，从活动超级块预约一个单位块，原子地从该超级块中弹出一个单位块并更新描述符。free 函数原子地将释放的内存块压入它所属的超级块并更新其描述符（见图 2.15）。Michael 的无锁内存分配器是异步信号安全的，且可以完全避免锁引发的优先级反转[33, 34]问题，同时，由于内存分配/释放操作的无锁性，它可以承受任意线程内存分配/释放操作的失败以及操作系统调度算法的抢占。

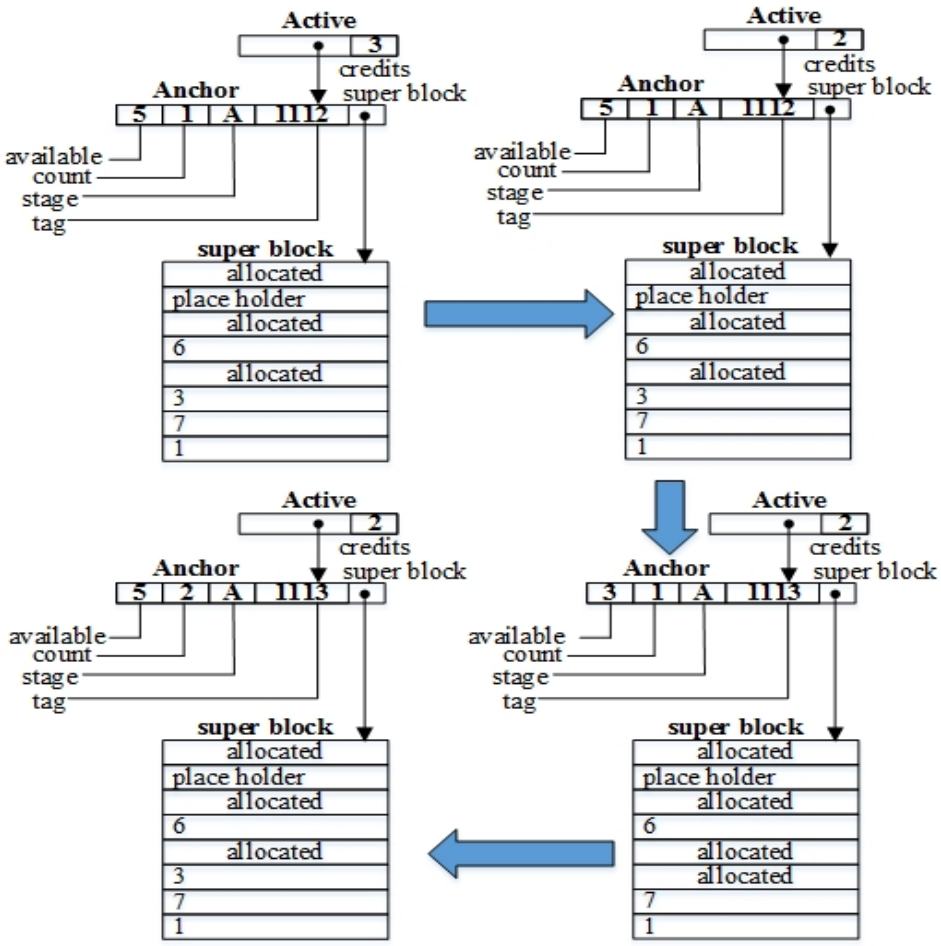

图 2.15 Michael 内存分配器

Fig. 2.15 Michael’s Dynamic Memory Allocator

然而，研究者们忽略了无锁数据结构动态内存管理的特殊问题，以及内存池在降低无锁数据结构动态内存消耗和降低无锁数据结构操作执行时间上的作用。对于动态无锁数据结构——无锁队列[90, 104]、无锁链表[80, 105, 106]、无锁栈[85, 88]、散列表[82, 91]——来说，它们的大部分操作都可以归结为将向系统申请动态内存构建新节点以执行插入操作，以及将数据结构的的节点删除并将内存归还给系统。与这些插入、删除操作相关的同步开销会随着竞争线程的增多而增加，继而引起数据结构吞吐量的下降和操作平均执行时间的增长，这一状况又为多核处理器引入的更多的可用线程而加剧。此时，内存分配器由于为所有线程可见，便成为瓶颈。

因而，对于无锁数据结构来说，除了使用更加高效的内存分配器之外，如何减少数据结构对动态内存的需求就显得尤为重要。内存池正可以解决这一问题：通过让线程使用私有和/或全局缓冲区，操作的内存需求可以先由私有和/或全局缓冲区处理，仅当缓冲区满或空时才使用全局的内存分配器，以减少对内存分配器并发访问的次数和频率（见图 2.16）。高效的内存池可以有效减少共享无锁数据结构的线程的相关动态内存消耗，间接降低无锁数据结构操作的平均执行时间。

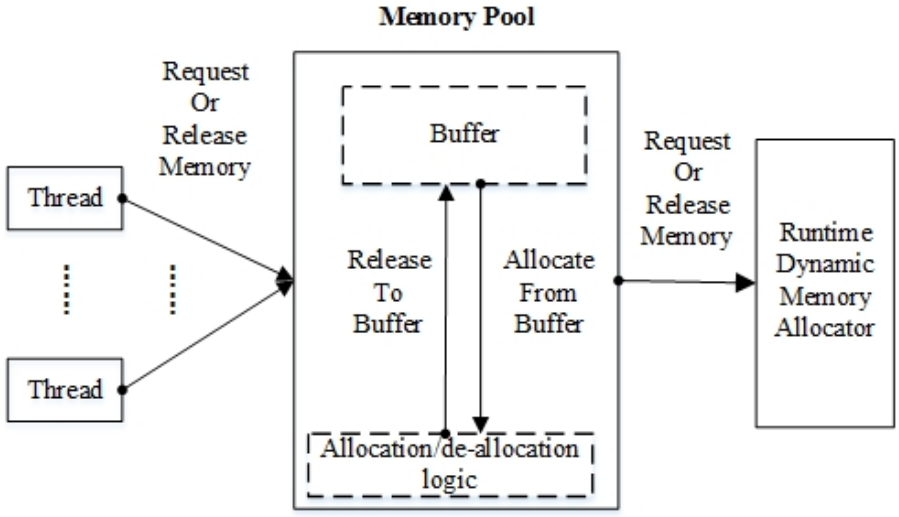

图 2.16 程序通过内存池与堆交互

Fig. 2.16 Interaction Between Application and Heap Through Memory Pool

2.3 负载均衡及资源均衡的动机

对于拥有多个执行线程/进程/计算节点的在线和/或分布式系统而言，负载在各个执行单元之间的分配直接影响系统的吞吐率和响应速度：不均衡的负载分布会导致计算单元的闲置和计算资源的浪费，降低系统的吞吐量和响应速度，这对协同工作的计算单元则更为严重——计算任务的总体进度取决于最慢的计算单元，负载不均衡导致某些计算单元负担过重而拖慢整个任务的进度（见图 2.17）。在线

系统的负载动态到达且不可预知，无法获得负载的总体分布，更是加重了负载均衡的难度和代价。

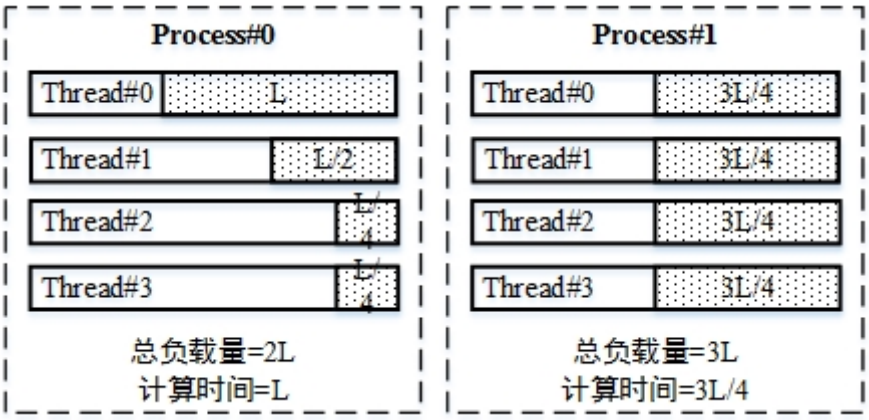

图 2.17 负载均衡与不均衡的对比

Fig. 2.17 Comparison Between Balanced and Unbalanced Load

负载均衡在共享内存多处理器/多核处理器系统中的研究主要集中于多线程/进程的高效调度以及任务在多线程/进程间的均衡分配以提高系统的吞吐率和响应能力。现有的负载均衡算法可以分为两类：

① 任务窃取（Work Stealing）：负载低的线程以一个一致/非一致的概率选定一个目标线程并试图从该线程处窃取一个/数个负载，

② 任务分担（Work Sharing）：线程以一个与自身负载量成正/反比的概率执行分担操作，选择一个随机目标线程并在这两个线程之间平分负载。

任务窃取最早由 Burton 和 Sleep[107]在对函数式程序并行执行的研究中提出，允许空闲处理器窃取相邻处理器的负载以解决组织成虚拟树的处理器网络中的的任务分派和负载均衡问题。Blumofe 和 Leiserson [4]提出了更为实用的在线随机任务窃取多线程调度算法，证明了该算法的时间和空间界，并指出任务窃取算法在通信效率上要高于任务分担算法。该算法将多个并行执行的线程分配到多个处理器上，每个处理器保持一个由线程数据结构组成的访问受限的双端队列——待执行队列（Ready Deque）。线程只能够被插入到待执行队列的尾部，但可以被从任意一端移除。处理器将属于自己的待执行队列看作一个调用栈，将线程压入队尾或者从队尾弹出，转移到其它处理器上的线程被从队列头部移除（见图 2.18）。初始时，所有处理器的待执行队列都为空，多线程计算任务的根线程被放入某个选定处理器的待执行队列，随后，其它处理器开始执行任务窃取。执行任务窃取的处理器随机选择一个目标处理器并检查其待执行队列，根据目标处理器待处理队列是否为空，执行任务窃取的处理器分别作如下处理：

① 目标处理器的待执行队列不为空：移除位于队列头部的线程结构，并开始该线程的执行，

② 目标处理器的待执行队列为空：随机另外选取一个目标执行任务窃取，直到窃取成功为止。

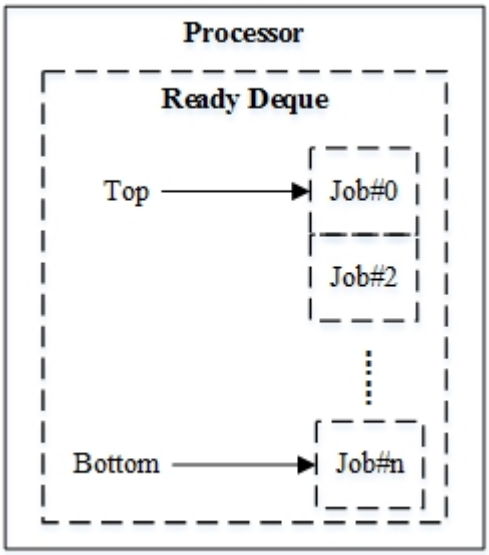

图 2.18 处理器待处理队列结构

Fig. 2.18 Ready Queue for a Processor

Arora 等人[5]给出了前述算法的一个无锁实现（ABP），并放宽了对任务类型和执行环境的限制，并用一个基于数组的有界无锁双端队列实现待执行队列以实现无锁性。该无锁双端队列（见图 2.19）由指向线程结构的指针数组、可以被 CAS 操作修改的头部和尾部组成。对头部和尾部的弹出操作使用 CAS 指令以实现原子性，同时，为避免 CAS 操作可能引发的 ABA 问题，头部包含一个标识域并被复合在一个存储器字之内以适合 CAS 操作的要求。通过让正常的 push/pop 操作和窃取操作分别在队列的两端执行，pop 和 steal 操作仅在观察到队列中剩余元素数量为一的时候才需要使用 CAS 操作来确保一致性，其他时间仅使用普通读写操作，这显著减少了窃取操作的开销。该算法的性能已被实际使用和实验所验证[108]，但有界数组的使用会导致溢出问题，这在动态性较强的多核处理器系统中更容易产生，限制了它的应用范围及在目标环境中的有效性。

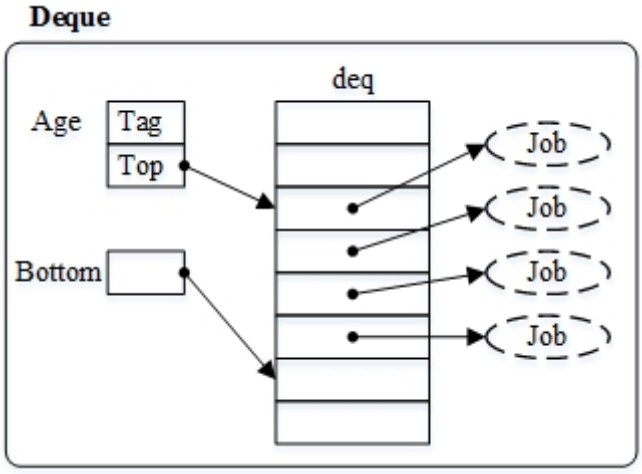

图 2.19 无锁双端队列

Fig. 2.19 Lock Free Deque

Handler 和 Shavit [10]改进了 Arora 等人的算法，使用扩展双端队列（Extented Deque），在实现一次窃取操作窃取目标队列中一半的元素的同时仍然保持操作的无锁性，并证明该算法能够更为快速地实现平衡。扩展双端队列使用循环数组而非固定大小的数组，并将队列头、标识和窃取位置标志复合到一个称为“窃取范围”（Steal Range）的连续空间中以满足 CAS 操作对内存字长度的限制并实现窃取一般元素（见图 2.20）。每个线程在本地保存自身上一次成功执行窃取操作时的“窃取范围”以判断是否有窃取操作在自身之前执行成功，执行窃取操作时，线程读取目标队列当前的“窃取范围”并试图用 CAS 操作更新之以完成窃取。

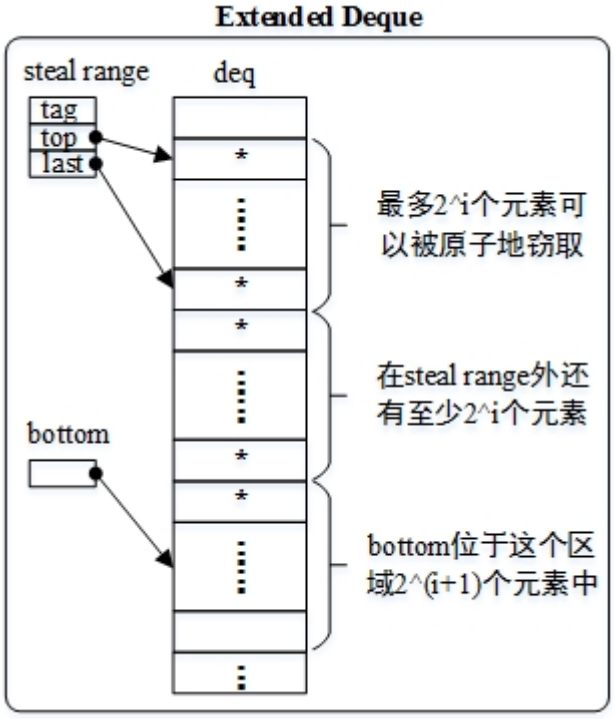

图 2.20 扩展双端队列

Fig. 2.20 Extended Deque

Handler 等人[109]提出了第一个动态内存无锁任务窃取队列, 该队列使用无锁双端队列解决 ABP 算法使用有界数组可能导致的溢出问题。该队列使用特殊设计的链接结构并限制头/尾指针的交互方式, 使得仍然可以使用与 ABP 算法相似方式对头/尾指针做比较来判断队列是否为空, 以将窃取操作对 CAS 指令的使用限制在队列最多只有一个元素的时候，同时，线程对私有队列的本地操作仅使用普通的读写指令（见图 2.21）。

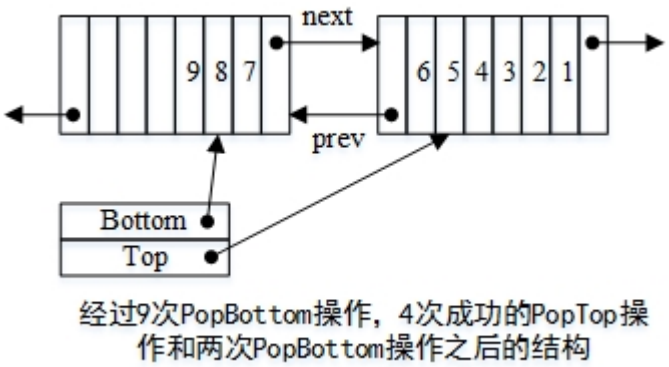

图 2.21 Handler 的任务窃取队列

Fig. 2.21 Handler’s Working Stealing Queue

Chase 和 Lev [11]提出了一个更为实用的无锁任务窃取队列, 该队列使用动态循环数组（dynamic cyclic array），既解决了有界数组的溢出问题又避免了大量动态内存的分配和释放，又能够提供更好的性能。该实现不需要在头部包含标志域，队列的容量可以通过头部和尾部的相对位置动态地计算，并可以动态、无锁地增加/缩减数组的大小以应对变化的问题规模（见图 2.22）：循环数组的尾部表示下一个压入操作将要压入元素的位置，且每次压入操作将尾部的值增一；头部标记当前位于队列头部元素的位置，每次窃取操作将其头部的值减一。

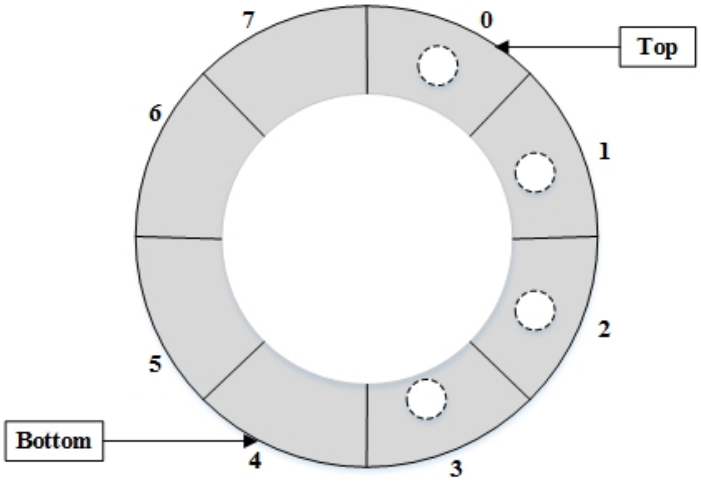

图 2.22 动态无锁循环任务窃取队列

Fig. 2.22 Dynamic Cirucular Lock Free Working Stealing Queue

Rudolph 等人[3]提出了基于任务分担思想的算法以解决多处理器系统的负载均衡问题。该算法为每个处理器维持一个本地负载堆，每次访问本地负载堆时，处理器以一个与本地负载堆中当前负载数量成反比的概率决定是否执行任务分担。执行任务分担的处理器随机地选择一个目标处理器，并将负载与该目标处理器分担使两个处理器的本地负载堆中的负载数量相等（见图 2.23）。

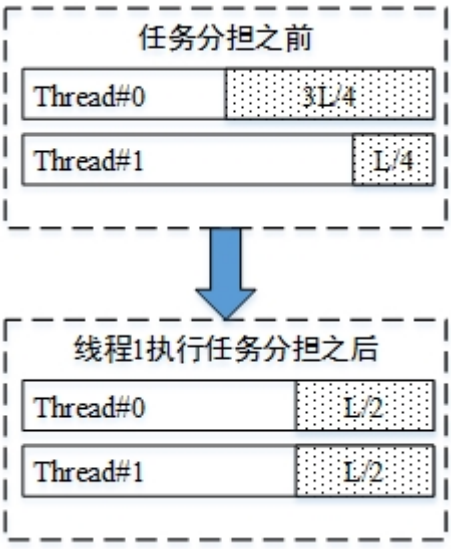

图 2.23 任务分担示意图

Fig. 2.23 Sementic Demenstration of Work Sharing

现有负载均衡研究关注吞吐率、响应性等宏观性能，并以应用程序、机器等高层实体为研究对象，以任务等为调度单位，忽略了更为低层次的负载均衡问题。事实上，多线程对内存的需求也即对内存分配器的使用，同样可以抽象为“逆向”负载均衡问题：内存分配器可以看作“逆向”任务产生器，各线程可以看作“逆向”任务执行器，分配/释放内存操作可以看作“逆向”负载，“逆向”负载均衡即是让内存分配器产生的“逆向”内存分配/释放操作均衡地分配到各“逆向”任务执行器——线程上（见图 2.24）。该“逆向”负载均衡问题可以通过首先将“逆向”负载通过负载池分配到线程的本地缓冲区中，各个线程在执行“逆向”负载时向负载池申请负载，负载池管理各个线程的本地缓冲区并提供平衡操作以将“逆向”负载在各个线程间均衡地分布，从而解决了“逆向”负载均衡问题。相对应地，无锁数据结构的内存池可以承担“逆向”负载池的作用，线程对无锁数据结构操作的内存需求可以由内存池满足，内存池管理线程的本地缓冲区并适时地在线程的本地缓冲区间执行负载均衡操作以尽可能地减少对内存分配器的使用。如此一来，便可以用负载均衡的算法来解决无锁数据结构相关的动态内存管理问题，使用带有负载均衡功能的内存池，就可以将无锁数据结构操作的动态内存操作尽可能地在本地解决，减少了动态内存消耗和由之引发的对内存分配器的压力，消弱了在多核处理器系统下内存分配器的瓶颈效应，减轻了系统内存的消耗，并间接减少操作的平均执行时间。

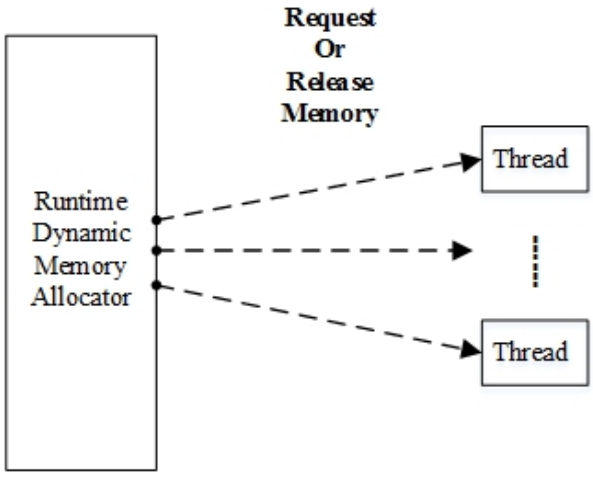

图 2.24 逆向负载均衡示意图

Fig. 2.24 Sementic Demenstration of Backward Load Balancing

2.4 本章小结

本章从并发数据结构、动态内存管理、负载均衡这几个方面对研究的背景进行了介绍。在并发数据结构领域，本章介绍该领域广泛使用的范式、同步方法和同步结构、在多核时代面临的挑战以及该领域在多核时代的最新成果和常用技术。在动态内存管理领域，本章列举了该领域的最新研究成果及共享内存多核处理器带来的更多的竞争线程在共享数据结构时对动态内存的大量消耗及其给内存分配其造成的压力，并指出了现有内存池在减少动态内存消耗方面的不足。在负载均衡领域，本章列举和分析了该领域的历史及最新研究成果，并将共享数据结构的线程对数据结构的操作相关的动态内存管理抽象为“逆向”负载均衡问题, 提出用负载均衡领域的研究成果与内存池技术结合以解决无锁数据结构相关的动态内存管理问题。

3 无锁资源窃取队列及无锁资源窃取内存池

内存池通常有两种，基于公共自由链表（见图 3.1）的以及使用线程私有的自由链表（见图 3.2）的。对于前者而言，在线程数量增多时公共自由链表就成为串行瓶颈——对公共自由链表的访问涉及的大量加锁/解锁操作会显著增加操作的开销。使用线程私有的自由链表可以消除此瓶颈：线程访问各自私有的自由链表，从而免于对公共资源的并发访问而导致的大量同步操作。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（比如操作系统调度、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，其它线程却可能仍有富余，从而导致线程对资源使用的不均衡。

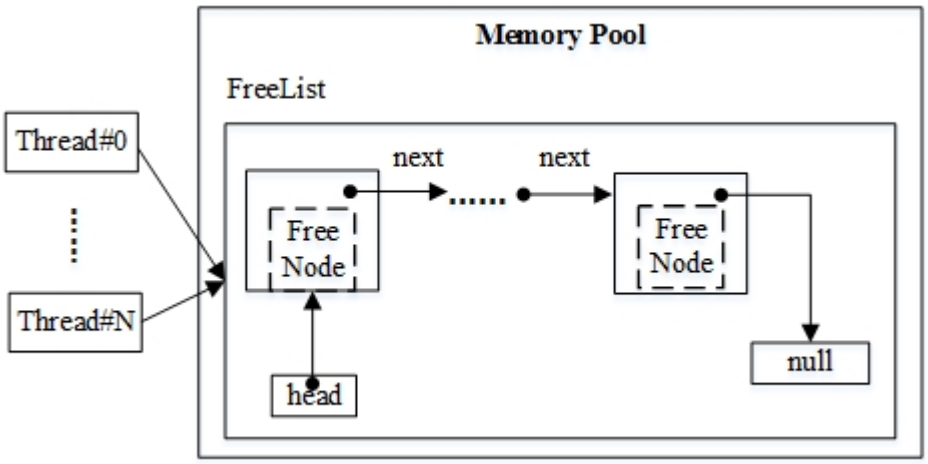

图 3.1 仅使用公共自由链表的内存池

Fig. 3.1 Memory Pool Based on Shared Free List

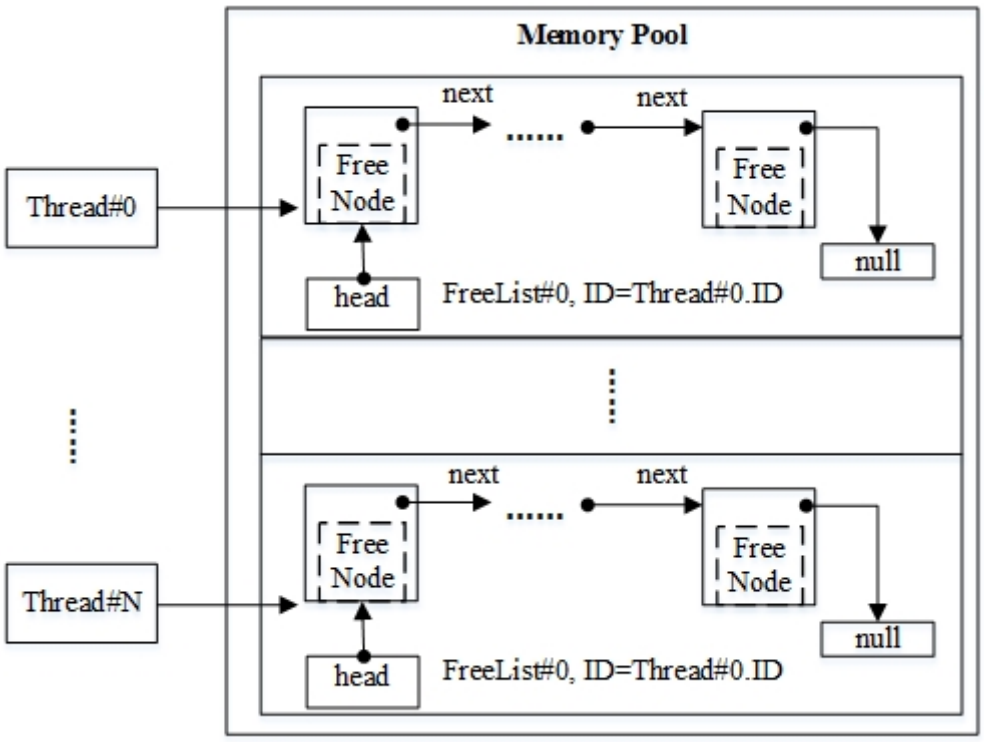

图 3.2 仅使用线程私有自由链表的内存池

Fig. 3.2 Memory Pool Based on Purely Thread Local Free List

基于资源窃取技术构建的内存池（见图 3.3）正可以解决这一问题：使用支持窃取操作的无锁资源窃取队列实现线程私有缓冲区，线程的动态内存需求将提交给内存池而非直接访问私有缓冲区和/或内存分配器；在处理线程的内存申请时，内存池调用队列提供的窃取操作在线程私有缓冲区之间协调以实现内存资源在各线程间的均衡分配。根据线程私有缓冲区是否为空， 内存池工作如下：

① 若线程私有缓冲区不为空，内存池则直接从中获取一个自由节点并返还给线程；

② 若线程缓冲区为空，内存池会随机选定一个目标线程并试图从该目标线程的私有缓冲区中窃取一个节点，若窃取成功，则将窃取到的节点返回给调用线程，若窃取失败则重新随机选定一个目标线程并尝试窃取，这一过程可以重复多次直到用尽尝试机会，此时内存池会访问动态内存分配器并申请一个新节点返回给调用线程。

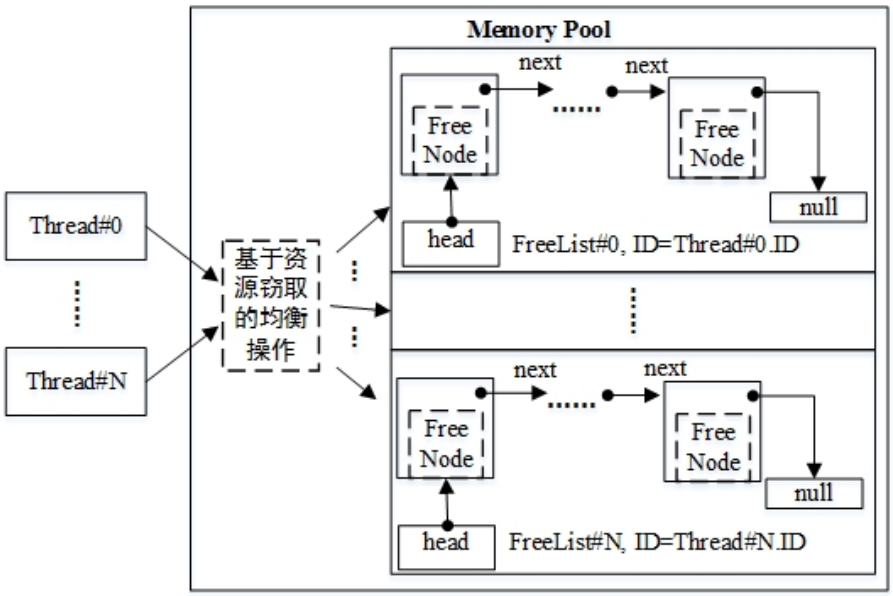

图 3.3 资源窃取内存池示意图

Fig. 3.3 Resource Stealing Memory Pool

3.1 无锁资源窃取队列

作为线程私有缓冲区的无锁资源窃取队列与 Lev 和 Chase 的动态无锁循环队列[11]相似，为支持节点窃取操作(StealNode)，需要对动态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：

① Dequeue 方法返回一个自由节点， Dequeue 方法仅可以被拥有该队列的线程调用，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 操作来尝试获取该节点，其它时候，该方法只是简单返回队列底部(bottom)的自由节点，

② StealNode 方法用于支持窃取操作，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跟 Dequeue 方法或其它线程在该队列上的并发窃取操作产生冲突，StealNode 总是试图获取位于目标队列顶部(top)的节点，

③ Enqueue 方法将一个可以安全释放的自由节点存入线程私有队列，该方法总是将新节点存入队列的底部，

④ ReFill 方法向操作系统申请一定数量的节点，由于此方法总是在调用线程窃取失败后调用，所以该方法会直接返回一个新节点给调用线程。

3.1.1 结构定义

资源窃取型无锁循环队列的含有 4 个域：Freenodes，用来存放自由节点；Bottom，用来标识队列的底端；Top，用来表示队列的顶端；Size 用来表示队列容量。具体如下：

类型定义. ResourceStealLFCircularQueue

```
STRUCT nodeStealLFCircularQueue
    int Bottom
    long Top
    int Size
    Node* Freenodes
END STRUCT
```

图 3.4 资源窃取队列的形式化定义

Fig. 3.4 Definition of the Resource Steal Circular Queue

由于不同的动态无锁数据结构使用的节点(node)各有不同, 无法固定节点类型, 基于绝大多数动态无锁数据结构，本文给出一个一般的节点定义，该节点含有两个域：DataElement，存放数据；Next，存放指向下个节点的指针，具体如下：

类型定义. Node

```
STRUCT Node
    Data DataElement
    Node* Next
END STRUCT
```

图 3.5 节点的形式化定义

Fig. 3.5 Definition of the Node

3.1.2 出队（Dequeue）

由于 Dequeue 方法仅有一个调用者，在大部分情况下都使用简单的队列操作, 只有当队列中仅剩一个自由节点时才可能跟并发调用的 StealNode 方法冲突, 此时需要使用开销较大的 CAS 操作以确保正确性。在此给出其伪代码描述（见图 3.6）。

算法 Dequeue

Input: localQueue

Output: result

```
D1:   localQueue.Bottom=localQueue.Bottom-1
D2:   oldTop=localQueue.Top
```

```
D3:    leftover=localQueue.Bottom-oldTop
D4:    IF leftover>0
D5:        result=localQueue.Freenodes[localQueue.Bottom%localQueue.Size]
D6:    ELSE IF leftover==0
D7:        IF CAS(&localQueue.Top, oldTop, oldTop+1)==TRUE
D8:            result=localQueue.Freenodes[localQueue.Bottom%localQueue.Size]
D9:        ELSE
D10:           result=NULL
D11:           localQueue.Bottom=oldTop+1
D12:       END IF
D13:   ELSE
D14:       result=NULL
D15:       localQueue.Bottom=oldTop
D16:   END IF
D17:   RETURN result
```

图 3.6 资源窃取队列的出队（Dequeue）方法

Fig. 3.6 Pseudo Code for Dequeue Method

3.1.3 节点窃取（StealNode）

StealNode 方法用来供其它线程调用以从目标队列中窃取一个自由节点，该方法通过试图在目标队列的 Top 域上执行一个 CAS 操作来实现节点窃取。由于 StealNode 方法可以同时被多个线程调用, 所以需要使用 CAS 操作以避免在目标队列上的调用的多个重叠的 StealNode 方法和/或 Dequeue 方法之间的并发访问冲突而导致错误。StealNode 方法首先获取目标队列当前容量的快照，若快照显示目标队列当前为空，则将空值作为结果返回给调用线程表示窃取操作失败，若不为空则尝试用 CAS 操作将目标队列得 Top 域值加一，若 CAS 操作成功则目标队列当前位于顶部的自由节点作为结果返回给调用线程, 若 CAS 操作失败则返回空值（见图 3.7）。

算法　StealNode

Input: victimQueue

Output: result

```
S1:    oldTop=victimQueue.Top
```

```
S2:    leftover=victimQueue.Bottom-oldTop
S3:    IF leftover<=0
S4:        result=NULL
S5:    ELSE
S6:        IF CAS(&victimQueue.Top, oldTop, oldTop+1)==TRUE
S7:            result=victimQueue.Freenodes[victimQueue.Top%victimQueue.Size]
S8:        ELSE
S9:            result=NULL
S10:       END IF
S11:   END IF
S12:   RETURN result
```

图 3.7 资源窃取队列的窃取（StealNode）方法

Fig. 3.7 Pseudo Code for StealNode Method

3.1.4 入队（Enqueue）

Enqueue 方法用来用来将一个可以安全释放的节点存入队列之中，该方法仅能被队列的拥有者调用不必考虑并发访问冲突问题，因而无需 CAS 操作即可实现：将待释放节点放入队列底部并将队列得 Bottom 域值减一即可（伪代码见图 3.8）。

算法　Enqueue

Input: localQueue, node

Output: result

```
E1:    result=FALSE
E2:    oldTop=localQueue.Top
E3:    size=localQueue.Bottom-oldTop
E4:    IF size<localQueue.Size-1
E5:        localQueue.Freenodes[localQueue.Bottom%localQueue.Size]=node
E6:        localQueue.Bottom=localQueue.Bottom+1
E7:        result=TRUE
E8:    END IF
E9:    RETURN result
```

图 3.8 资源窃取队列的入队（Enqueue）方法

Fig. 3.8 Pseudo Code for Enqueue Method

3.1.5 填充（ReFill）

ReFill 方法向操作系统申请一定数量（通常与队列最大容量相等）的节点存入队列中以供线程申请和窃取，同时，由于调用线程总是在窃取失败之后才调用自身私有队列的 ReFill 方法，因此，该方法还会向调用线程返回一个自由节点。类似地，ReFill 方法也只能被拥有该队列的线程调用，所以同样不必考虑并发访问冲突问题。其伪代码描述见图 3.9。

算法　ReFill

Input: localQueue

Output: result

```
R1:  bottomTmp=localQueue.Bottom
R2:  FOR i=0 TO localQueue.Size-1
R3:      localQueue.Freenodes[bottomTmp%localQueue.Size]=malloc(sizeof(node))
R4:      bottomTmp=bottomTmp+1
R5:  END FOR
R6:  bottomTmp=bottomTmp-1
R7:  result=localQueue.Freenodes[bottomTmp%localQueue.Size]
R8:  localQueue.Bottom=bottomTmp
R9:  RETURN result
```

图 3.9 资源窃取队列的填充（ReFill）方法

Fig. 3.9 Pseudo Code for ReFill Method

3.2 无锁资源窃取内存池

使用资源窃取技术的无锁内存池算法提供两个方法：GetNewNode 和 Retirenode。GetNewNode 用来从内存池中返回一个新节点；而 RetireNode 则将一个节点返回给内存池。本文用线程私有的支持节点窃取的无锁循环队列实现线程私有缓冲区以减少窃取操作的开销，同时，使用资源窃取内存池的动态无锁数据结构还需要集成 Michael 的风险指针（Hazard Pointers）[83]以确保多线程环境下动态内存的安全回收问题。

3.2.1 结构定义

内存池对象含有 3 个域：PrivateQueues，用来存放各线程私有的资源窃取型无锁循环队列；MaxTries，线程偷窃节点的最大尝试次数；NumTreads，表示当前线程数。具体如下：

类型定义. NSPool

```
STRUCT NSPool
    nodeStealLFCircularQueue* PrivateQueues
    int MaxTries
    int NumThreads
END STRUCT
```

图 3.10 资源窃取内存池的形式化定义

Fig. 3.10 Definition for the Resource Steal Pool

3.2.2 获取新节点（GetNewNode）

线程调用 GetNewNode 方法来从内存池获取一个新节点, GetNewNode 方法试图从线程的私有队列中获取一个自由节点，若失败则试着从其它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，具体步骤如下（伪代码实现见图 3.11）：

① 获取当前线程(CurrentThread)的私有队列，看其是否为空，若为空则令 AttemptionCount=0 并转到步骤 2，否则从 CurrentThread 的私有队列(localQueue)中返回一个自由节点（调用该 localQueue 的 Dequeue 方法），

② 若 AttemptionCount>MaxTries 则转到步骤 5，否则随机选定一个目标线程(VictimThread)并获取其 ThreadID,

③ 若 VictimThread.ThreadID==CurrentThread.ThreadID，则令 AttemptionCount+=1，转到步骤 2，否则转到步骤 4,

④ 试着从 VictimThread 的 localQueue 中窃取一个自由节点（调用该 localQueue 的 StealNode 方法），若窃取成功则返回窃取到的节点，不成功则令 AttemptionCount+= 1 并转到步骤 2,

⑤ 向操作系统申请数量为 FreenodesCount 的节点，将它们填充到 CurrentThread 的 localQueue 中, 从该 localQueue 获取一个节点（调用该 localQueue 的 Dequeue 方法），并将此节点返回给调用线程。

算法　GetNewNode

Input: MemoryPool, DataElement, ThreadID

Output: result

```
G1:   localQueue=MemoryPool.PrivateQueues[ThreadID]
G2:   result= Dequeue(localQueue)
G3:   IF result==NULL
G4:       AttemptionCount=0
G5:       WHILE result==NULL AND AttemptionCount<MemoryPool.MaxTries
G6:           VictimID=rand()%MemoryPool.NumThreads
G7:           IF VictimID== ThreadID
G8:               CONTINUE
G9:           ELSE
G10:              VictimlocalQueue=MemoryPool.PrivateQueues[VictimID]
G11:              result= StealNode(VictimlocalQueue)
G12:          END IF
G13:       END WHILE
G14:  END IF
G15:  IF result==NULL
G16:      result= ReFill(localQueue)
G17:  END IF
G18:  result.DataElement=DataElement
G19:  result.Next=NULL
G20:  RETURN result
```

图 3.11 资源窃取内存池的获取新节点（GetNewNode）方法

Fig. 3.11 Pseudo Code for GetNewNode Method

3.2.3 释放节点（Retirenode）

线程调用 Retirenode 方法来试图将一个节点返回给内存池，该方法被集成在使用风险指针的动态无锁数据结构的 Scan 方法中，由 Scan 方法调用，以将可以安全释放的节点返回给内存池，或者在线程的私有队列已满时将可以安全释放的节点直接释放，返还给操作系统，具体步骤如下（伪代码实现见图 3.12）：

① 根据传入的线程 ID 获取该线程的私有队列，

② 调用私有队列的 Enqueue 方法尝试将节点放入私有队列中, 若成功则退出, 否则转步骤 3,

③ 直接通过动态内存分配器的 free 方法将节点内存释放。

算法　RetireNode

Input: MemoryPool, node, ThreadID

Output: None

```
R1:   localQueue=MemoryPool.PrivateQueues[ThreadID]
R2:   result=Enqueue(localQueue, node)
R3:   IF result==FALSE
R4:       free(node)
R5:   END IF
```

图 3.12 资源窃取队列的释放节点（RetireNode）方法

Fig. 3.12 Pseudo Code for RetireNode Method

3.3 证明

在下文中, 将对资源窃取无锁队列和资源窃取无锁内存池的安全性、相关操作的无锁性以及可线性化性给出证明。

为简便起见将简称 ResourceStealLFCircularQueue 为队列, 用 Queue(n)表示一个拥有 n 个自由节点的 ResourceStealLFCircularQueue。

3.3.1 安全性（Safety）

定理 3.13 StealNode 方法的执行不会导致队列进入不一致状态。

证明：通过对队列中的自由节点数量进行归纳来证明本定理。

基本步：由图 3.7 的行 S3 可知, 对于 Queue(0), StealNode 方法不改变其状态, 显然不会导致队列进入不一致状态；对于 Queue(1), 根据图 3.6 的行 D1, Dequeue 方法对 Bottom 的减一操作不是原子的, 正在进行的 StealNode 方法有可能观察不到该操作的后果而仍然试图对 Queue(1)施行操作, 但是, 由于 StealNode 使用 CAS 操作来试图改变 Top 的值且由算法 3.3 的行 D7 可知, Dequeue 方法在此时也对 Top 执行 CAS 操作, 因而：

① 若 StealNode 的 CAS 操作失败, 则 Queue(1)的状态未被其改变,

② 若 StealNode 的 CAS 操作成功, 则由 CAS 操作的性质可知, Dequeue 方法的 CAS 操作或其它同时进行的 StealNode 方法的 CAS 操作均失败, 得到 Queue(0)。

以上任一情况均不会导致队列进入不一致状态。

归纳步：对于任意 K>1，假设 StealNode 方法的执行不会导致 Queue(K)进入不一致状态，将证明，对 Queue(K+1)亦然。

StealNode 同过对 Queue(K+1)的 Top 执行 CAS 操作令其增加一来实现节点窃取，尽管 StealNode 可以被多个线程并发调用，但是，基于 CAS 操作的性质，Queue(K+1)的 Top 上的多个同时执行的 CAS 操作只有一个成功，得到 Queue(K)且不会导致队列进入不一致状态。

根据归纳法则，定理得证。

定理 3.14 Dequeue 方法不会导致队列进入不一致状态。

证明： 通过对队列中自由节点的数量作归纳来证明本定理。

基本步：对于 Queue(0)，由图 3.6 的行 D14 和 D15 可知，Dequeue 操作不会导致队列进入不一致状态；对于 Queue(1)：

③ 若在 Dequeue 方法执行过程中那个，Queue(1)上没有 StealNode 方法被调用，则 Dequeue 方法得到 Queue(0),

④ 若在 Dequeue 方法执行过程中，Queue(1)上有 StealNode 方法被调用，则根据算法图 3.6 的行 D7 可知，若 Dequeue 方法的 CAS 操作成功则得到 Queue(0)，若失败修改队列状态并返回空值。

以上任意两种情况均不会导致队列进入不一致状态。

归纳步：当 K>1，假设 Dequeue 方法的执行不会导致 Queue(K)进入不一致状态，将证明，对于 Queue(K+1)亦然。

由算法 3.3 的行 D4、D5 和算法 5 的行 S7 可知，当 K>1 时，Dequeue 方法仅仅改变 Queue(K+1)的 Bottom 值，并且不会跟 StealNode 方法产生冲突，因而 Dequeue 方法不会导致队列进入不一致状态。

根据归纳法则，定理得证。

定理 3.15 ReFill 方法和 Enqueue 方法不会导致队列进入不一致状态。

证明：由图 3.9 和图 3.8 可知，ReFill 方法和 Enqueue 都只有一个调用者，且只改变队列的 Bottom 值，显然不会导致队列进入不一致状态。

3.3.2 无锁性（Lock Free）

若某个操作能保证，在它执行有限步之后，总有其它操作完成，则称该操作是无锁的 (Lock-free)[7]，在下面给出无锁性的简要证明。

定理 3.16 StealNode 方法是无锁的。

证明： StealNode 执行的唯一 CAS 操作位于图 3.7 的行 S7，由 CAS 操作的性质可知，只有两种情况会导致该处 CAS 操作的失败：

① K>1，对 Queue(K)，有多个 StealNode 方法试图从 Queue(K)窃取节点，且

这些 StealNoe 方法中某一个的 CAS 操作与当前 StealNode 的 CAS 操作重叠，且成功更新了 Queue(K)的 Top 值，

② K=1，对 Queue(1)，当前 StealNode 方法和其他 StealNode 方法同时试图用 CAS 操作更新 Queue(1)的 Top 值或者除此之外还有一个 Dequeue 方法试图用 CAS 操作更新 Queue(1)的 Top 值，并且该 Dequeue 方法成功或其它 StealNode 方法之一的 CAS 操作成功。

以上任意一种情况下，当前 StealNode 方法的 CAS 操作失败都是其它方法 CAS 操作成功的结果，所以，定理得证。

定理 3.17 Dequeue 方法是无锁的。

证明： Dequeue 方法执行的唯一一个 CAS 操作位于图 3.6 的行 D7，导致该 CAS 操作失败的唯一原因是某个 StealNode 方法成功地用 CAS 操作功能更新 Queue(1)的 Top 值，也即，图 3.6 中行 D7 处 CAS 操作的失败是 StealNode 方法 CAS 操作成功的结果，所以，定理得证。

定理 3.18 GetNewNode 方法是无锁的。

证明： 由于图 3.11 的行 G5 处的循环有上限，而且循环内调用的 StealNode 方法本身是无锁的，所以，GetNewNode 方法也是无锁的。

3.3.3 可线性化性（Linearizability）

可线性化[58]提供了一种幻觉——并发进程/线程在对象上应用的操作是在其调用和获得响应之间的某个点上即刻生效的，也暗示着，并发对象的所提供的操作的意义可以用前置和后置条件[109, 110]给出。

资源窃取型内存池的状态就是组成它的各个线程私有的支持节点窃取的无锁循环队列的状态，因而这二者的可线性化是等价的，本文将只给出支持节点窃取的无锁循环队列的可线性化的证明。

支持节点窃取的无锁循环队列的每个操作都有确定的线性化点，通过之前的安全性证明，可以得出它的每个操作的响应都和队列在该操作的线性化点上的状态相一致。这些线性化点如下：

① 每个返回非空值的 Dequeue 方法的线性化点位于图 3.6 的行 D5 和 D8。

② 每个返回空值的 Dequeue 方法的线性化点位于图 3.6 的行 D10 和 D14。

③ 每个返回非空值的 StealNode 方法的线性化点位于图 3.7 的行 S7。

④ 每个返回空值的 StealNode 方法的线性化点位于图 3.7 的行 S4 和 S9。

定理 3.19 资源窃取窃取无锁循环队列是可线性化的。

证明： 由上述线性化点的定位可知，资源窃取窃取无锁循环队列的每个操作都有明确的线性化点，因而它是可线性化的。

定理 3.20 资源窃取内存池是可线性化的。

证明：由于作为资源窃取内存池的组成部分的每个资源窃取无锁循环队列都是可线性化的，因而资源窃取内存池也是可线性化的。

3.4 本章小结

本章提出了一个支持资源窃取的无锁循环队列并用该队列实现线程私有缓冲区而构建了一个供无锁数据结构使用的内存池，该内存池通过资源窃取无锁循环队列的窃取操作将无锁数据结构操作相关的动态内存分配在各个线程私有缓冲区间平衡，以减少共享无锁数据结构操作的动态内存消耗量和操作平均执行时间。同时，本章还给出了资源窃取无锁循环队列和资源窃取内存池相关操作的安全性、无锁性以及可线性化性的证明。

4 资源窃取与归还队列及资源均衡内存池

资源窃取内存池可以有效地平衡各个线程私有队列中的自由节点数量，将线程的内存需求尽量限制在本地，减少了对内存分配器的直接访问，这在共享内存多核处理器环境下可以大大消减内存分配器上的瓶颈效应。但是，除了动态内存申请，并发线程对共享无锁数据结构操作涉及的节点删除操作需要将内存归还给系统，这同样引起对内存分配器的访问，而资源窃取内存池却恰恰忽略了这一点，使得相当一部分的并发动态内存释放操作被直接提交给内存分配器（见图 4.1），降低了系统的性能和扩展性，在并发度高的情况下，这些直接到达内存分配器的内存释放操作带来的额外开销会随着并发线程的增多逐渐吞噬通过窃取操作达成的平衡条件带来的性能收益。

为此，本文在资源窃取队列和资源窃取内存池的基础上改进，提出了资源窃取和归还队列，并用该队列构建了资源窃取和归还内存池来更好地平衡无锁数据结构的内存消耗。资源窃取和归还队列除了允许窃取操作外，还支持归还操作，基于该队列构建的资源窃取和归还内存池允许线程将待释放节点归还到其它线程的私有队列中，以减少由内存分配器直接处理的内存释放操作的数量（见图 4.2），如此一来，内存池的性能和扩展性都得到了增强。

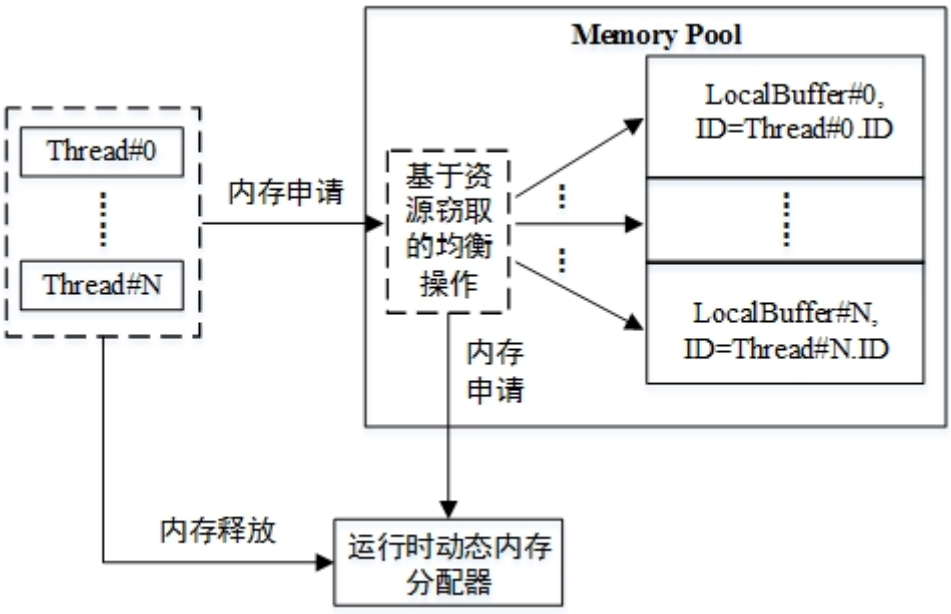

图 4.1 未经平衡的内存释放操作与经过平衡的内存申请操作

Fig. 4.1 Comparison Between Balanced and Unbalanced De-allocation Operation

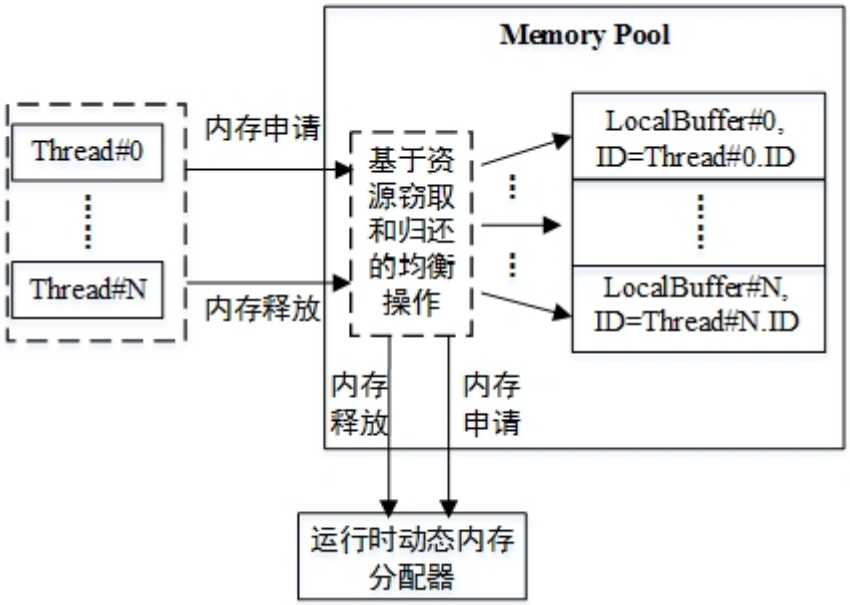

图 4.2 经过平衡的内存释放/申请操作

Fig. 4.2 Balanced Dynamic Memory Allocation/De-allocation Operation

4.1 资源窃取队列（Resource Stealing-Returning Queue，NSRQueue）

允许线程将节点归还到其它线程的私有队列中，给队列设计引入了额外的复杂性。由于队列顶部会被 StealNode 方法并发地访问，直接让归还操作将待释放节点归还到队列顶部将会大大加剧多线程对队列顶部的竞争，在队列顶部产生顺序访问瓶颈，随着并发线程的增多，该瓶颈会持续吞噬平衡操作带来的性能收益并限制系统的扩展性。若令归还操作将待释放节点归还到队列底部，则会迫使入队和出队操作使用 CAS 操作以确保多线程访问的安全性，这将改变设计的初衷——线程对私有队列的入队和出队操作比来自其他线程的窃取和/或归还操作更为频繁，因而它们的开销要尽可能的低——得不偿失。解决的方法在于分布和对队列语义的放松：Shavit[112]指出，在共享内存多核处理器广泛应用的时代，对数据结构并发度和扩展性要求越来越高。随着处理器核数/可用线程数的增加，数据结构的各个方面：一致性条件、活性条件、结构分布的水平都必须有所放松，以满足程序对数据结构并发度和扩展性的要求。

为此，本文提出了新的队列设计方案——资源窃取和归还队列（见图 4.3）——以支持窃取和归还操作，通过对队列语义的放松，该方案既满足了并发度又兼顾了扩展性。资源窃取和归还队列由两部分构成：Q1 和 Q2，这本质上是两个访问受限的独立队列，并提供 Dequeue、Enqueue、StealNode 和 ReturnNode 四个方法。

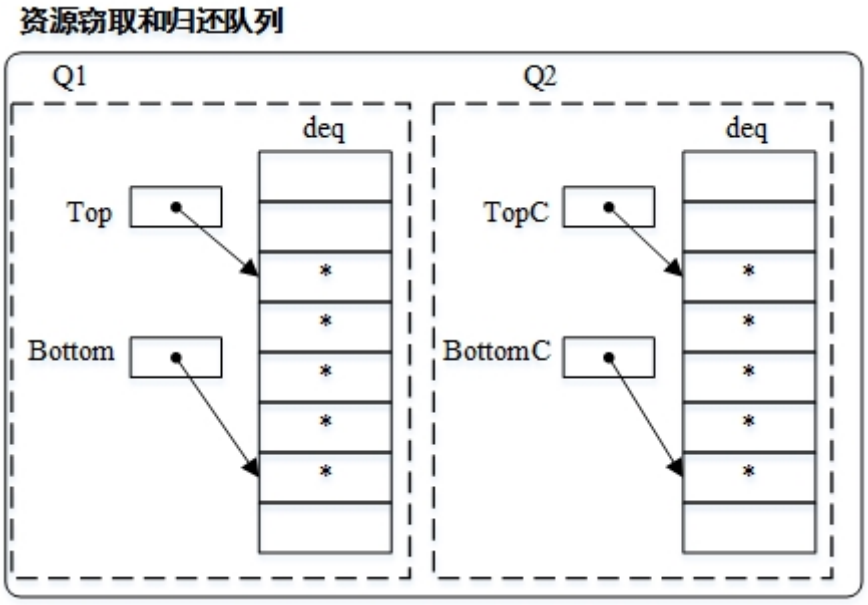

图 4.3 资源窃取和归还队列示意图

Fig. 4.3 Sementic Demenstration of Resourcing Stealing and Returning

Q1 与仅支持窃取操作的资源窃取队列相似，是一个基于数组的无锁双端循环队列，资源窃取和归还队列的 Dequeue、Enqueue、StealNode 三个方法会访问 Q1：Dequeue 方法不再是无锁的，它尝试获取位于 Q1 的 Bottom 处的节点并在失败时尝试从 Q2 处获取节点；Enqueue 方法尝试将一个待释放节点放置到 Q1 的底部；StealNode 方法仍然是无锁的，它试图窃取位于 Q1 的 Top 处的节点。

用于支持额外的归还操作的 Q2 则仅仅是一个基于数据的普通循环队列，资源窃取和归还队列的 Dequeue 和 ReturnNode 方法会访问 Q2：Dequeue 在获取 Q1 的 Bottom 处的节点失败后，会尝试获取 Q2 顶部的节点，为此，该方法会在 Q2 的 Top 处旋转直到获取位于该处的节点；ReturnNode 方法试图将一个待释放节点放置到目标队列的 Q2 域的底部。

4.1.1 结构定义

资源窃取和归还队列（Resource Stealing-Returning）支持四个操作，并拥有八个数据域：

① Q1，一个基于数据的无锁双端循环队列，作用和资源窃取队列的 Freenodes 数组相似；

② Top, Q1 的顶部；

③ Bottom，Q1 的底部；

④ Size，Q1 的容量；

⑤ Q2，用于支持归还操作的基于数据的普通循环队列；

⑥ TopC, Q2 的顶部；

⑦ BottomC, Q2 的底部；

⑧ SizeC, Q2 的容量。

给出资源窃取和归还队列的定义如下：

类型定义. NSRQueue

```
STRUCT NSRQueue
    node* Q1[]
    volatile long Top
    volatile long Bottom
    long Capacity
    node* Q2[]
    volatile long TopC
    volatile long BottomC
    long CapacityC
END
```

图 4.4 资源窃取和归还队列的形式化定义

Fig. 4.4 Definition of the Resource Steal and Return Queue

4.1.2 出队（Dequeue）

NSRQueue 实例上的 Dequeue 方法只能被该队列的拥有者线程调用, 这意味着该方法不会有并发调用，该方法分两个阶段运行：

① 在第一阶段，若 Q1 非空且位于 Q1 底部的节点被成功获取，则返回，否则进入第二阶段。当队列中有多余一个自由节点时，该方法对队列的修改仅限于队列的 Bottom 域，且修改操作仅使用普通的读写指令。当队列中只有一个自由节点时，该方法会尝试使用 CAS 操作修改队列的 Top 域以获取自由节点以避免来自其他线程的 StealNode 方法的并发调用引起的访问冲突。

② 在第二阶段，Dequeue 方法尝试从 Q2 处获取自由节点。若 Q2 非空，位于 Q2 顶端的自由节点将被返回给调用者，否则将返回 NULL 给调用者以表示获取节点失败。由于 Q2 同时还面临着来自 ReturnNode 方法的访问，该方法将一个待释放节点放入目标队列的 Q2 的底部——先原子地增加 Q2 的 Bottom 域并在成功后将待释放节点放入新的 Bottom 指向的位置。而对 Q2 的 Bottom 域值的增加和将待释放节点无法在一个原子操作内完成，于是，就可能会有一个时段，在此期间 Q2

的 Bottom 域值被来自 ReturnNode 方法的 CAS 操作成功增加但其指向的位置尚未被填上新值（见图 4.5）。因此，只有在 Q2 的 Top 指向的位置被填充上新值之后 Dequeue 方法才能获取它，此时，Dequeue 需要等待 ReturnNode 方法的对该值的写入操作的完成，在读取该值之后还需要将其重设为 NULL 以表明该值已经被成功获取。

图 4.6 给出了 Dequeue 方法的伪代码实现。

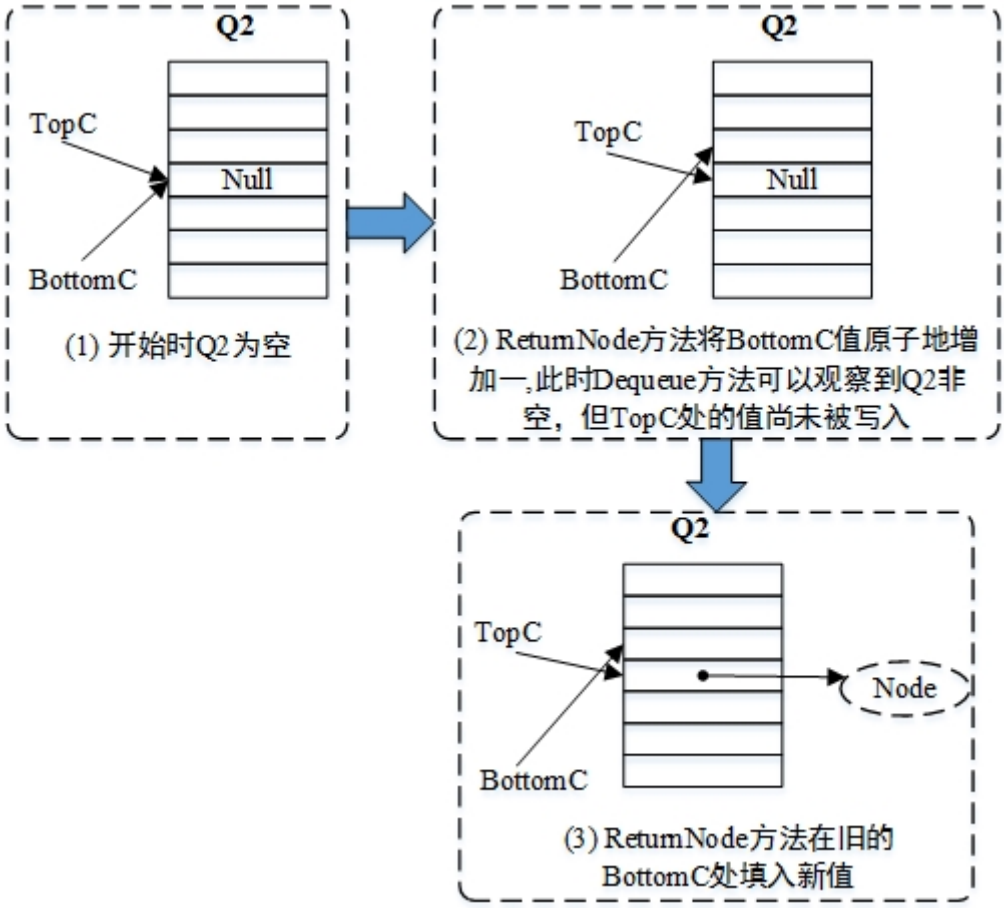

图 4.5 出队操作需要避免的错误

Fig. 4.5 The Error That Dequeue Operation Must Avoid

算法 Dequeue

Input: localQueue

Output: result

1: result=NULL

2: localQueue.Bottom-=1

3: oldTop=localQueue.Top

```
4:      size=localQueue.Bottom-oldTop
5:      IF size>0
6:          result=localQueue.Q1[localQueue.Bottom%localQueue.Capacity]
7:      ELSE IF size==0
8:          IF CAS(&localQueue.Top, oldTop, oldTop+1)==TRUE
9:              result=localQueue.Q1[localQueue.Bottom%localQueue.Capacity]
10:         ELSE
11:             oldBottomC=localQueue.BottomC-1
12:             size=oldBottomC-localQueue.TopC
13:             IF size>=0
14:                 WHILE(localQueue.Q2[localQueue.TopC%localQueue.CapacityC]== NULL)
15:                     spin
16:                 END WHILE
17:                 result=localQueue.Q2[localQueue.TopC%localQueue.CapacityC]
18:                 localQueue.Q2[localQueue.TopC%localQueue.CapacityC]=NULL
19:                 localQueue.TopC+=1
20:             END IF
21:         END IF
22:         localQueue.Bottom=localQueue.Top+1
23:     ELSE
24:         oldBottomC=localQueue.BottomC-1
25:         size=oldBottomC-localQueue.TopC
26:         IF size>=0
27:             WHILE(localQueue.Q2[localQueue.TopC%localQueue.CapacityC]==   NULL)
28:                 spin
29:             END WHILE
30:             result=localQueue.Q2[localQueue.TopC%localQueue.CapacityC]
31:             localQueue.Q2[localQueue.TopC%localQueue.CapacityC]=NULL
32:             localQueue.TopC+=1
33:         END IF
34:     END IF
35:     RETURN result
```

图 4.6 资源窃取和归还队列的出队（Dequeue）方法

Fig. 4.6 Pseudo Code for Dequeue Method

4.1.3 归还（ReturnNode）

ReturnNode 方法可以看作是 StealNode 方法的反向操作, 用于将一个待释放节点放置到目标队列 Q2 域的底部。由于在一个 NSRQueue 上可能有来自多个线程的对 ReturnNode 方法的并发调用, 该方法对队列的修改要使用 CAS 操作以避免多线程调用时的并发访问冲突。该方法首先获取 Q2 中剩余自由节点数量的快照，若 Q2 未满，则尝试用 CAS 操作原子地将 Q2 的 Bottom 域值加一，并在成功后将待释放节点放置在 Q2 的底部，否则返回 FALSE 表明失败，伪代码见图 4.7。

算法　ReturnNode

Input: destQueue, node

Output: result

```
1:    result=FALSE
2:    oldBottomC=destQueue.BottomC
3:    size=oldBottomC-destQueue.TopC-1
4:    IF size<destQueue.CapacityC-1
5:        IF CAS(&destQueue.BottomC, oldBottomC, oldBottomC+1)==TRUE
6:            destQueue.Q2[oldBottomC%destQueue.Capacity]=node
7:            result=TRUE
8:        END IF
9:    END IF
10:   RETURN result
```

图 4.7 资源窃取和归还队列的返还节点（ReturnNode）方法

Fig. 4.7 Pseudo Code for ReturnNode Method

4.1.4 窃取（StealNode）

NSRQueue 的 StealNode 操作与 NSQueue 的窃取操作类似，在目标队列 Q1 域非空时尝试获取位于 Q1 域顶部的自由节点，并使用 CAS 操作避免多线程调用时的并发访问冲突，伪代码见图 4.8。

算法　StealNode

Input: victimQueue

Output: result

```
1:  result=NULL
2:  oldTop=victimQueue.Top
3:  size=victimQueue.Bottom-oldTop
4:  IF size>=1
5:      IF CAS(&victimQueue.Top, oldTop, oldTop+1)==TRUE
6:          result=victimQueue.Q1[victimQueue.Top%victimQueue.Capacity]
7:      END IF
8:  END IF
9:  RETURN result
```

图 4.8 资源窃取和归还队列的的节点窃取（StealNode）方法

Fig. 4.8 Pseudo Code for StealNode Method

4.1.5 入队（Enqueue）

与 NSQueue 的 Enqueue 方法类似，NSRQueue 的 Enqueue 方法也只可以被该队列实例的拥有者线程调用。该方法尝试在 Q1 未满的时候将待释放队列放入 Q1 的底部，由于没有并发访问冲突，该操作仅使用普通的读写操作，伪代码实现见图 4.9。

算法　Enqueue

Input: localQueue, node

Output: Boolean result

```
1:  result=FALSE
2:  oldTop=localQueue.Top
3:  size=localQueue.Bottom-oldTop
4:  IF size<localQueue.Capacity-1
5:      localQueue.Q1[localQueue.Bottom%localQueue.Capacity]=node
6:      localQueue.Bottom+=1
7:      result=TRUE
8:  END IF
9:  RETURN result
```

图 4.9 资源窃取和归还队列的入队（Enqueue）方法

Fig. 4.9 Pseudo Code for Enqueue Method

4.1.6 再填充（ReFill）

资源窃取和归还队列的 ReFill 与资源窃取队列的 ReFill 方法相似, 这里不再赘述，仅给出其伪代码实现（见图 4.10）。

算法　ReFill

Input: localQueue

Output: result

```
R1:   bottomTmp=localQueue.Bottom
R2:   FOR i=0 TO localQueue.Size-1
R3:       localQueue.Q1[bottomTmp%localQueue.Size]=malloc(sizeof(node))
R4:       bottomTmp=bottomTmp+1
R5:   END FOR
R6:   bottomTmp=bottomTmp-1
R7:   result=localQueue.Freenodes[bottomTmp%localQueue.Size]
R8:   localQueue.Bottom=bottomTmp
R9:   RETURN result
```

图 4.10 资源窃取与归还队列的填充（ReFill）算法

Fig. 4.10 Pseudo Code for ReFill Method

4.2 资源均衡内存池(Resource Balancing Memory Pool，NSRPool)

资源均衡内存池（NSRPool）提供两个方法：GetNewNode，用于从内存池中获取一个新节点；RetireNode 方法，用于将一个待释放节点通过内存池释放。与仅使用窃取操作来平衡线程私有缓冲区中自由节点数目的 NSPool 相似，NSRPool 仍然为每个线程维持一个私有的本地缓冲区，该缓冲区用资源窃取和归还队列（NSRQueue）实现，与前者不同的是，RetireNode 方法在线程私有缓冲区上调用 Enqueue 失败后不是直接将待释放节点通过内存分配器释放, 而是选取一个目标线程并尝试将待释放节点返还到目标线程的私有缓冲区中，以减轻对内存分配器的压力并在高并发下降低其平静效应。

4.2.1 结构定义

资源均衡内存池（NSRPool）的结构与 NSPool 相似，区别仅在于线程私有缓冲区的实现以及配合 RetireNode 方法使用的归还操作最大尝试次数，此处使用支持归还节点的 NSRQueue 而非仅支持窃取节点的 NSPool，形式化定义如下。

类型定义. NSRPool

```
STRUCT NSPool
    NSRQueue LocalQueue[]
    int NumThreads
    int MaxStealTries
    int MaxReturnTries
END
```

图 4.11 资源均衡内存池的形式化定义

Fig. 4. 11 Semantic Definition for Resource Balance Memory Pool

4.2.2 获取新节点（GetNewNode）

资源均衡内存池（NSRPool）的 GetNewNode 方法与资源窃取内存池（NSPool）的类似，这里不再赘述，仅给出其伪代码实现（见图 4.12）。

算法 GetNewNode

Input: NSRPool,ThreadID

Output: result

```
1:  localQueue=nsrPool.LocalQueues[threadID]
2:      result=popBottom(nsrQueue)
3:      IF result==NULL
4:          numTried=0
5:          victimID=rand()%nsrPool.NumThreads
6:          WHILE result==NULL AND numTried<nsrPool.MaxTries
7:              victimQueue=nsrPool.LocalQueues[victimID]
8:              result=steal(victimQueue)
9:          END WHILE
10:         IF result==NULL
11:             result=ReFill(localQueue)
12:         END IF
13:     END IF
14:     RETURN result
```

图 4.12 资源均衡内存池的获取新节点（GetNewNode）方法

Fig. 4.12 Pseudo Code for GetNewNode Method

4.2.3 释放节点（RetireNode）

作为平衡操作的一部分，NSRPool 的 RetireNode 方法（伪代码实现见图 4.13）分三个阶段运作：

① 在第一阶段，通过调用线程的 ID 获取其私有 NSRQueue 实例并在该实例上调用 Dequeue 方法，若 Dequeue 方法返回 TRUE，则表明待释放节点已经被成功放置到线程私有缓冲区中，否则进入第二阶段；

② 在第二阶段——也即归还阶段，随机选取一个目标线程并在该线程的私有 NSRQueue 实例上调用 ReturnNode 方法尝试将待释放节点放置到该线程的私有缓冲区中，若 ReturnNode 方法返回 TRUE，表明成功将待释放节点放置到目标线程的私有缓冲区中，否则进入第三阶段。为提高性能，在实践中第二阶段会被重复一定次数以尽可能将待释放节点保留在内存池中，而非直接通过内存分配其释放给系统；

③ 在第三阶段，由于所有试图将待释放节点保留在内存池的的尝试都以失败，待释放节点将通过内存分配器释放给系统。

算法　　RetireNode

Input: nsrPool, threadID, node

Output: None

```
1:  localQueue=nsrPool.LocalQueues[threadID]
2:  result=popBottom(localQueue)
3:  IF result==FALSE
4:      numTried=0
5:      WHILE numTried<nsrPool.MaxTries AND result==FALSE
6:          dest=rand()%nsrPool.NumThreads
7:          destNSRQueue=nsrPool.LocalQueues[dest]
8:          result=destNSRQueue.retun(node)
9:      END WHILE
10:     IF result==FALSE
11:         free(node)
12:     END IF
13: END IF
```

图 4.13 资源均衡内存池的释放节点（RetireNode）方法

Fig. 4.13 Pseudo Code for RetireNode Method

4.3 证明

在下文中，将对资源窃取和归还队列和资源均衡内存池的安全性、相关操作的无锁性以及可线性化性给出证明。将简称具有 n 个自由节点的 NSRQueue 为 Queue(n)。

4.3.1 安全性（Safety）

定理 4.14 NSRQueue 的 StealNode 方法的不会导致目标队列进入不一致状态。

证明：由于 NSRQueue 的 StealNode 方法与 NSQueue 的 StealNode 方法本质上相同，由定理 3.13 可知 NSRQueue 的 StealNode 方法不会导致目标队列进入不一致状态。

定理 4.15 NSRQueue 的 Dequeue 方法不会导致队列进入不一致状态。

证明： NSRQueue 的 Dequeue 会修改队列实例的 Q1 和 Q2 两个域，其对 Q1 域的修改本质上与 NSPool 的 Dequeue 方法相同，由定理 3.14 可知对 Q1 的修改操作不会导致队列进入不一致状态。

下面证明对 Q2 的修改也不会导致队列进入不一致状态。

由于 Dequeue 方法仅能被一个线程调用，由图 4.6 的第 13 行可知，该方法不会进入无穷等待状态。由图 4.6 的第 16、17、18 行可知，该方法在获取 Q2 顶部的自由节点后 Q2 仍处于一致状态。由图 4.7 的第 4、5 行可知，ReturnNode 对方法对 Q2 的 Bottom 域的原子的增加不会越界，因而 Dequeue 方法对 Q2 的 Top 域的修改不会跟 ReturnNode 方法对 Q2 的 Bottom 域的访问相冲突。

综上所述，定理得证。

定理 4.16 NSRQueue 的 ReturnNode 方法不会导致队列进入不一致状态。

证明：通过对 Q2 中的自由节点数量进行归纳来证明本定理。

基本步：由图 4.7 的第 4 行可知，对于 Queue(NSRQueue.SizeC-1)，ReturnNode 方法不改变其状态，显然不会导致队列进入不一致状态。

归纳步：对于任意 n>=0 且 n<NSRQueue.SizeC-2，假设 ReturnNode 方法的执行不会导致 Queue(n)进入不一致状态，将证明，对 Queue(n+1)亦然。

ReturnNode 通过对 Queue(n+1)的 Button 执行 CAS 操作令其增加一来实现节点归还，尽管 ReturnNode 可以被多个线程并发调用，但是，基于 CAS 操作的性质，Queue(n)的 Bottom 上的多个同时执行的 CAS 操作只有一个成功，得到 Queue(n+1)，n+1<NSRQueue.SizeC-1，且不会导致队列进入不一致状态。

根据归纳法则，定理得证。

定理 4.17 NSRQueue 的 ReFill 方法和 Enqueue 方法不会导致队列进入不一致状态。

证明：由图 4.9 和图 4.10 可知，ReFill 方法和 Enqueue 都只有一个调用者，且

只改变队列的 Bottom 值，显然不会导致队列进入不一致状态。

4.3.2 无锁性（Lock Free）

定理 4.18 NSRQueue 的 StealNode 方法是无锁的。

证明：由于 NSRQueue 的 StealNode 方法与 NSQueue 的 StealNode 方法本质上相同，由定理 3.16 可知 NSRQueue 的 StealNode 也是无锁的。

定理 4.19 NSRQueue 的 ReturnNode 方法是无锁的。

证明： ReturnNode 方法执行的唯一一个 CAS 操作位于图 4.7 的第 5 行，导致该 CAS 操作失败的唯一原因是某个 ReturnNode 方法成功地用 CAS 操作更新 Queue(n)，n<NSRQueue.SizeC-1 的 Bottom 域值，也即，图 4.7 重第 5 行处 CAS 操作的失败是来自其他线程的对 ReturnNode 方法的并发调用中的某一个 CAS 操作成功的结果，所以，定理得证。

定理 4.20 NSRPool 的 RetireNode 方法是无锁的。

证明： 由于图 4.13 第 5 行处的循环有上限，而且循环内调用的 ReturnNode 方法本身是无锁的，所以，NSRPool 的 GetNewNode 方法也是无锁的。

4.3.3 可线性化（Linearizability）

资源窃取和归还队列的每个操作都有确定的线性化点，通过之前的安全性证明，可以得出它的每个操作的响应都和队列在该操作的线性化点上的状态相一致。这些线性化点如下：

① 每个返回非空值的 Dequeue 方法的线性化点位于图 4.6 第 6 行、第 9 行、第 17 行以及第 30 行。

② 每个返回空值的 Dequeue 方法的线性化点位于图 4.6 第 20 和第 33 行。

③ 每个返回非空值的 StealNode 方法的线性化点位于图 4.8 第 6 行。

④ 每个返回空值的 StealNode 方法的线性化点位于图 4.8 的第 7 行和第 8 行。

⑤ 每个返回值为 TRUE 的 ReturnNode 方法的线性化点位于图 4.7 第 7 行。

⑥ 每个返回至微 FALSE 的 ReturnNode 方法的线性化点位于图 4.7 第 8 行和第 9 行。

定理 4.21 资源窃取和归还队列是可线性化的。

证明： 由上述线性化点的定位可知，资源窃取和归还队列的每个操作都有明确的线性化点，因而它是可线性化的。

定理 4.22 资源均衡内存池是可线性化的。

证明： 由于作为资源均衡内存池的组成部分的每个线程私有资源窃取和归还队列都是可线性化的，因而资源均衡内存池也是可线性化的。

4.4 本章小结

为解决无锁数据结构操作相关的内存释放操作在高并发情况下产生的瓶颈问题，本章在改进资源窃取无锁循环队列的基础上提出了一个支持资源窃取和归还的循环队列并用该队列实现线程私有缓冲区而构建了一个供无锁数据结构使用的内存池，该内存池通过资源窃取和归还队列的窃取和归还方法将无锁数据结构操作相关的动态内存申请/释放在各个线程私有缓冲区间平衡，以减少共享无锁数据结构操作的动态内存消耗量和操作平均执行时间并提高内存池的扩展性。同时，本章还给出了资源窃取和归还队列和资源均衡内存池相关操作的安全性、无锁性以及可线性化性的证明。

5 实验结果及分析

为测量资源窃取内存池和资源均衡内存池的性能，实验使用著名的Michael&Scott 无锁队列[90]和无锁栈[83]。为使这两种数据结构避免使用动态内存时的 ABA 问题，本文用 Michael 的风险指针[83]——与 Herlihy 等人的 ROP 方案[84]本质上相同，对它们的实现进行了修改。在广泛的并发度下测量了使用这两种内存池和使用普通线程私有队列的内存池时无锁队列和无锁栈的平均内存消耗和操作平均执行时间并给出了对比结果。同时，实验还测量了资源窃取内存池和资源均衡内存池的平衡操作的性能。

5.1 实验方法及环境

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内存池和资源均衡内存池的性能及在高并发环境下的可扩展性。基准测试算法用 ANSI C 语言编写, 实验在 Microsoft Visual Studio 2010 Express Edition 环境下运行，三种内存池的实现均采用同样大小的线程私有队列，并使用同样的编译优化级别。操作系统是 Window 7 Professional Editon。平均动态内存消耗量用线程执行期间直接由系统内存分配器处理的动态内存申请/释放次数衡量，内存池的平衡操作的性能用运行完毕后各线程私有队列中剩余自由节点的数量跟平均值的方差来衡量。

本文采用 Herlihy 等人[113]的方法获取实验数据，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。直接由系统内存分配器处理的动态内存申请/释放数量用线程执行 100 万次随机操作的过程中调用 malloc 函数的平均次数表示，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，取 5 次结果的平均值作为最终结果。

5.2 集成方法

无锁数据结构广泛 CAS 操作来实现对内存字的原子访问, 尽管 CAS 操作足够高效（仅需数个 CPU 周期），但其缺陷也是明显的。在现存处理器体系结构下，CAS 操作仅能同时原子地访问/修改一个内存字, 这不仅限制了 CAS 操作的应用范围，增加了应用 CAS 操作设计、实现高性能并发数据结构的难度，还引入了 ABA 问题：多个并发线程尝试对共享数据结构施行 CAS 操作时，若其中一个（线程 T）观察到该内存处的值为 A，并以此发起 CAS 操作并在操作执行前被系统挂起，在

线程 T 挂起期间，其它线程先将该处内存的值先改为 B 随后再改回 A，随后线程 T 被系统调度执行，它发起的 CAS 操作观察到目标内存处的值为 A，因而 CAS 操作执行成功（尽管共享数据接构的状态已经发生了改变）——这极有可能破坏数据结构的一致性（见图 5.1）。

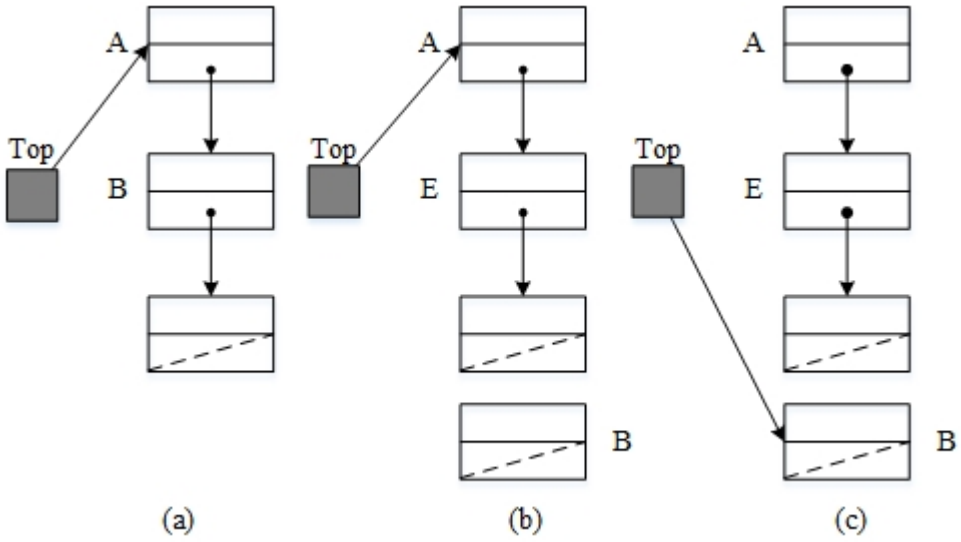

图 5.1 ABA 问题（a）起始时的无锁栈，此时线程 T1
正准备发起操作 CAS(&Top, A, B)，在操作执行前被操作系统挂起；
（b）线程 T2 将节点 A 和 B 删除并发到本地缓冲区以备重用，随后，
线程 T3 压入新节点 E，线程 T2 重用节点 A 并将其压入栈中；
（c）线程 T1 被调度执行，CAS 操作成功，使得 Top 指向已被删除
的节点 B，无锁栈的一致性被破坏。

Fig. 5.1 Sementic Demenstration of the ABA Problem

除了导致数据结构出现不一致之外，ABA 问题还使得无锁数据结构无法自由地使用动态内存：并发线程通过 CAS 操作试图修改内存区值的时候，并无法得知目标内存区的有效性——该区域可能已经被释放给系统，这时，CAS 操作对目标内存区的修改就会导致访问越界，甚至产生页面错误使系统崩溃（见图 5.2）。

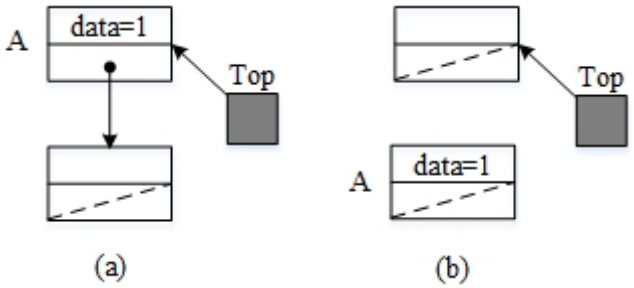

图 5.2 CAS 操作试图修改已释放内存（a）初始时线程 T1 试图
发起操作 CAS(&A.data, 1, value)，但在操作执行前被系统挂起；
（b）线程 T2 将 A 删除并释放给系统，随后 T1 被调度执行，
CAS 操作试图修改已被释放的内存，产生错误。

Fig. 5.2 How CAS Operation Leads to Error

Michael [83]（Hazard Pointer，风险指针）及 Herlihy 等人[84]（Repeat Offender Problem，ROP）分别独立地解决了这个问题，尽管名称不同，但他们的方法在本质上是相同的：给每个共享无锁数据结构的线程关联一些风险指针，每个线程的风险指针可以被其它线程读，但只能被该线程本身修改；线程把将要用 CAS 操作修改的指针记录在本地（风险指针），若某个线程要释放一个节点，该线程会将指向节点的指针积累在本地，达到一定规模（R）之后，就会将本地积累的待释放指针跟其它线程的风险指针进行交叉查询，未出现在其它线程风险指针中的就可以安全地被释放。

本文用风险指针来对无锁队列和无锁栈进行改造以使其能够避免 ABA 问题并安全使用动态内存, 为此, 在无锁链表和无锁栈中添加了改造的 RetireNode 和 Scan 方法（祥见 5.2.2 节），并修改了数据结构的操作以添加指针有效性检查。

5.2.1 对数据结构操作的修改

按照以下步骤对数据结构进行修改以兼容风险指针：

① 识别每个风险引用的创建点和最后使用点，这两点之间需要用风险指针对该引用加以保护，风险引用的数量即为所需要的风险指针的数量，

② 对每个风险引用，在其创建后及最后一个使用点之间创建一个风险指针并将引用值写入该风险指针，同时，在每次使用该引用时，都要先确认风险指针是安全的。

给出对无锁队列的修改作为例子（图 5.3 和 5.4），其中加黑的非斜体部分是为使用风险指针而添加的。

算法　改造后的 **Enqueue 方法**

Input: data, Queue

Output: None

```
1:   node=NewNode()
2:   node->data=data
3:   node->next=null
4:  WHILE TRUE
5:        t=Queue.Tail
6:        *hazardPointer0=t
7:        IF Queue.Tail!=t
8:            continue
9：       END IF
10:       next=t->next
11:      IF Queue.Tail!=t
12:           continue
13:      IF next!=null
14:           CAS(&Queue.Tail, t, next)
15:           continue
16:      END IF
17: END WHILE
18:  CAS(&Queue.Tail, t, node)
```

图 5.3 用风险指针改造后的 Enqueue 方法

Fig. 5.3 Pseudo Code for the Revised Enqueue Method

算法　　改造后的 Dequeue 方法

Input: Queue

Output: data

```
1:  WHILE TRUE
2:        h=Queue.Head
3:        *hazardPointer0=h
4:        IF Queue.Head!=h
5:            continue
6:        END IF
```

```
7:        t=Queue.Tail
8:        next=h->next
9:        *hazardPointer1=next
10:       IF Queue.Head!=h
11:           continue
12:       END IF
13:       IF next==null
14:           RETURN null
15:       END IF
16:     IF h==t
17:           CAS(&Queue.Tail, t, next)
18:           continue
19:       END IF
20:       data=next->data
21:       IF CAS(&Queue.Head, h, next)==TRUE
22:           break
23:       END IF
24:   END WHILE
27:   RetireNode(h)
26:   RETURN data
```

图 5.4 用风险指针改造后的 Dequeue 方法

Fig. 5.4 Pseudo Code for the Revised Dequeue Method

5.2.2 RetireNode 和 Scan

当线程从共享无锁数据结构中删除节点时，要先通过 RetireNode 方法将指向节点的指针在本地累积起来，当线程本地累积的待释放指针数达到阈值的时候，就调用 Scan 方法将本地累积的指针和其它线程持有的风险指针进行交叉查询，未出现在其它线程风险指针中的节点就可以被安全地释放，RetireNode 方法和 Scan 方法的伪代码实现分别见图 5.5 和 5.6。

算法　　RetireNode

Input: node

Output: None

```
1:    current_thread.priList.push(node)
2:    current_thread.priList.size++
2:    IF current_thread.priList.size>=R
3:        Scan(current_trhead.priList)
4:    END IF
```

图 5.5 RetireNode 方法的伪代码实现

Fig. 5.5 Pseudo Code for RetireNode Method

算法　　Scan

Input: priList

Output: None

```
1:    FOR EACH pointer n IN MemoryPool.hazard_pointers
2:        IF n!=NULL
3:            pList.insert(n)
4:        END IF
5:    END FOR
6:    FOR EACH pointer n IN privateList
7:        IF pList.Find(n)
8:          tmpList.push(n)
9:        ELSE
10:           free(n)
11:       END IF
12:   END FOR
```

图 5.6 Scan 方法的伪代码实现

Fig. 5.6 Pseudo Code for Scan Method

5.3 内存消耗

本文首先测量数据结构相关的线程平均动态内存消耗，结果如图 5.7 和 5.8。从图 5.7 和 5.8 可以看出，在多于一个线程的情况下，使用资源窃内存池和资源均衡内存池时线程的平均动态内存消耗显著低于使用普通内存池时的内存消耗量。特别是在使用资源均衡内存池时，线程的平均动态内存消耗量随竞争线程数量的增多——从 2 增加到 32——而持续降低。

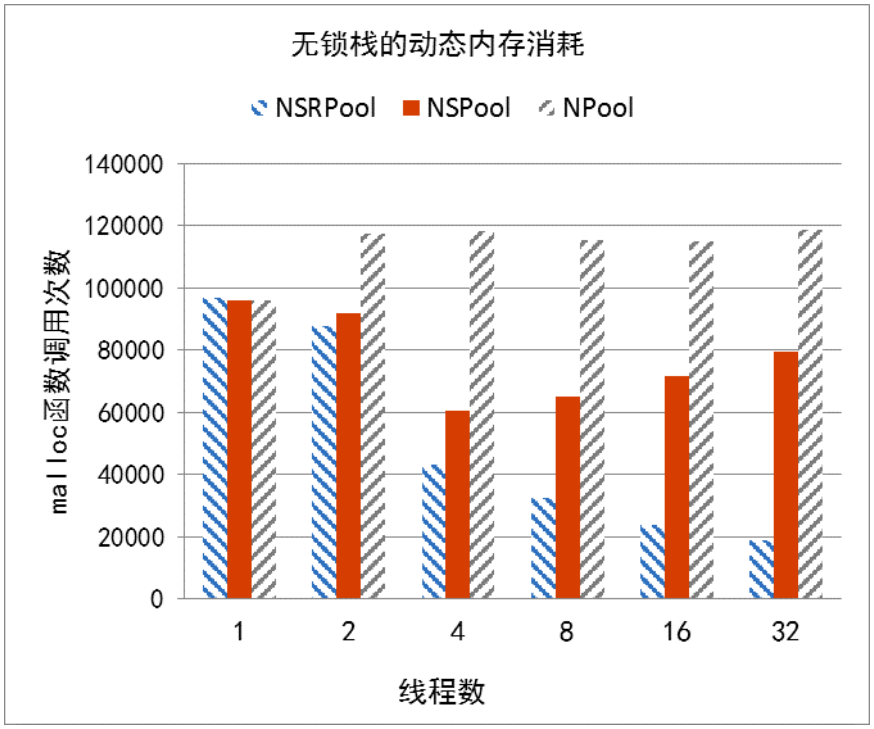

图 5.7 无锁栈的动态内存消耗情况

Fig. 5.7 Dynamic Memory Consumption of the Lock Free Stack

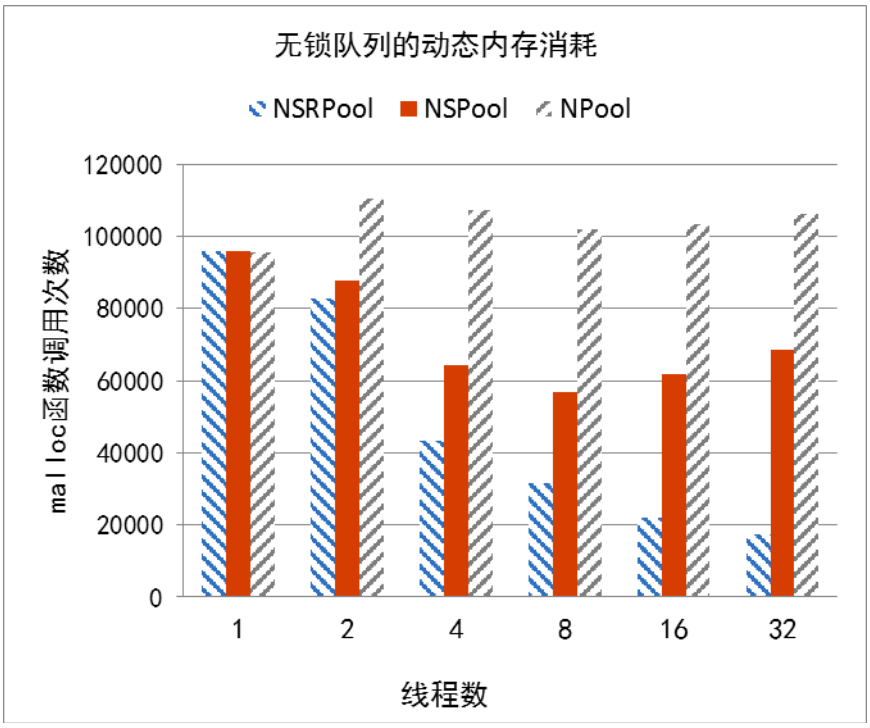

图 5.8 无锁队列的动态内存消耗情况

Fig. 5.8 Dynamic Memory Consumption of the Lock Free Queue

图 5.7 和 5.8 还显示，当使用资源窃取内存池时，线程的平均动态内存消耗量

尽管显著低于使用普通内存池时的动态内存消耗量，但随着竞争线程数的从 1 增加到 32，使用资源窃取内存池时线程的平均动态内存消耗量却呈先减（从 1 个线程增加到 8 个线程时）后增（从 8 个线程增加到 32 个线程时）。原因在于，当竞争线程数量比较大的时候仅依靠在线程本地缓冲区为空时执行窃取操作无法有效地在各个线程的私有缓冲区间达成平衡，这一点也被 Shavit 在[10]中证明。

此外，由于资源均衡内存池的使用，线程的平均动态内存消耗量随竞争线程数从 1 增加到 32 而持续降低，这与资源窃取内存池相比是很大的改进。究其原因，在于资源均衡内存池引入的额外的均衡操作更好地平衡了各个线程私有缓冲区中的自由节点数。归还操作的引入带来的效率和扩展性的提升来自两个方面：

① 尝试将待释放节点归还到其他线程的私有缓冲区中而不是在对本地缓冲区的入队操作失败后直接通过内存分配器将其释放归还给系统，系统有更多的机会重用已分配的节点，而且，

② 尝试将待释放节点归还到其他线程的私有缓冲区帮助在线程的私有缓冲区间达成更好的平衡条件，降低了线程在本地缓冲区上调用的 Dequeue 方法和在其他线程私有缓冲区上调用的 StealNode 方法的失败率，间接减少了动态内存的消耗量，这也反映在资源均衡内存池和资源窃取内存池在同样并发线程数量下相差巨大的 malloc 函数调用次数上。

5.4 操作执行时间

第二个实验测量使用内存池时无锁数据结构的操作平均执行时间。从图 5.9 和 5.10 可以观察到，除了使用单线程的情况之外，使用资源均衡内存池和使用资源窃取内存池时无锁数据结构的操作平均执行时间均比使用普通内存池时的要低。而且，除了单线程的情况之外，使用资源均衡内存池时无锁数据结构的操作平均执行时间使用资源窃取内存池时的也要低。

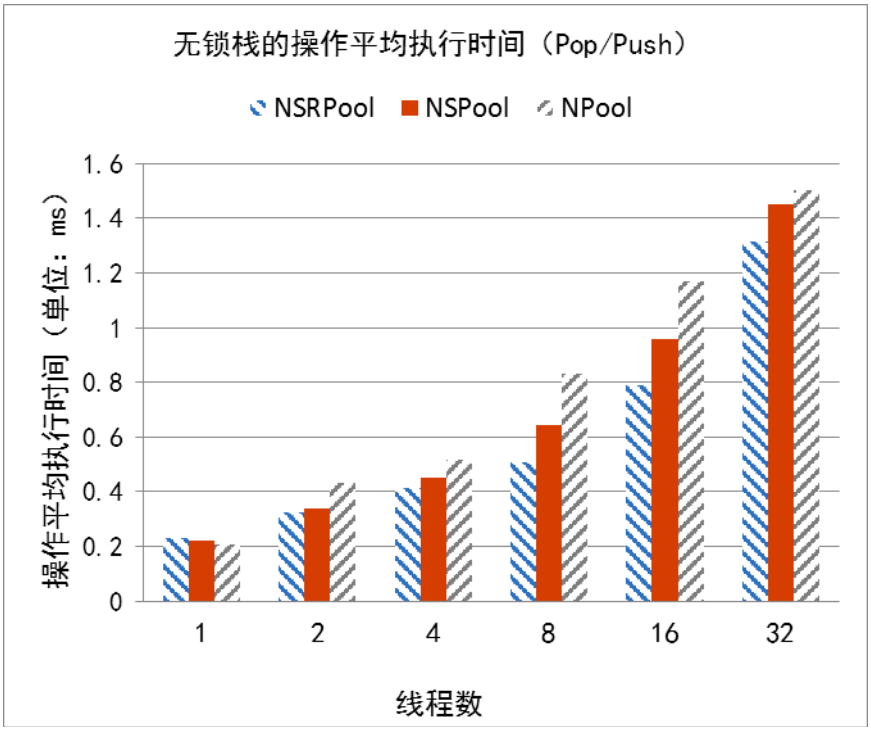

图 5.9 无锁栈的操作平均执行时间

Fig. 5.9 Average Operation Execution Time of the Lock Free Stack

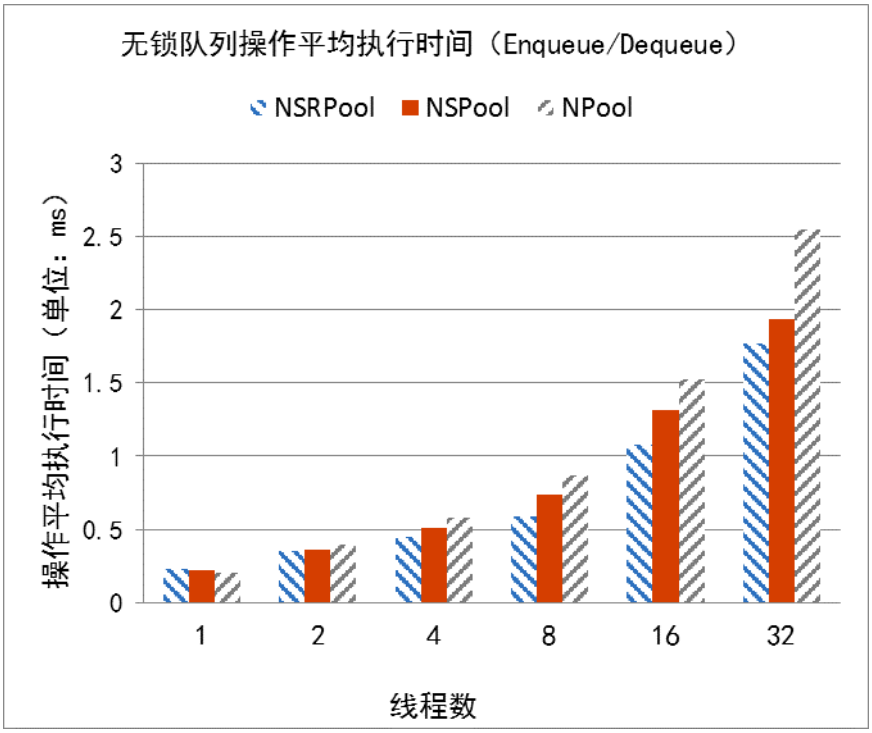

图 5.10 无锁队列的操作平均执行时间

Fig. 5.10 Average Operation Execution Time of the Lock Free Queue

和普通内存池相比，资源窃取内存池通过引入窃取操作平衡了各个线程私有

缓冲区之中的自由节点数量，减少了线程对动态内存的消耗量并消除了对一部分非必要的 malloc 函数和 free 函数的调用，提升了系统的性能，这点可以通过使用这两种内存池时数据结构的操作平均执行时间上的差值而反应出来。

通过引入归还操作，使用资源均衡内存池的无锁数据结构的操作平均执行时间和使用资源窃取内存池时的相比有了更进一步的提升。归还方法给了待释放节点额外的机会以在直接被通过内存分配器归还给系统之前尽可能地被归还到某个其他线程的私有缓冲区中，这在窃取操作的基础上促成了更好的平衡条件，提升了系统的性能和扩展性。归还操作对无锁数据结构性能和扩展性的提升体现在三个方面：

① 归还操作使得待释放节点有更多的机会被其它线程重用，减少了线程对 free 函数的调用次数，也相对减少了其它线程对 malloc 函数的调用次数，降低了无锁数据结构操作涉及的动态内存管理的时间消耗，降低了操作的平均执行时间，

② 其次，归还操作促成了更好的平衡条件，使得无锁数据结构操作需要的动态内存更多地由线程的私有缓冲区中的自由节点和内存池在各个线程的私有缓冲区之间“腾挪”解决，减少了对 malloc 函数的调用次数，降低了操作的平均执行时间，

③ 再者，归还操作促成的更好的平衡条件降低了无锁数据结构对动态内存的需求，减少了对内存分配器的访问次数和频率，这高并发情况下这大大减轻了单一内存分配器上的瓶颈效应，提升了无锁数据结构在共享内存多核处理器环境下的性能和应对高并发情况下的扩展性。

另外，从图 5.9 和 5.10 还可以看出，资源均衡内存池和资源窃取内存池对无锁数据操作执行效率的提升在竞争线程数量达到 32 时达到顶点，随后，在竞争线程超过 32 时开始提升效应下降并逐渐与使用普通内存池时持平。这可以归咎于硬件线程结构的缺少导致高并发度下，更多的并发线程导致的更长的线程切换等待时间，以及窃取和归还这两个均衡操作引起的并发开销。

5.5 平衡结果

第三个实验测量均衡操作（窃取和/或归还）的效果，在广泛的并发水平下分别考察了使用三种内存池（NPool，NSPool，NSRPool）的情况下自由节点在各个线程缓冲区中的最终分布，以验证均衡操作的效果，结果如图 5.11 和 5.12 所示。

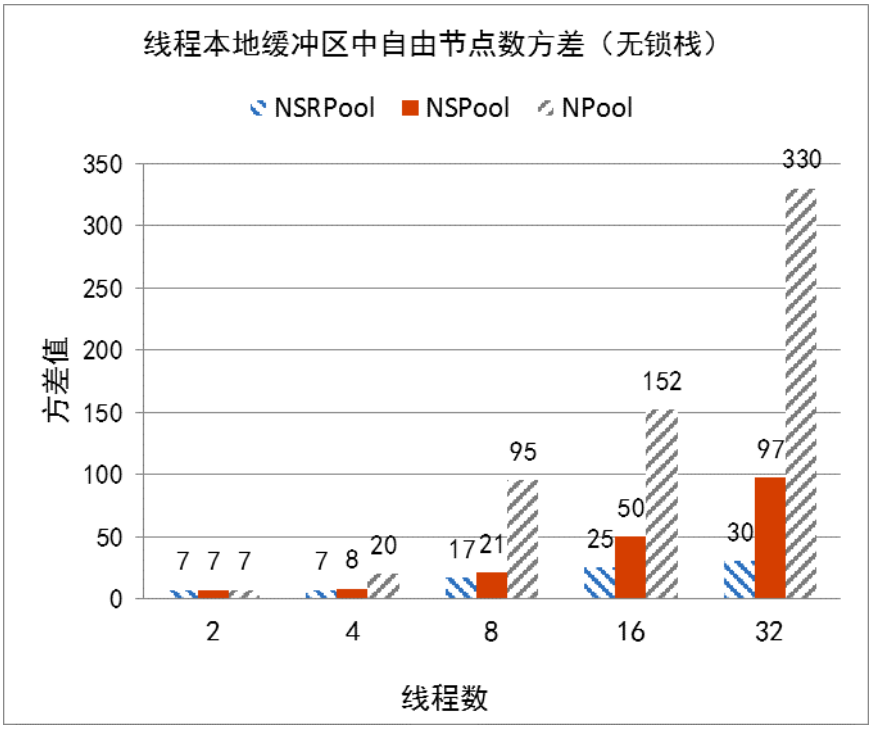

图 5.11 无锁栈的均衡情况

Fig. 5.11 Balancing Result of the Lock Free Stack

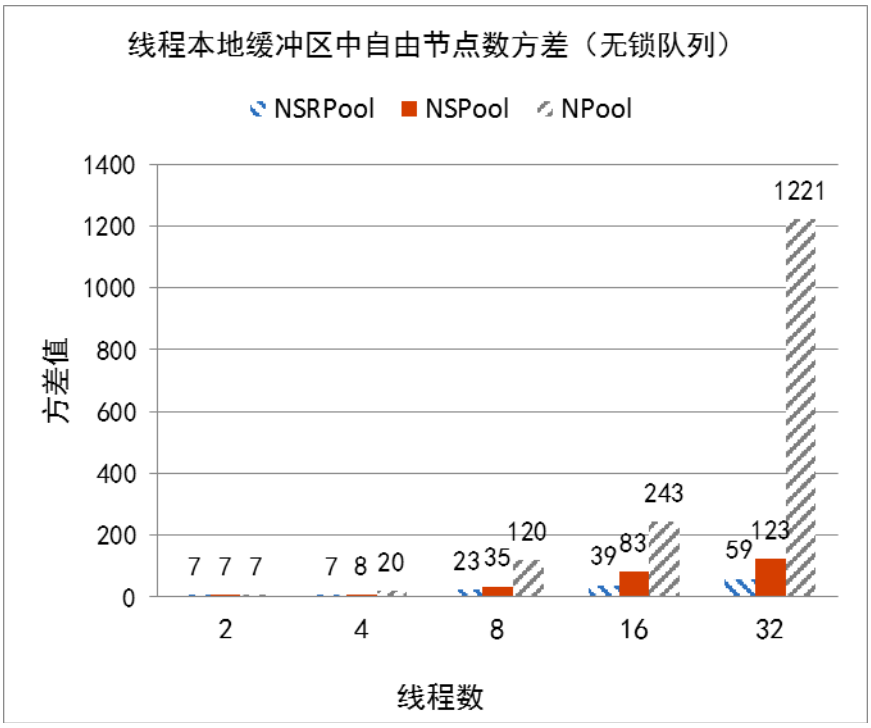

图 5.12 无锁队列的均衡情况

Fig. 5.12 Balancing Result of the Lock Free Queue

从图 5.11 和 5.12 可以看出，在各个并发水平上，使用本文提出的两种内存池

时，各线程本地缓冲区中自由节点数量的方差均低于使用 NPool 时的情况，且这个差值随着竞争线程数的增加而稳步扩大, 这要归功于均衡操作（窃取和/或归还）。

尤其值得注意的是，在使用 NPool 时，自由节点在各线程缓冲区中的分布差异较大，而竞争线程数量的增加、程序动态性的增强和操作系统的调度更是导致线程的运行轨迹各异，从而使各线程在自由节点拥有量上产生了极大的不均衡，且这种不均衡随着并发水平的升高而迅速加剧。图 5.11 和 5.12 显示，在竞争线程数从 2 增长到 64 的过程中，使用 NPool 时线程本地缓冲区中自由节点数量的方差从仅为个位数急剧增加到数千。这也印证了之前对无锁数据动态内存消耗的分析，即多线程并发访问会导致内存资源在各线程本地缓冲区中分布不均衡。

图 5.11 和 5.12 还显示，与使用 NPool 时线程本地缓冲区中自由节点数量的方差随着竞争线程数量增多而迅速增长相比，在使用 NSPool 和 NSRPool 时，方差均比较小且增长缓慢。其原因在于，均衡操作（窃取和/或归还）将自由节点在各个线程本地缓冲区之间腾挪，频繁的腾挪带来了自由节点在各线程间较为均衡的分布，在竞争线程数增加的同时，均衡操作也越来越频繁，因而方差的增长较为缓慢。特别地，使用 NSRPool 时的方差在各个并发水平下均低于使用 NSPool 时的情况，且方差在竞争线程数达到 32 后开始下降，这是因为归还操作和窃取操作同时使用时大大增强了内存池的均衡能力，更进一步地改进了自由节点在各线程本地缓冲区间分布的均匀程度。

5.6 本章小结

本章通过详尽的实验分别从目标无锁数据结构的动态内存消耗量、无锁数据结构操作的平均执行时间和线程本地缓冲区中自由节点数量的方差这三个方面对本文提出的两种内存池的性能作了详尽的考察，并与使用普通内存池时的情况作了对比。实验显示，本文提出的两种内存池在上述三个指标上的表现均显著优于普通内存池，且性能随着并发水平的提高而增长，具有较好的可扩展性。实验还表明，本文提出的均衡操作达到了设计的初衷，在各个并发水平下，均衡操作均能有效实现自由节点在各个线程本地缓冲区中的均匀分布，并能随着并发水平的升高有效控制自由节点分布差异的增长。另外，为了能让无锁数据结构安全使用动态内存并避免 ABA 问题，本文还用风险指针对两种目标数据结构（无锁队列和无锁栈）进行了修改，并给出了和风险指针集成的方法。

6 总结和展望

6.1 研究工作总结

受制于硅晶体的物理特性，延续了数十年的处理器主频竞赛告一段落，取而代之的是处理器核心数量的竞赛——将多个主频适中的核心集中到一块处理器中，通过提高处理器同一时间能够执行的指令数量来提高程序性能。这给应用程序的设计带来了挑战——如何在确保正确性的同时在设计中融入更多的并发性，实现这一点的主要手段就是并发数据结构（程序=数据结构+算法）[111]。作为并发数据结构领域的佼佼者，无锁数据结构提供了比基于锁的数据结构更高的并发度和扩展性，因而更能适应多核时代的需求。但是，由于多线程并发访问给无锁带来了更大的动态内存需求以及随之而来的更多、更频繁的动态内存管理操作，降低了无锁数据结构的性能。

本文从负载均衡的角度对无锁数据结构的动态内存管理进行了研究，将其抽象为“逆向”负载均衡问题，并应用负载均衡的思想提出了两种新型并发队列及与之分别对应的两种内存池，来改进无锁数据结构的动态内存管理。本文的主要工作如下：

① 对并发数据结构领域进行了研究和分析，总结了并发数据结构领域的各种常用同步结构和技术，指出了各种技术的原理、适用性和相对的优缺点。同时，还指出了在多核时代传统并发数据结构、同步结构和技术所面临的挑战，多核时代的新型同步结构和技术、无锁数据结构的相对性能优势以及多核时代并发数据结构的发展趋势。

② 对动态内存管理领域的常用技术和手段做了研究和总结，分析了应用程序动态内存管理原理以及几种高效并发动态内存分配器的原理、设计和实现，指出了无锁数据结构动态内存管理的特殊问题和挑战以及现有研究对该问题的忽略。在此基础上，分析了内存池在降低无锁数据结构动态内存消耗方面的作用，并提出用内存池来改进无锁数据结构的动态内存管理问题。

③ 考查了负载均衡的起源和发展，对负载均衡在计算机系统（分布式和共享内存）任务调度领域的应用作了深入的研究，分析了几种基于任务窃取、任务分担的调度方式和相应数据结构的设计与实现，提出了用负载均衡的思想给无锁数据结构的动态内存管理问题建模并用负载均衡的思想解决这一问题的动机，并将共享内存多核处理器环境下无锁数据结构的动态内存管理抽象为“逆向”负载均衡问题。

④ 分析了现有内存池在共享内存多核处理器环境下的性能缺陷，基于负载均

衡模型提出了一种并发队列并在此基础上构建了一种基于窃取技术的内存池，该内存池可以有效地将内存（自由节点）在各线程的本地缓冲区之间较为均衡地分布，在总体上降低无锁数据结构相关的动态内存消耗量的同时降低操作的执行时间。同时，鉴于前述内存池仅基于窃取技术所带来的局限性，设计了一种同时支持窃取操作和归还操作的并发队列，并将该并发队列用作线程本地缓冲区构建内存池，该内存池进一步降低了无锁数据结构相关的动态内存消耗和操作平均执行时间。

⑤ 将本文提出的两种内存池和普通内存池应用于两种常用的无锁数据结构，并对它们的各个性能指标作了对比和分析，证明了本文提出的内存池在无锁数据结构相关动态内存消耗量、操作平均执行时间和扩展性这三个指标上都显著优于普通内存池。此外，还考量了本文提出的两种内存池在将内存资源（自由节点）在各个线程本地缓冲区之间均衡分布的能力，实验显示，均衡操作完全符合设计的期望。

6.2 未来展望

无锁数据结构在共享内存多核处理器环境下的动态内存管理涉及诸多方面的问题，本文的研究工作仅仅考虑了通过均衡操作减少动态内存消耗量以实现性能提升这一个方面，还有诸多方向需要探索和研究：

① 本文提出的内存池将并发队列用作线程本地缓冲区，但队列本身的在高并发度下的扩展性不佳，由此带来的并发访问冲突会不可避免地降低均衡操作的性能。因此，需要进一步研究效能和扩展性更好的数据结构，例如与消解技术[112]或衍射技术[113]相结合。

② 本文提出的内存池在执行均衡操作时，均使用固定的失败重试次数，这使得内存池缺乏对不同环境的适应性。需要设计适配机制，使得失败重试次数可以根据内存池的运行记录动态调整。

③ 现有的内存池使用固定大小线程本地的缓冲区，无视实际情况下在系统内存大小、目标数据结构和任务类型方面的区别，需要设计适配机制使得内存池能够根据运行记录、系统环境、目标数据结构的环境相关的参数动态调整线程本地缓冲区的大小。

致　　谢

若无我的导师杨小帆教授的支持和教导，我无法想象今天论文完成的喜悦和成就感。杨教授对我研究工作的鼓励一直是我的动力来源，他于我一直持有开放的心态和极大的耐心，相信我能够在自己的研究领域里有所斩获。在这三年的学习和研究生涯中，杨教授对学术研究孜孜不倦的追求和悉心付出，特别是他对研究纯粹的兴趣和执着，不仅在学术上，更在为人上给我树立了一个值得模仿和看齐的榜样。

另一个我需要特别感谢的人是图灵奖得主 Dijkstra 教授，作为荷兰的第一个程序员，他对计算机科学和程序设计的执着一直鼓舞着我不断学习和进步，他对专业的执着和信念在大学时代数次给我以激励，他的形象有如一盏明灯，在数个迷茫的日子和灰心丧气的关头给我指明了方向。

图灵奖得主 Knuth 教授是我另外的一个动力源泉，他对程序设计充满自豪感的执着和实践精神一直鼓舞着我，令我在任何时候都敢于大声地表明，我最大的兴趣是编程。C++委员会主席 Hurb Sutter 先生的一系列富有见地的文章促使我选择并行计算作为我硕士的研究方向，布朗大学的 Maurice Herilihy 教授和特拉维夫大学的 Nir Shavit 教授在并发数据结构领域开创性的研究引导我进入并发数据结构领域并将其作为我毕业论文的题目。

我很荣幸能在重庆大学认识几个充满创造力和激情的朋友。刘少东为我的研究提供了不小的帮助，和他的交谈总是充满激情和灵感的火花。不论在学习上还是生活上，施伟总是在我需要帮助的时候伸出援手，他对学习的努力和执着也影响着我。

我的父母一直是我攻读硕士学位最大的激励和动力来源，正因为有了父母的鼓励和不断的关爱，我才能完成整个硕士阶段的学习和研究。还有我的弟弟，尽管研究方向不同，但他总是能在需要的时候给我以中肯的建议和支持。

刘恒

二 O 一三年四月　于重庆

参考文献

[1] M. Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes [C]. In Proceedings of the annual ACM Symposium on Principles of Distributed Computing, Monterey, 2002.

[2] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking Memory Management Support for Dynamic-Sized Data Structures [J]. ACM Transactions on Computer Systems, 2005, 23(2): 146-196.

[3] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines [C]. In Proceedings of the 3rd annual ACM symposium on Parallel algorithms and architectures, 1991: 237-245.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing [J]. Journal of the ACM, 1999, 46(5): 720 – 748.

[5] N. Arora, R. Blumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors [C]. In Proceedings of the 10th annual ACM symposium on parallel algorithms and architectures (SPAA), 1998.

[6] M. Herlihy. A methodology for implementing highly concurrent data structures [C]. In Proceedings of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 1990: 197–206.

[7] M. Herlihy. Wait-free synchronization [J]. ACM Transactions on Programming Languages and Systems, 1991, 11(1): 124–149.

[8] A. Reza, A. Tabatabai, C. Kozyrakis, and B. Saha. Unlocking Concurrency [J]. ACM Queue, 2007, 25: 24-33.

[9] B. Cantrill and J. Bonwick. Real World Concurrency [J]. Communications of the ACM, 2009, 51(11): 34-39.

[10] D. Handler and N. Shavit. Non-Blocking steal-half work queues [C]. In Proceedings of the 21st annual ACM symposium on pricinples of distributed computing, 2002.

[11] D. Chase and Y. Lev. Dynamic circular work-stealing deque [C]. In Proceedings of the 2005 annual ACM symposium on parallel algorithms and architectures (SPAA), 2005: 21-28.

[12] 刘恒, 杨小帆, 供动态无锁数据结构使用的资源窃取型无锁内存池[J]. 计算机应用研究, 2012, 29(10): 3772-3775.

[13] D. Knuth. The Art of Computer Programming: Fundamental Algorithms (Third Edtion) [M]. Addison-Wesley, 1997.

[14] M. Weiss. Data Structures and Algorithm Analysis (Second Edition) [M]. Addison-Wesley, 2002.

[15] R. Sedgewick and K. Wayne. Algorithms (Fourth Edition). Addison-Wesley, 2011.

[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms (Third Edtion) [M]. MIT Press, 2009.

[17] A. Levitin. Introduction to the Design and analysis of algorithms [M]. Addison-Wesley, 2006.

[18] J. Keinberg and E. Tardos. Algorithm Design [M]. Addison-Wesley, 2005.

[19] K. Fraser. Practical Lock-Freedom [D]. Ph.D. Thesis, Computer Laboratory, University of Cambridge, 2004.

[20] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In Proceedings of the 15th annual ACM symposium on Parallel algorithms and architectures, pp. 314 – 323, NY, June 2004.

[21] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit application [C]. In Proceedings of the 15th annual ACM symposium on Parallel algorithms and architectures, 2004: 31 – 39.

[22] M. Bach. The Design of the UNIX Operating System [M]. Prentice Hall, 1986.

[23] M. Maggio and D. krumme. A flexible system call interface for interprocessor communication in a distributed memory multicomputer [J]. ACM SIGOPS Operating Systems Review, 1991, 25(2): 4-21.

[24] R. Bryant and D. O'Hallaron. Computer Systems: A Programmer's Perspective [M]. Prentice Hall, 2002.

[25] W. Li, S. Mohanty, and K. Kavi. A page-based hybrid (software-hardware) dynamic memory allocator [J]. IEEE Computer Architecture Letters, 2006, 5(2).

[26] P. R. Wilson, M. S. Johnstone, Michael Neely, and David Boles, Dynamic storage allocation: A survey and critical review [C]. In 1995 International Workshop on Memory Management, 1995: 1-116,.

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

[28] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities [C]. In Proceedings of the Joint Computer Conference, 1967: 483-485,.

[29] M. Ben-Ari. Principles of Concurrent and Distributed Programming (Second Edition) [M]. Addison-Wesley, 2006.

[30] C. A. R. Hoare. Communicating Sequential Processes (CSP) [M]. Prentice Hall International,

Hemel Hempstead, 1985–2004.

[31] H. Sutter and J. Larus. Software and the concurrency revolution [J]. ACM Queue, 2005, 3 (7).

[32] E. Dijkstra. Cooperating Sequential Processes (EWD-123) [M]. The origin of concurrent programming, Springer-Verlag, 2002.

[33] A. Tanenbaum. Modern Operating Systems [M]. Prentice Hall, 2007.

[34] A. Silberschatz, P. Galvin, and G. Gagne. Operating System Concepts [M]. Wiley, 2008.

[35] L. Lamport. A new solution of Dijkstra's concurrent programming problem [J]. Communications of the ACM, 1974, 17(8): 453-455.

[36] C. A. R. Hoare. Monitors: An Operating System Structuring Concept [J]. Communications of the ACM, 1974, 17(10): 549-557.

[37] H. Boehm, A. Demers, and C. Uhler. Implementing multiple locks using Lamport's mutual exclusion algorithm [J]. ACM Letters on Programming Languages and Systems, 1993, 2(1-4): 46-58.

[38] D. Gilbert. Modeling spin locks with queuing networks [J]. ACM SIGOPS Operating Systems Review, 1978, 12(1): 29-42.

[39] T. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors [J]. IEEE Transactions on Parallel and Distributed Systems, 1990, 1(1): 6-16.

[40] A. Downey. The Little Book of Semaphores (Second Edition) [D]. http://www.greenteapress.com/semaphores/index.html, 2005.

[41] K. Apt. Edsger Wybe Dijkstra (1930–2002): A Portrait of a Genius [J]. Formal Aspects of Computing, 2002, 14: 92-98.

[42] L. Bic and A. Shaw. Operating Systems Principles [J]. Prentice Hall, 2003.

[43] E. Dijkstra. The mathematics behind the Banker's Algorithms (EWD-623) [M]. Selected Writings on Computing: A Personal Perspective, 1982: 308-312.

[44] L. Lamport. The mutual exclusion problem: Part I—a theory of interprocess communication [J]. Journal of the ACM, 1986, 33(2): 313–326,.

[45] L. Lamport. The mutual exclusion problem: Part II—statement and solutions [J]. Journal of the ACM, 1986, 33(2): 327–348.

[46] L. Lamport. A fast mutual exclusion algorithm [J]. ACM Transactions on Computer Systems, 1987, 5(1): 1–11.

[47] C. Shub. Preemption costs in round robin scheduling [C]. In Proceedings of the 1978 annual ACM conference, 1978, 2: 868-874.

[48] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin [C]. In Proceedings of the 14th ACM SIGPLAN symposium

on principles and practice of parallel programming, 2009: 65-74.

[49] M. Russinovich and D. Solomon. Microsoft Windows Internals [M]. Microsoft Press, 2004.

[50] B. Hill, J. Bacon, C. Burger, J. Jesse, and I. Krstic. The Official Ubuntu Book [M]. Prentice Hall, 2006.

[51] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. Soft Real-Time Systems: Predicatability vs. Efficiency [M]. Spriner-verlag, 2005.

[52] P. Laplante. Real-Time Systems Design and Analysis [M]. John Wiley & Sons, 2004.

[53] D. Bovet and M. Cesati. Understanding the Linux Kernel (Third Edition) [M]. O'Reilly Media, 2005.

[54] J. Richter and C. Nasarre. Windows via C/C++ [M]. Microsoft Press, 2007.

[55] L. Suh-Yin and L. Ruey-Long. A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems [J]. IEEE Transactions on Knowledge and Data Engineering, 1996, 8(1): 144-156.

[56] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combing and the synchronization-parallelism tradeoff [C]. In Proceedings of the 22nd annual ACM symposium on parallelism in algorithms and architectures, 2010: 355-364.

[57] A. Roy, K. Fraser, and S. Hand. A transactional approach to lock scalability [C]. In Proceedings of the 20th annual ACM symposium on parallelism in alogrithms and architectures, 2008: 101-103.

[58] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects [J]. ACM Transaction on Programming Languages and Systems, 1990, 12(3): 463-492.

[59] L. Lamport. Specifying concurrent program modules [J]. ACM Transactions on Programming Languages and Systems, 1983, 5(2): 190–222.

[60] C. Papadimitriou. The serializability of concurrent database updates [J]. Journal of the ACM (JACM), 1979, 26(4): 631–653.

[61] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming [M]. Morgan Kaufmann, 2008.

[62] M. Herlihy and N. Shavit. On the Nature of Progress [C]. In Proceedings of the 15th international conference on Principles of Distributed Systems, 2011: 313-328.

[63] K. Fraser and T. Harris. Concurrent Programming without Locks [J]. ACM Transactions on Computer Systems, 2007, 25(2).

[64] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures [C]. In Proceedings of the Twentieth Annual International Symposium on Computer Architecture, 1993: 289–300.

[65] N. Shavit and D. Touitou. Software transactional memory [J]. DistributedComputing, 1997, Special Issue (10): 99–116.

[66] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures [C]. In Proceedings of the 22nd annual ACM Symposium on Principles of Distributed Computing, 2003: 92–101.

[67] D. Dice, O. Shalev, and N. Shavit. Transactional locking II [C]. In Proceedings of The 12th International Symposium on Distributed Computing (DISC2006), 2006: 194–208.

[68] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions [C]. In Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2005: 48–60.

[69] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques [C]. In Proceedings of the 16th International Symposium on Computer Architecture, 1989: 396–406.

[70] T. E. Anderson. The performance of spin lock alternatives for sharedmoney multiprocessors [J]. IEEE Transactions on Parallel and Distributed Systems, 1990, 1(1): 6–16.

[71] G. Granunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors [J]. IEEE Computer, 1990, 23(6): 60–70.

[72] J. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors [J]. ACM Transactions on Computer Systems, 1991, 9(1): 21–65.

[73] T. Craig. Building FIFO and priority-queueing spin locks from atomic swap [D]. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, 1993.

[74] P.Magnussen, A. Landin, and E.Hagersten. Queue locks on cache coherent multiprocessors [C]. In Proceedings of the Eighth International Symposium on Parallel Processing (IPPS), IEEE Computer Society, 1994: 165–171.

[75] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout [J]. ACM SIGPLAN Notices, 2001, 36(7): 44–52.

[76] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks [C]. In Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, 2002: 31–40.

[77] M. Moir, V. Marathe, and N. Shavit. Composite abortable locks [C]. In Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006: 1-10.

[78] P. Courtois, F. Heymans, and D. Parnas. Concurrent Control with Readers and Writers [J]. Communications of the ACM, 1971, 14(10): 667-668.

[79] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks [C]. In Proceedings of the 21st annual ACM symposium on parallelism in algorithms and architectures, 2009:

101-110.

[80] J. Valois. Lock-free linked lists using compare-and-swap [C]. In Proceedings Of the 14th annual ACM symposium on Principles of Distributed Computing, 1995: 214–222.

[81] T. Harris. A pragmatic implementation of non-blocking linked-lists [C]. In Proceedings of the 15th International Conference on Distributed Computing, 2001: 300–314.

[82] M. Michael. High performance dynamic lock-free hash tables and list-based sets [C]. In Proceedings of the 14th annual ACM symposium on Parallel Algorithms and Architectures, 2002: 73–82.

[83] M. M. Michael, Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects [J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(6): 491-504.

[84] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures [C]. In Proceedings of the 16th International Symposium on Distributed Computing, Springer-Verlag Heidelberg, 2002, 2508: 339–353,.

[85] R. Treiber. Systems programming: Coping with parallelism [D]. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[86] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared - memory multiprocessors [J]. Journal of Parallel and Distributed Computing (JPDC), 1998, 51(1): 1–26.

[87] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks [J]. Theory of Computing Systems, 1997, 30: 645–670.

[88] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm [C]. In Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures, 2004: 206–215.

[89] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared - memory multiprocessors [J]. Journal of Parallel and Distributed Computing, 1998, 51(1): 1–26.

[90] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms [C]. In Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing, 1996: 267–275.

[91] O. Shalev and N. Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables [J]. Journal of the ACM, 2006, 53(3): 379-405.

[92] W. Weihl. Local atomicity properties: modular concurrency control for abstract data types [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1989, 11(2): 249–282.

[93] W. Li, S. Mohanty, and K. Kavi. A page-based hybrid (software-hardware) dynamic memory allocator [J]. IEEE Computer Architecture Letters, 2006, 5(2).

[94] P. Wilson, M. Johnstone, M. Neely, and David Boles. Dynamic storage allocation: A survey and critical review [C]. The 1995 International Workshop on Memory Management, 1995: 1-116.

[95] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs [J]. ACM SIGPLAN Notice, 1992, 27(12): 71-80.

[96] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations [C]. In Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), 1995: 179–188.

[97] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality [C]. In Proceedings of the 6th International Conference on Supercomputing, Distributed Computing, 1992 : 323–334.

[98] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches [J]. IEEE Transactions on Computers, 1994, 43(6): 651–663.

[99] Doug Lea. A Memory Allocator [D].http://gee.cs.oswego.edu/dl/html/malloc.html.

[100] W. Gloger. Dynamic Memory Allocator Implementations in Linux System Libraries [D]. http://www.dent.med.uni-muenchen.de/~wmglo/.

[101] E. Berger. Memory Management for High-Performance Applications [D]. PhD thesis, University of Texas at Austin, August 2002.

[102] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications [C]. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000: 117–128.

[103] M. Michael. Scalable lock-free dynamic memory allocation [C]. In Proceedings of the 26th ACM SIGPLAN conference on Programming language design and implementation, 2004: 35 – 46.

[104] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to implement scalable and lock-free fifo queues [C]. In Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures, 2005: 253-262.

[105] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm [C]. In Proceedings of the 9th International Conference on Principles of Distributed Systems, 2005: 3-16.

[106] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists [C]. In Proceedings of the 23rd annual ACM symposium on principles of distributed computing, 2004: 50-59.

[107] F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors [C]. In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, 1981: 187–194.

[108] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments [C]. In Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 1998.

[109] D. Hendler, Y. Lev, and N. Shavit. A dynamic-sized nonblocking work stealing deque [C]. In Proceedings of the 18th International Symposium on Distributed Computing, Springer-Verlag Heidelberg, 2004, 3274: 188–200.

[110] C. A. R. Hoare. An Axiomatic Basis for Computer Programming [J]. Communications of the ACM, 1969, 12(10): 576-583.

[111] A. Stepanov and P. McJones. Elements of Programming [M]. Addison-Wesley, 2009.

[112] N. Shavit. Data structures in the multicore age [J]. Communications of the ACM, 2011, 54(3): 76-84.

[113] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A Provably Correct Scalable Concurrent Skip List [C]. In Proceedings of the 10th International Conference on Principles of Distributed Systems, 2006.

附　录

A 作者在攻读学位期间发表的论文

[1] 刘恒，杨小帆，供动态无锁数据结构使用的资源窃取型无锁内存池[J]. 计算机应用研究，2012，29(10): 3772-3775.