·体系结构与软件技术· 文章编号:1000-3428(2017)01-0086-07 文献标志码:A 中图分类号:TP302

基于跳表与布隆过滤器的混合关键任务调度方法

黄姝娟,容晓峰,肖 锋,茹 媛

(西安工业大学 计算机科学与工程学院,西安 710021)

摘 要: 传统实时任务对共享数据的访问通常采用锁机制,该机制可能会引起死锁、优先级翻转以及 CPU 饥饿的现象。如果应用在混合关键系统中,可能会导致关键级别翻转。针对上述问题,提出一种跳表与布隆过滤器相结合的同步方法。该方法将混合关键任务的优先级调度队列采用跳表数据结构存储,实现该数据结构的无锁算法,并通过基于锁机制的布隆过滤器判断其是否已被调度执行。实验结果表明,与传统的基于锁机制的位图、堆结构以及 ELB-trees 的同步机制方法相比,该方法能减少死锁现象的发生和降低优先级翻转的几率,并且在关键级别翻转时,提升多核运行的效率。

关键词: 多核;实时调度;周期;同步机制;数据结构

中文引用格式: 黄姝娟,容晓峰,肖 锋,等. 基于跳表与布隆过滤器的混合关键任务调度方法[J]. 计算机工程,2017,43(1):86-92.

英文引用格式: Huang Shujuan,Rong Xiaofeng,Xiao Feng,et al. Hybrid Critical Task Scheduling Method Based on Skip List and Bloom Filter[J]. Computer Engineering,2017,43(1):86-92.

Hybrid Critical Task Scheduling Method Based on Skip List and Bloom Filter

HUANG Shujuan,RONG Xiaofeng,XIAO Feng,RU Yuan

(School of Computer Science and Engineering,Xi'an Technological University,Xi'an 710021,China)

【Abstract】 Traditional real-time tasks usually access the shared data using the lock mechanism. This lock synchronization mechanism may cause some phenomena such as deadlock, priority inversion and CPU starvation. It may also cause criticality inversion when used in the mixed-criticality system. So this paper proposes a new synchronization method that combines skip-list and Bloom filter for the mixed-criticality system. The skip-list is used as a shared priority queue and implemented with the lock-free algorithm and the Bloom filter based on the lock mechanism is used for querying if the task is scheduled. Experimental results show that the method has greatly reduced the phenomena of deadlock and priority inversion and increased the multi-core efficiency after the criticality inversed compared with the traditional lock mechanism and ELB-trees.

【Key words】 multi-core;real-time scheduling;cycle;synchronization mechanism;data structure

DOI: 10.3969/j. issn. 1000-3428. 2017. 01. 015

0 概述

在嵌入式多核平台下,不同处理器核对共享优先级队列的同步访问是其中一个重要环节。其实现同步的算法大多基于加锁机制。有些是在顺序操作的顶部加锁,即采用粗粒度锁机制;有些则是使用多个锁分别保护一小部分共享资源的算法,即采用细粒度锁机制。粗粒度锁机制访问共享资源,不但使得并行性效率不高,而且还要求 put 和 get 操作必须按照顺序执行,这样才能保证一致性[1]。基于细粒度锁机制的共享队列如文献[2]的跳跃表(Skip-list),要求为每个指针加锁,虽然扩展了并发优先级队列的功能,但锁的数目比结点数目还多,随着结点数量增多,空间需求量会增大,且优先级反转现象也难以避免。为了提高多核并行的效率,文献[3]提出了一个基于 Work-Stealing 的数据收集器来实现数据并行操作,由于需要细粒度锁机制来保证 Work-Stealing 的有效执行,当收集器数量增加到一定程度

基金项目: 国家自然科学基金面上项目(61572392);陕西省工业科技攻关项目(2015GY031);陕西省教育厅基金(15JK1342);西安工业大学校长基金(XAGDXJJ14015)。

作者简介: 黄姝娟(1975—),女,讲师、博士,主研方向为嵌入式与分布式计算;容晓峰,教授、博士;肖 锋,副教授、博士;茹 媛,讲师。

收稿日期: 2016-02-24 **修回日期:** 2016-04-15 **E-mail:** 349242386@qq.com

时,易发生死锁现象。

为了解决互斥锁引起的死锁、优先级反转和CPU饥饿现象,研究者建议为共享数据结构使用非阻塞算法,典型的非阻塞算法是Lock-free[4-6]。如文献[4]针对共享哈希表提出的快速无等待哈希表算法。文献[5]提出的适用于并行安全网关流水线模型中共享数据缓冲区操作的无锁队列算法。文献[6]针对共享内存提出的基于硬件CAS原语的无锁同步算法。文献[7]提出一种多核架构下Linux网络报文缓冲区重用的无锁算法。文献[8]提出了一种无锁同步的并行介度中心算法。文献[9]针对边界网关协议中共享先进先出(First In First Out, FIFO)队列实现的无锁并发操作。文献[10]提出一种Lock-free机制实现的高效无锁B树(Efficient and Lock-free B-tree,ELB-trees)。因使用较少的原子操作,从而减少较高的执行开销。上述这些共享数据结构虽然都进行了无锁算法实现,但由于混合关键任务要求存储具有不同关键级别下多种类型的数据值,而这些共享数据结构都不能明确显示此类特征,直接应用在混合关键任务的调度过程中易导致关键级别反转和优先级反转现象。而Skip-list数据结构,能够体现出混合关键任务的特征。文献[11]针对Skip-list数据结构提出一种基于多维度列表的完全一致性的Lock-free算法,保障了最坏查询时间$O(\mathrm{lb}N)$,但应用在混合关键系统中会出现较高的放弃率和大量的事务重启。文献[12]提出了一种将Lock-free机制和锁机制结合起来,实现了一个较为可行的并行FIFO队列算法,是当前一种比较好的解决方案,但不能满足混合关键任务多种优先级变化的需要。文献[13]则详细地将锁机制以及无锁机制进行了分析,说出了各自的优缺点。

本文在文献[11]的基础上,将基于Lock-free算法的Skip-list的数据结构应用到混合关键任务调度过程中,并采用基于锁机制的布隆过滤器(Bloom Filter)来实现任务状态的查找,并和只用锁机制或者无锁机制实现的并发优先级队列算法进行对比。

1　相关内容

1.1　混合关键系统

随着多核处理能力的提高,设计者将更多的安全关键任务集成在同一个平台上,这种系统称之为混合关键系统。在该系统中的安全关键任务称之为混合关键任务。这种系统在不同时刻,任务将处于不同的安全关键级别。要想让各项任务都能够根据自己的安全关键级别合理、顺利地得到执行,就必须对各安全关键级别下的所有任务进行统一的调度。研究混合关键任务的调度问题就成为当前嵌入式多核平台的焦点问题[8-9]。

混合关键系统的Task模型如下:在具有k个级别的混和关键任务系统中,每一个任务可以用一个3维向量来表示:$\boldsymbol{T}_i=(c_i,p_i,x_i)$。其中,$\boldsymbol{c}_i$是一个$k$维向量,用来分别表示该任务处于各个级别下所对应的最坏执行时间,即$\boldsymbol{c}_i=\{c_i(1),c_i(2),\cdots,c_i(k)\}$;$p_i$则表示该任务的执行周期,也是连续2次执行工作(job)被发布的最短时间间隔;$x_i(x_i\in\{1,2,\cdots,k\})$表示该任务所处的关键级别,在某一确定的时刻,它的值也是确定的。

在混和关键系统的任务模型中,每一个任务$\boldsymbol{T}_i$可以隐含无数个job,第j个job用$\boldsymbol{J}_{i,j}=(\gamma_{i,j},d_{i,j},x_{i,j},\boldsymbol{c}_{i,j})$来表示,其中,$\gamma_{i,j}$是发布时刻,任务$\boldsymbol{T}_i$的job每隔$p_i$个时间间隔发布并执行一次,每次job发布之后,它的发布时刻$\gamma_{i,j}$就确定了;$d_{i,j}$是时限,规定为该job发布之时经过p_i之后的时刻,即$d_{i,j}=\gamma_{i,j}+p_i$;$x_{i,j}$为关键级别,job的关键级别与任务的关键级别相同,即$x_{i,j}=x_i$;同样,job的$\boldsymbol{c}_{i,j}$与任务的$\boldsymbol{c}_i$保持一致。一般情况下,根据任务的3维向量来确定所有已发布job的4维向量,进而全面确定这些job的参数,再根据相应的调度算法对这些job进行调度。

1.2　Skip-list数据结构

Skip-list(跳表)是基于分层的、替代平衡树的一种数据结构。和平衡树不同的是,跳表对于平衡的实现具有随机化特征,即跳表的插入和删除操作没有平衡树要求那么严格,是实现较为简单的一种数据结构,如图1所示。图1(a)表示一个2层跳表,第2层的链表是每隔1个节点增加一个指向它之后距离为2的指针,当在该链表中查找某个结点时,只需搜索不超过$\lceil n/2\rceil+1$个结点(n为列表中元素的个数)就可以找到该结点。图1(b)表示一个3层跳表,第3层链表是每隔3个结点就增加一个指向它之后距离为4的结点的指针,在该链表查找一个结点时,只要搜索不超过$\lceil n/4\rceil+2$个结点。如此类推,如果$i(i\geqslant 2)$层跳表的第2^{i-1}个结点都有一个指向它之后距离为2^{i-1}的结点的指针,那么只要搜索不超过$\lceil n/2^{i-1}\rceil+2^{i-2}$个结点即可,而此时每层只使指针的数目增加了$\lceil n/2^{i-1}\rceil$个。这样的数据结构搜索速度是很快的。

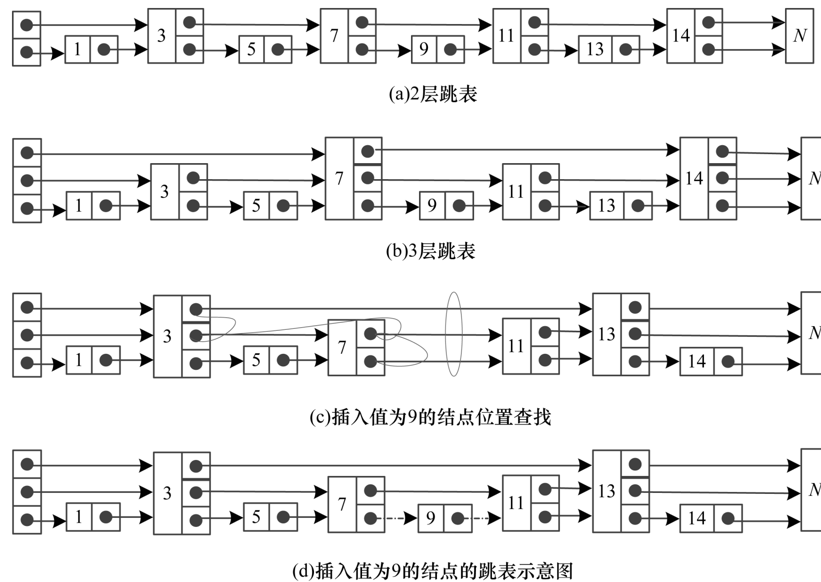

图 1 Skip-list 数据结构

要实现在跳跃表中进行插入和删除操作,需要先搜索要插入的位置。搜索的方法是从首结点的最高级别开始搜索,如果此级别的下一个结点的值小于插入的值,那么继续在此级别上遍历下一个结点;否则降低一个级别再搜索,一直到最低级别。例如在图 1(c)的结构中,要插入值为 9 的结点,那么搜索的路径如图中曲线所示。搜索每一个级别时,要记录下该级别最后读取的结点,在之后的插入操作中使其相应级别的后继指针指向新结点。插入后的列表如图 1(d)所示,其中虚线部分是被改变的指针。

1.3 Bloom Filter 数据结构

Bloom Filter 是 Bloom 在 1970 年提出的一种利用 Hash 函数对时间和空间进行平衡折中的数据管理方法[14]。该方法曾经在文献[15]中被用来判断状态爆炸间的冲突状态,本文则将其用于来判断任务是否被执行,以便协调处理器核资源的使用情况。

Bloom Filter 是一种空间效率很高的数据结构,它利用位数组来判断一个元素是否属于该集合。它的工作原理是使用 k 个相互独立的哈希函数,将集合中的每个元素映射到 $\{1,2,\cdots,m\}$ 的范围中。对任意一个元素 x,第 i 个哈希函数映射的位置 $H_i(x)$ 就会被置为 1。如图 2 中,$k=3$ 时 X_1 元素的 3 个哈希值都为 1,说明 X_1 是集合 S 的元素;X_2 的 3 个哈希值都为 1,说明 X_2 也是集合 S 的元素。

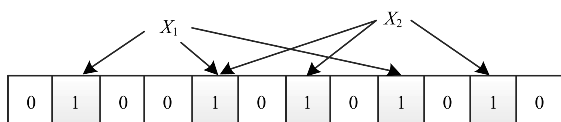

图 2 2 个元素的哈希值选中同一位置示意图

Bloom Filter 的这种高效性是有一定代价的,因为存在多个元素的哈希值选中同一位置的情况,在判断一个元素是否属于某个集合时,有可能会把不属于这个集合的元素误认为属于这个集合(False Positive),这就是所谓的冲突(Collision)。当冲突发生时,有多种策略可供选择,比如用链表将映射到同一位置的元素串起来,或者在冲突发生时再进行哈希映射,直到找到空位为止等。

2 混合关键系统调度过程中的同步机制

2.1 COSBF 设计方案

本文采用 2 种机制来保障安全关键任务的执行以及负载均衡。一种是利用 Bloom Filter 来判断被选择的关键任务的 job 运行状态;另一种是基于 Lock-free 算法实现 Skip-list 数据结构,用以存放将要被调度任务的 job。系统设计方案如图 3 所示。对于一个具有 k 个关键级别的 n 个混合关键任务的系统,采用全局调度算法将关键任务调度到 m 个核上执行。在系统整体调度过程中,首先,利用 Skip-list 数据结构的创建、插入等操作,将需要调度的 n 个混合关键任务,按照不同关键级别,插入到 Skip-list 数据结构中。其次,空闲核从 Skip-list 队列中选择当前关键级别中优先级最高任务的 job 或者是被挂起的 job 准备执行。先判断是否已经发生时限丢失,如果是那么删除该 job,选择下一个 job 执行。否则,通过 Bloom Filter 来查询将要被调度 job 所处的状态,如果是未执行状态,那么取出执行。否则从 Skip-list 中选择当前关键级别中下一个任务对应的 job 准备执行。在运行过程中,如果有高关键级别任务(高优先级任务)被发布,则抢占正在处理低关键级别(低优先级任务)的处理器核,将低关键级别的 job 插入到 Skip-list 链表中,并将 Bloom Filter 中相应的 job 状态设定为未执行状态。最后,不同的核可以同时访问 Skip-list 数据结构。

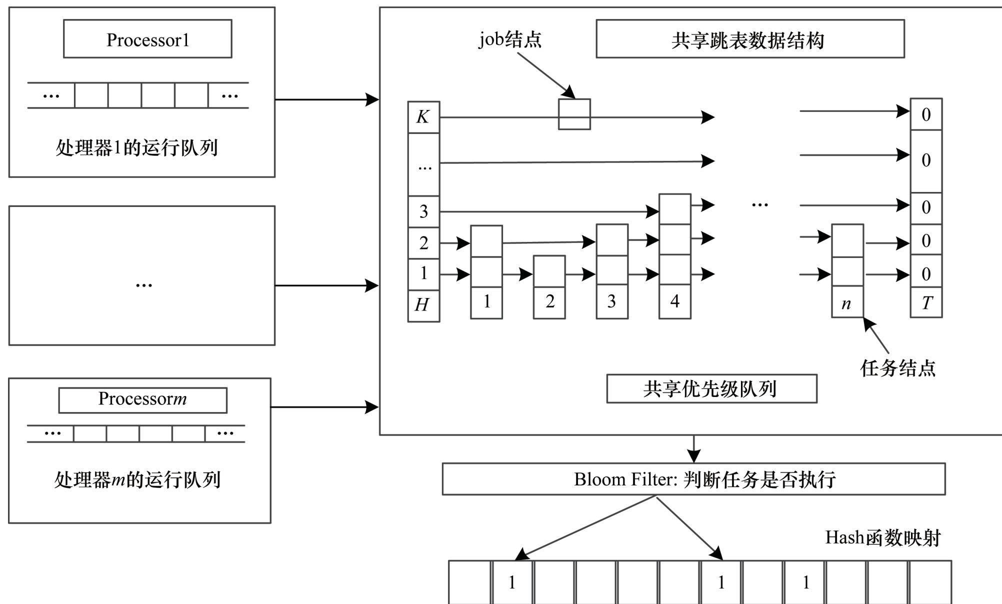

图 3　基于 COSBF 的共享优先级队列设计方案

2.2　COSBF 同步算法实现

在基于全局优先级调度算法的基础上,对优先级队列 Skip-list 采用 Lock-free 访问方法以及 Bloom Filter 状态查询法,而在 Bloom Filter 中最为关键的技术是位数选择和哈希函数的设置。图 4 显示了整体的调度流程。

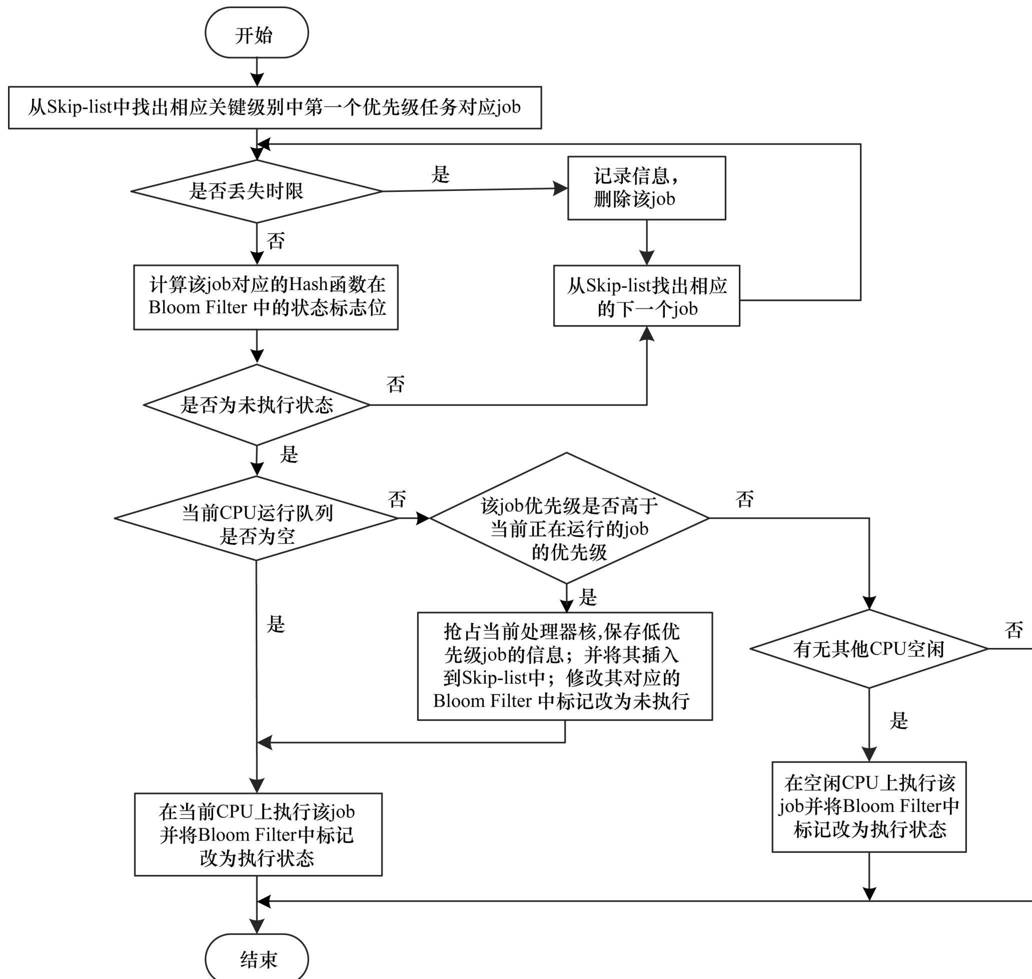

图 4　COSBF 算法执行的总体流程

本文中 Bloom Filter 位数的选择以及哈希函数的选择，都和混合关键系统中任务数量以及处理器数量有关。例如，一个混合关键系统有 100 个混合关键任务，需要分配到 4 个处理器核上，如果平均分配则每个核上将会分配 25 个任务，那么每个 job 则需要⌈lb25⌉=5 个位来表示，也就是说需要 5 个 Hash 函数值来判断一个 job 的执行状态，那么 Bloom Filter 的位数一共可选择为 4×5=20 位。

当多个处理器核同时访问 Skip-list 时，只有一个核会获得哈希函数对应的 Bloom Filter 状态位的修改权。当其他处理器核获知该 job 已经执行时，则会选择其他 job 执行。由于对 Skip-list 的插入、删除以及修改操作都依据 Lock-free 同步机制，因此可以提高多核对共享优先级队列的并行操作效率。

当开始创建 Skip-list 优先级队列时，需要更新当前任务的发布时间，创建完成之后，即可从 Skip-list 中提取任务，算法如下：

1）选取 Skip-list 中已发布的 job 或阻塞 job。

2）通过 Bloom Filter 判断该 job 是否正在执行，如果正在执行，那么从 Skip-list 任务链表中重新选取 job 结点。

3）如果该 job 未执行，那么继续判断该 job 是否超过时限时间，如果超过时限时间，那么输出该 job 丢失的时限时间，将该 job 删除，更新 job 的发布时间，将 job 重新发布。返回第 1）步在 Skip-list 任务链表中重新选取 job 结点。

4）如果该 job 未超过时限，则调度该 job，并将 Skip-list 链表中删除该 job。

在整个调度过程中，对 Bloom Filter 进行操作主要有如下方面：

1）Bloom Filter 根据任务数量和处理器数量确定位数以及哈希函数。如果当查询状态是读的过程，并行读内存是不需要加锁机制。

2）当可以调度该 job 时，根据 Hash 函数值的计算，找到 Bloom Filter 上相应的位置，如果是 0，那么需要对该位置进行加锁并直接写入 1，然后解锁。如果该位置已经是 1，那么表明是重复使用的位置，需要对重复计数器加锁并使重复计数器值增加 1，然后解锁。

3）当该 job 完成时，根据 Hash 函数值的计算，在 Bloom Filter 上将其标识为未运行值 0。同样，如果该位置为非重复使用位置（即重复使用值为 0），那么对该位置进行加锁。然后修改值为 0 后解锁。如果是重复使用值不为 0，那么需要对重复计数器加锁并使重复计数器值减少 1，然后解锁。

Bloom Filter 细粒度加锁过程的流程如图 5 所示。

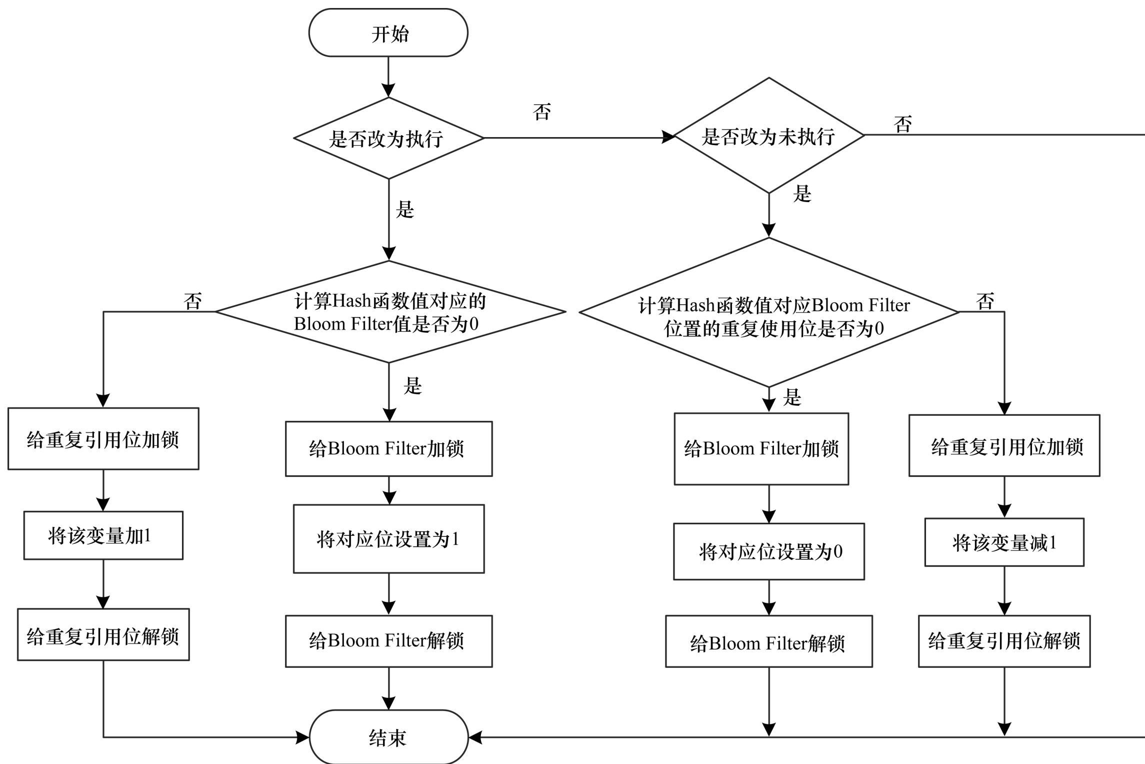

图 5 细粒度 Bloom Filter 加锁过程

3　仿真实验

为了比较上述调度同步操作的性能,首先,将本文所提出的调度方法先和 Linux 系统中基于传统锁机制的实时调度的位图结构和堆结构进行仿真实验比较,位图的纵向层次表示不同关键级别,横向表示不同优先级别以适应混合关键任务的调度。堆的层次表示不同关键级别层次,优先级顺序从左向右依次为从高到低的顺序。

位图和堆都采用图 1 所示的全局调度管理队列的模式并采用锁机制进行同步。其次,与无锁的 ELB-trees 在安全关键级别反转方面进行比较。由于无锁的 ELB-trees 只能有一个关键值,因此在设计时只能按照优先级的关键值来设定。

3.1　仿真实验平台

测试的环境是在一个 Intel (R) Core (TM) 2 Quad Q8400 多核平台上。该平台是一个 32 位的机器包含 4 核芯片运行在 2.66 GHz。每个核支持4 个硬件线程,总共 16 个逻辑核。每个核有芯片 Cache 为 128 KB,8-way 的 L1 级 Cache,以及一个共享、统一的 4 MB 16-way set associative L2 级 Cache。测试系统被配置有 4 GB 芯片外主存。大部分相关系统开销(如调度开销、上下文开销等)都忽略不计。

3.2　任务集产生方法

随机产生 100 个任务集,每个任务集中随机产生 50 个任务,执行 $2 \leqslant k \leqslant 4$ 级关键级别的任务系统,每个混合关键任务都是暗含死线的任务,即 $p_i = d_i$。任务利用率按照级别分别被随机统一分布在 $(0,1/k)$, $(1/k,2/k)$, $(2/k,3/k)$, $(3/k,4/k)$ 之间并且周期被选择从[10 ms,100 ms],任务的发布时间统一为 0 时刻。任务每个级别的最坏执行时间都基于利用率和周期计算即从方程 $c_i(k) = u_i^k \times p_i$ 中获得,其中,u_i^k 为 k 级别的利用率;p_i 为周期。周期被定义为整型,执行代价可以是非整型。这些任务集被动态地添加到 Skip-list 数据结构或者位图的数据结构中,并分配到 $m \in \{2,4,8,16,32\}$ 个核上。

3.3　结果分析

在整个仿真实验过程中,为了描述算法之间的性能差异,采用多次模拟求平均值的方法按照在某段时间内的吞吐率、核总利用率以及平均响应时间 3 个方面进行性能分析,如图 6 和表 1 所示。

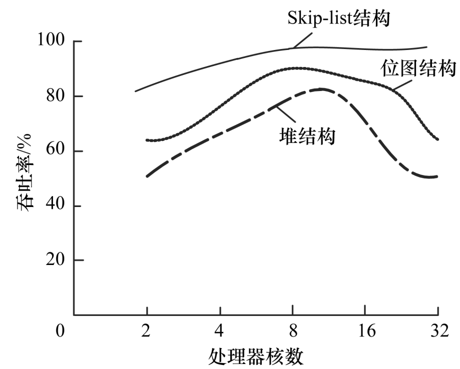

图 6　3 种情况下的吞吐率

表 1　不同核数下 3 种机制的核总利用率和响应时间

核数	Skip-list 数据结构		位图结构		堆结构	
	核总利用率	平均响应时间/ms	核总利用率	平均响应时间/ms	核总利用率	平均响应时间/ms
2	0.997	1.300 0	0.981	1.71	0.952	1.96
4	0.976	1.021 0	0.950	1.50	0.906	1.82
8	0.943	1.004 3	0.893	1.63	0.867	1.89
16	0.910	1.500 0	0.845	1.96	0.792	2.18
32	0.899	1.540 0	0.784	2.03	0.724	2.30

从图 6 可以看出,3 种共享数据结构刚开始随着核数的增加吞吐率都在上升,由于核数增加同时会增加对锁机制的竞争,因此位图结构和堆结构随着核数增加成功调度 job 的数量呈现下降趋势。且堆结构的数据调整过程比位图慢,加锁时间长,吞吐率下降比较厉害。但由于 Skip-list 是无锁访问,其锁竞争主要是 Bloom Filter 的状态修改问题,因此下降趋势趋于平缓。但总的来说随着锁机制资源竞争导致的串行效果,核数的增加并不能使得吞吐率呈现一直增长的趋势。

从表 1 可以看出,3 种方法核总利用率都会随着核数的增加而减少,原因是任务数量一定,核数增加到一定的情况下会产生剩余资源,核利用不是很充分。而平均响应时间则是随着核数增加先减少而后增加。原因是随着核数的增加任务被调度的速度加快,平均响应时间会减少,而又由于核之间的锁机制的同步竞争会使进程产生阻塞现象,使得就绪任务不能及时得到调度,因此平均响应时间又会增加。但 Skip-list 的核总利用率和平均响应时间表现是最好的,且随着核数增加而增加,减少幅度比其他两者要慢。

图 7 展示了采用 Skip-list 数据结构与无锁的 ELB-trees 数据结构在关键级别变化前后的系统吞吐率效果。可以看出,在关键级别转换前,两者的

系统吞吐率相差不是很大,但在关键级别反转之后,由于 ELB-trees 的关键值只有一个,就是优先级,而当需要查询关键级别较高的任务时就要重新调整树中结点之间的关系,因此失去了原有的 B⁺trees 的优势,系统吞吐率也就随之降低。而 Skip-list 数据结构本身就考虑了双重关键值,因此,不存在重新调整结点之间关系的问题,系统吞吐率就比较稳定。

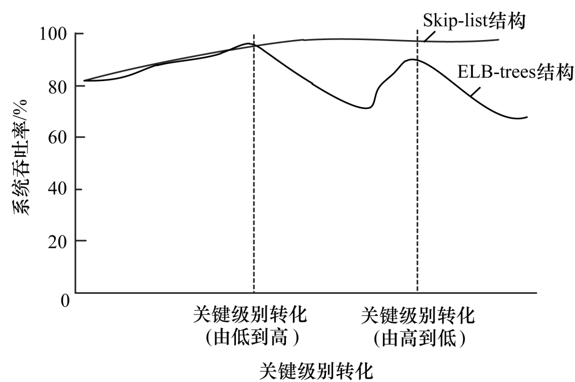

图 7 关键级别转化过程中 2 种数据结构系统吞吐率的比较

4 结束语

本文从混合关键系统调度过程中的同步机制出发,分析了基于传统锁机制的共享优先级队列面临的问题,提出一种基于 COSBF 的共享资源访问机制及设计方案。给出了锁与无锁算法相结合的 Skip-list 数据结构和 Bloom Filter 的实现过程。实验结果表明,该方案在平均响应时间和核利用率方面都表现较优。下一步的研究方向是将这种锁与无锁相结合的机制扩展到更多的数据结构中。

参考文献

[1] Kristijan D,Daniel B. A Survey of Concurrent Priority Queue Algorithms[C]//Proceedings of IEEE International Parallel & Distributed Processing Symposium. Washington D. C., USA:IEEE Computer Society,2008:1-6.

[2] Nir S, Itay L. Skiplist-based Concurrent Priority Queues[C]//Proceedings of the 14th International Parallel and Distributed Processing Symposium. Washington D. C., USA: IEEE Computer Society, 2000: 263-268.

[3] Aleksandar P, Dmitry P, Martin O. Efficient Lock-free Work-stealing Iterators for Data-parallel Collections[C]//Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing. Washington D. C.,USA:IEEE Computer Society,2015:248-252.

[4] 李鹏飞,张坤龙,康超凡. 基于低冲突帮助机制的快速无等待哈希表算法[J]. 计算机工程,2015,41(11):52-58.

[5] 高志民,姚 崎. 面向并行安全网关流水线模型的无锁队列算法[J]. 北京交通大学学报,2010,34(5):8-13.

[6] 吴 昊,季振洲,朱素霞. 基于硬件 CAS 原语的高效多字无锁同步算法[J]. 电子学报,2013,41(11):2127-2134.

[7] 姚 崎,刘吉强,韩 臻,等. 面向多核处理器的 Linux 网络报文缓冲区重用机制研究[J]. 通信学报,2009,30(9):102-108.

[8] 涂登彪,谭光明,孙凝晖. 无锁同步的细粒度并行介度中心算法[J],软件学报,2011,22(5):986-995.

[9] 高 蕾,赖明澈,龚正虎. 一种 TBGP 协议无阻塞路由通告技术[J]. 华中科技大学学报,2009,37(12):59-63.

[10] Bonnichsen L F, Karlsson S, Probst C W. ELB-trees an Efficient and Lock-free B-tree Derivative [C]//Proceedings of the 6th IEEE International Workshop on Multi-/Many-core Computing Systems. Washington D. C., USA:IEEE Computer Society,2013:1-10.

[11] Zhang Deli, Dechev D. A Lock-free Priority Queue Design Based on Multi-dimensional Linked Lists[J]. Journal of IEEE Transactions on Parallel and Distributed Systems,2016,27(3):613-626.

[12] Changwoo M, Young I E. Integrating Lock-free and Combining Techniques for a Practical and Scalable FIFO Queue[J]. Journal of IEEE Transactions on Parallel and Distributed Systems,2015,26(1):1910-1922.

[13] 王文义,王兴启. 基于多核处理器的有锁编程与非阻塞算法研究[J]. 中原工学院学报,2010,21(4):15-18.

[14] Brodera A, Mitzenmacher M. Network Applications of Bloom Filters: A Survey [J]. Journal of Internet Mathematics,2004,16(1):485-509.

[15] Rodrigo T S,Silvano D Z,Bernard B. A General Lock-free Algorithm for Parallel State Space Construction [C]//Proceedings of the 9th International Workshop on Parallel and Distributed Methods in Verification. Washington D. C., USA:IEEE Computer Society,2010:8-16.

编辑 刘 冰