第六讲 自旋锁和争用

一．引言

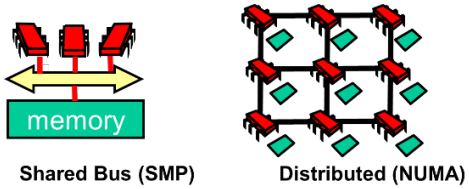
前几讲关心的是并发对象的正确性和可计算性。从本讲开始，我们逐一推出典型并发对象的具体实现，并分析其应用性能。在顺序计算中，系统底层对程序员是透明的。但在并发计算中，实现一个高性能的并发对象，必须了解底层的系统结构。按照 Flynn 的分类，计算可分为三类：

（1）SISD (单处理器)：单指令流、单数据流。

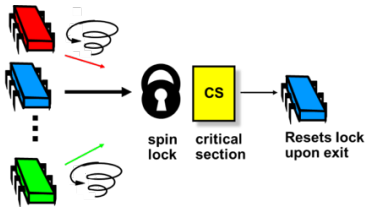
（2）SIMD (向量)：单指令流、多数据流。

（3）MIMD (多处理器)：多指令流、多数据流。

共享内存计算属于第三类，存在两种架构，见下图。一种是 SMP（UMA）架构，通常采用共享总线的方式来访问内存，处理器对内存单元的访问是对称的；另一种是分布式的(CC)-NUMA 架构，处理器通过内部互联网访问内存，一个处理器访问本地内存（自己结点的内存）的访问周期远低于访问其它结点内存的访问周期。两种架构在内存争用、通信竞争和通信延迟等方面存在较大差异。



锁是并发编程中最常用的同步和互斥方法。我们在前面提到，锁并不是基本的构造单位。本讲讨论如何正确、高效地实现并发线程使用的共享锁机制。锁分为两类，一类是阻塞的自旋锁，遇忙等待；另一类是非阻塞的，遇忙休眠。多核计算中更多使用自旋锁。本讲介绍各种自旋锁的实现方式。下一讲会介绍非阻塞锁和更一般的同步方式。



在性能方面，主要有两个重要指标：**延迟和吞吐率**。延迟是一个方法从调用开始到结束所经历的时间；吞吐率是在单位时间内完成的方法调用的个数。并发系统一般更关心吞吐率指标。在本讲，针对最基础的并发对象，我们着重考察每个锁方法调用的平均延迟。

影响锁的性能主要有两个因素，一是热点竞争造成的性能损失，多个线程申请同一把锁，在竞争中相互干扰，这些无用功和开销会导致共享锁的性能下降，应尽量避免；另一个因素来自锁本身的顺序特性，即使在没有热点竞争的情况下，获得锁的线程也只能顺序访问临界区。这个顺序瓶颈因素对性能的影响是无法避免的。在设计锁时，主要考察热点竞争的根源以及如何消除热点竞争的影响。

二．TAS 锁和 TTAS 锁

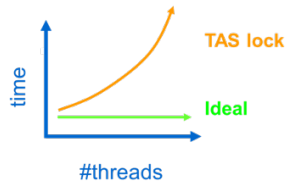
最简单的锁是用 RMW 寄存器构造的 TAS（Test-and-Set）锁。getAndSet 方法以原子的方式交换寄存器的布尔值，其返回值告诉我们之前的值是真还是假。锁的方法包括 lock()和 unlock()。

```
public class AtomicBoolean { //java.util.concurrent.atomic
  boolean value;
  public synchronized boolean getAndSet(boolean newValue) { //交换
     boolean prior = value;
     value = newValue;
     return prior;
  }
}
```

TAS 锁

```
class TASlock {
  AtomicBoolean state = new AtomicBoolean(false); //锁的状态
  void lock() {
    while (state.getAndSet(true)) {} //自旋，直到获得锁（状态从假到真）
  }
  void unlock() {
    state.set(false); //释放锁
  }
}
```

自旋锁使用 O(1)大小的空间（常量级）。理想状态下，每次锁请求和释放所需要的时间应该是不变的。所以，随着线程数的增加，理想的延迟时间应该是一条水平线。TAS 锁的延迟却随着线程数的增加而急剧增长，表明其可扩展性（scalability）很差。



下面把 TAS 锁升级为 TTAS 锁（Test-and-Test-and-Set Lock），其加锁过程分为两个阶段：

1、潜伏阶段：持续对寄存器进行读操作，直到锁“看起来”是空闲的；

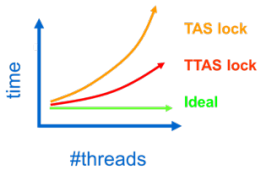
2、突袭状态：执行加锁操作（getAndSet），如果失败，则回到潜伏阶段。

释放 TTAS 锁的过程与 TAS 锁相同。

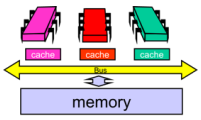
实现代码：

```
class TTASlock {
  AtomicBoolean state = new AtomicBoolean(false);
  void lock() {
    while (true) {
      while (state.get()) {} //自旋读，直到锁被释放
      if (!state.getAndSet(true))
        return; //加锁成功
    }
}}
```

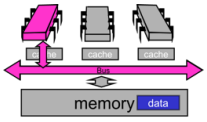
从性能上看，TTAS 锁的性能比 TAS 锁有一定提升，但可扩展性依然较差。



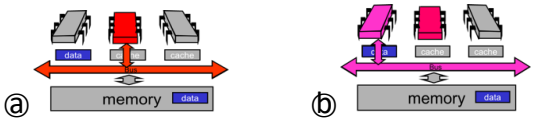
下面从系统架构的角度分析锁的性能（热点竞争的根源）。基于总线的结构如下：



总线具有广播机制，在任何时候，若多处理器或内存中有一方在发送信息，其它各方都在监听。处理器与缓存（cache）之间的交互是非常快的，而处理器与内存之间的通信一般会慢一个数量级。处理器先访问本地 cache，如果命中则直接返回数据，否则需要从内存或其它 cache 中把数据存入本地 cache，并返回数据。当线程请求数据时，发送广播，内存会把数据发送至相应的缓存。

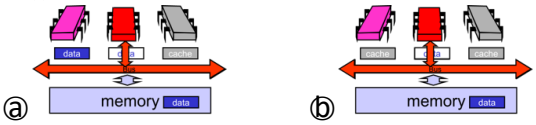


当另一个线程请求同样的数据（如下图ⓐ所示）时，已获得数据的 cache 可以把数据发送至相应的缓存（如下图ⓑ所示），即不从内存中读取该数据。

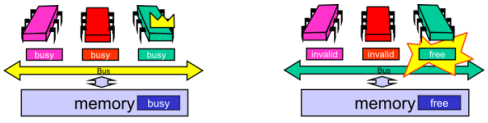


当多个数据副本存在于 cache 和内存中时，若有线程修改数据，如何避免混淆？答案是 cache 一致性协议（cache coherence protocol）。

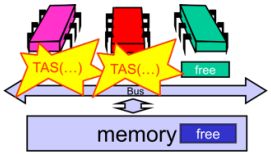
Cache 一致性协议维护每个内存单元的多个数据副本之间的一致性，就像单核系统中的缓存一样。以写无效的 cache 一致性协议为例。在该协议中，缓存有三个状态：Invalid（无效）表示无数据；Valid（有效）表示有数据，且与内存一致；Dirty（脏）表示数据已被修改，与内存不一致。在修改数据时会拦截其它数据请求（如下图ⓐ所示），并禁止其它线程读取；告知其它 cache，数据无效，然后写入内存（如下图ⓑ所示）。



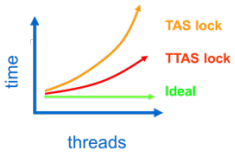
TAS 加锁时调用的 getAndSet()操作会锁住总线，其它线程只能等待。总线本身是一个顺序瓶颈。



TTAS 加锁时的自旋读操作，在锁被占用的情况下，除了第一次是从内存把 false 读至本地 cache，其余时间都忙等在本地 cache，不占据总线，从而减少了对总线的争用，所以 TTAS 锁的性能优于 TAS 锁。但当锁被释放时，TTAS 锁和 TAS 锁一样，cache 一致性协议会 invalidate 其它 cache 中的数据副本，导致所有其它线程几乎同时调用 test-and-set 操作，这大大降低了 TTAS 锁的性能。



从正确性的角度而言，TAS 锁与 TTAS 锁是等价的，都保证了无死锁互斥。TTAS 锁可以减少一定的总线争用，但这两种锁的性能并不理想。



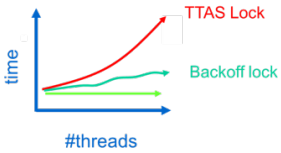
三、回退锁（Backoff Lock）

如果锁看起来是空闲的，但却没能获得，避免争用的好办法是回退一段时间，而不是立刻再次争用。争用是指多个线程试图同时获得锁。当线程发现锁空闲但却没能获得时，说明存在争用，在重试前先回退。为确保发生争用的并发线程不会进入锁步（lock-step）模式，线程回退的时长应该是随机的。当线程再次争用失败时，回退时长的上限应加倍。

实现

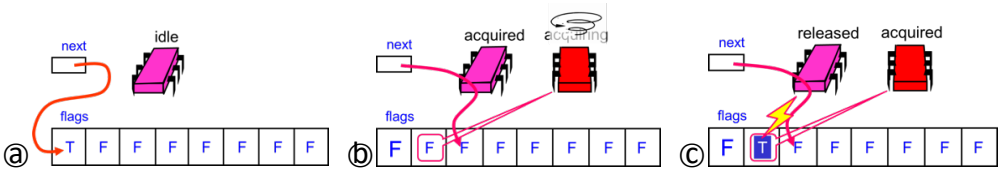
```
public class Backoff implements lock {
  public void lock() {
    int delay = MIN_DELAY; //回退时长的上限
    while (true) {
      while (state.get()) {} //等待锁空闲
      if (!lock.getAndSet(true))
        return; //获得锁
      sleep(random() % delay); //随机回退一段时间
      if (delay < MAX_DELAY)
        delay = 2 * delay; //回退时长的上限加倍
}}}
```

回退锁的性能比 TTAS 锁好得多，但性能波动较大，与 MIN_DELAY 和 MAX_DELAY 的参数设定紧密相关。在部署回退锁时，需要测试不同的参数设定。实验表明最优值与处理器个数和运算速度密切相关。



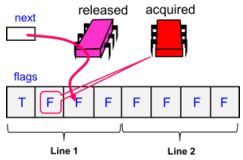
四、队列锁（Queue Lock）

有效解决争用的办法仍然是将单个锁变量扩充为锁数组或者队列，以资源换效率。最早的队列锁是 Anderson 提出的。Anderson Queue Lock 是一个布尔数组 flag 构成的队列。每个线程对应 flag 数组中的一位，称为槽（slot）。如果 flag[j]为 T，那么槽 j 的线程可以获得锁。初始时，flag[0]为 T（下图ⓐ）。加锁时，线程首先调用 getAndIncremental()进行排队，指定自己的 slot，然后忙等在该 slot 上（下图ⓑ），直到该 slot 的 flag 变为 T（下图©）。释放锁时，线程把自己的 slot 变为 F，将下一个 slot 变为 T（下图©）。

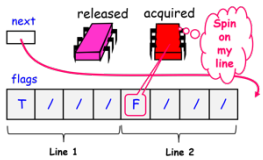



```
class ALock implements Lock {
  boolean[] flags={true,false,…,false};
  AtomicInteger next = new AtomicInteger(0); //下一个等待 slot
  ThreadLocal<Integer> mySlot;
  public lock() {
    mySlot = next.getAndIncrement(); //设置下一个等待 slot
    while (!flags[mySlot % n]) {}; //自旋，忙等在自己的 slot 上
    flags[mySlot % n] = false; //获得锁，占用中
  }
  public unlock() {
    flags[(mySlot+1) % n] = true; //下一个线程可用
  }
}
```

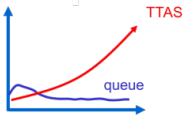
线程在申请锁时，只忙等在自己的 slot 上，实际上是忙等在本地 cache 上，可以有效消除热点争用。但争用仍有可能发生。



存在一种假共享现象，当相邻的数据项共享单一 cache 线时，对其中一个数据项的写会使该数据项所在的 cache 线无效。对于那些恰好进入同一 cache 线但对应于其它线程的 slot 来说，虽然它们的值没有改变，但由于整个 cache 线已无效，这些线程被迫从内存重读 false 至相应的 cache 中，导致性能损失。避免假共享的方法是填充数组项，使每个数组项都独占一行 cache 线，不同的数据项被映射至不同的 cache 线中。



与 TTAS 锁对比，Anderson 队列锁的性能曲线几乎是水平的，具有很好的可扩展性。而且，Anderson 锁具有 FIFO 特性，能够保证系统的公平性。Anderson 锁的唯一缺点是资源开销太大，每个数组项须独占一行 cache 线。

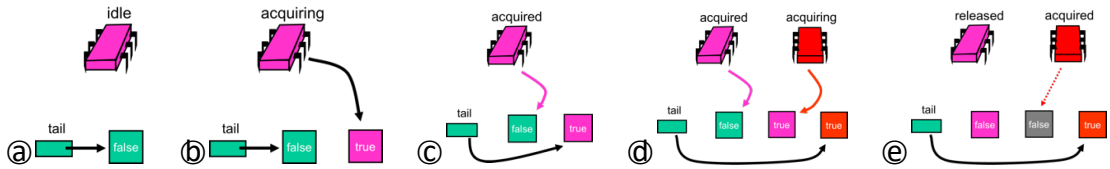


五．CLH 队列锁

CLH 队列锁是 Craig、Landin 和 Hagersten 合作提出的，解决了 Anderson 锁占用资源过多的缺点。CLH 锁的队列与通常的队列不同，其链接是隐式的，由线程

的局部变量和 Qnode 结点共同构成。Qnode 结点记录了锁的状态。如果其状态域为 true，则相应的线程要么已经获得锁，要么正在等待锁；如果该域为 false，则相应的线程已经释放了锁。

初始化时，tail（队尾）指向 Qnode 对象（下图ⓐ）。粉色线程申请锁时，创建其 Qnode 对象，并将其状态域置为 true（下图ⓑ）。随后，交换 tail 和其 Qnode 结点引用，使该 Qnode 结点成为新队尾，同时获取指向前驱 Qnode 结点的引用（即原队尾，下图ⓒ）。线程忙等在前驱结点上。接下来，红色线程申请锁时，创建其 Qnode 对象，将其状态域置为 true，交换 tail 和其 Qnode 结点引用（下图ⓓ）。红色线程忙等在前驱结点上（即粉色线程创建的 Qnode 结点）。粉色线程释放锁时，修改其 Qnode 结点的状态域为 false，该结点可以被红色线程将来复用。红色线程发现前驱结点的状态域变成 false，即可获得锁（下图ⓔ）。



Alock 锁的空间开销大，同步 L 个不同的锁对象需要 *O*(LN)大小的空间。CLH 锁只需要 *O*(L+N)。其中，L 是锁的个数，N 是线程个数。

CLH 队列锁实现

```
class Qnode {
  AtomicBoolean locked = new AtomicBoolean(true); //新加入的结点
}
class CLHLock implements Lock {
  AtomicReference<Qnode> tail = new Qnode(); //队尾
  tail.locked.set(false); //队尾初始时，锁可用
  ThreadLocal<Qnode> myNode = new Qnode(); //线程 Qnode
  public void lock() {
    Qnode pred = tail.getAndSet(myNode); //加入队尾
    while (pred.locked) {}; //自旋在 pred 结点
  }
  public void unlock() {
    myNode.locked.set(false); //通知后继线程
    myNode = pred; //回收前驱 Qnode 结点，并复用
  }
}
```

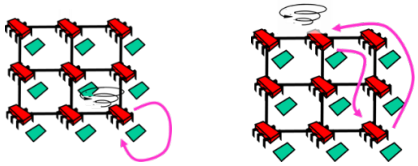
注：lock()方法允许多个线程（线程数大于 2）同时调用 getAndSet()方法访问 tail，其原子性保证了每个调用都会返回，但返回值不同（这与共识协议的要求不同），相当于这些线程把各自的 Qnode 结点“顺序地”加入队尾。仅 tail 初始化时所指向的 Qnode 结点的状态域为 false，表示锁可用。

优点：常量级空间，满足 FIFO 特性，释放锁只使其后继线程对应的 cache 无效，性能损耗很少（在 UMA 架构下）。

缺点：在无 cache 的 NUMA 系统架构下性能很差。

NUMA 结构：非均匀内存访问架构，Non-Uniform Memory Access。在此架构下，线程访问本地处理器内存的速度远快于访问其它处理器的内存。申请 CLH 锁的线程忙等在前驱 Qnode 结点上。在 NUMA 架构没有 cache 的情况下，每次

查询可能需要访问远程处理器的内存，性能损失很大。

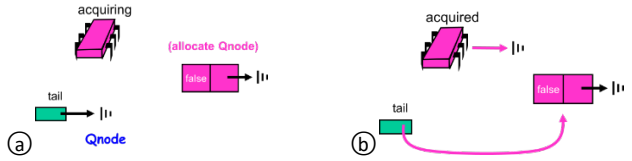


解决的办法是让每个线程忙等在自己的 Qnode 结点上，即 MCS 队列锁。

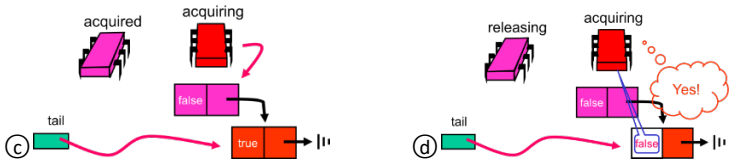
六．MCS 队列锁。

MCS 队列锁是 Mellor-Crummey 和 Scott 提出的，满足 FIFO 特性，只在本地内存单元自旋，拥有常量级空间大小。看起来是一个很理想的 spin 锁（在不考虑 abortable 的情况下）。

MCS 队列锁的链表是显式的，每个 Qnode 结点包含指向后继结点的 next 指针。初始化时，tail 指向 null 结点。粉色线程申请锁时，发现 tail 指向 null 结点，表明没有前驱结点（下图ⓐ）。粉色线程直接获得锁，并将其 Qnode 结点加入队尾（下图ⓑ）。



接下来，红色线程申请锁，发现 tail 指向非 null 结点，创建其 Qnode 结点，将其状态域置为 false，并将该 Qnode 加入队尾（修改原队尾指向结点的 next 指针指向该 Qnode 结点，然后忙等在此 Qnode 结点上，直到前驱线程修改此 Qnode 结点的状态域为 false。粉色线程释放锁时，把后继结点的状态域置为 false，红色线程即获得锁，粉色线程的 Qnode 结点可回收复用。



MCS 队列锁实现

```
class Qnode {
    boolean locked = false;
    Qnode next = null;
}
class MCSLock implements Lock {
    tail = new AtomicReference<Qnode>(null);
    public void lock() {
        Qnode Qnode = new Qnode(); //新结点
        Qnode pred = tail.getAndSet(Qnode); //加入队尾
        if (pred != null) { //若队列不空
                Qnode.locked = true; //准备自旋
                pred.next = Qnode; //将前驱结点的 next 指向新结点
                while (Qnode.locked) {} //在新结点上自旋
    }}
```

```
public void unlock() {
    if (Qnode.next == null) {
        if (tail.CAS(Qnode, null) //没有后继线程
            return;
        while (Qnode.next == null) {} //等待后继结点加入队尾
    }
    Qnode.next.locked = false; //通知后继结点
}}
```

注：在 unlock()方法中，**if (tail.CAS(Qnode, null) return;**一句非常重要。它检查当前线程的 Qnode 结点是否是 tail 指向的最后一个结点。如果是，那么 tail 直接指向 null 结点，队列为空；否则，表明后继线程正在将其 Qnode 结点加入队尾，但这个链接过程还未完成！这时，调用 unlock()的线程（解锁线程）需要忙等，直到链接完成。然后，解锁线程将后继 Qnode 结点的状态域置为 false，完成锁的释放。

七．超时锁

在实际系统中，特别是实时系统中，通常要求线程在一段时间内申请锁。如果在这段时间内没获得锁，则中止申请。具有这一特性的锁称为可中止的锁（Abortable lock）。TAS 锁、TTAS 锁和回退锁本身就具有可中止特性，但对于队列锁而言，若线程半途中止，但其 Qnode 结点在队列中没有妥善处理的话，后继的所有结点对应的线程可能会陷入无穷等待。

以 CLH 锁为例，考察可超时中止的 CLH 锁。为了应对超时中止的情况，Qnode 结点中增加了指向前驱结点的指针 prev。等待锁的线程可通过 prev 指针组成的链条反向回溯。此外，增加一个固定（静态）结点 Available，表示锁已释放，可以直接获得锁。回溯的过程是反向通过那些超时中止的结点，一直回溯到最近一个等待中的结点，或者回溯到 Available 结点。

```
class Qnode {
   Qnode prev = null;
}
public class TOLock implements Lock {
   static Qnode AVAILABLE = new Qnode(); //哨兵结点
   tail = new AtomicReference<Qnode>(null); //队尾
   ThreadLocal<Qnode> myNode;
   public boolean lock(long timeout) {
      Qnode Qnode = new Qnode(); //创建并初始化
      myNode.set(Qnode);
      Qnode.prev = null;
      Qnode myPred = tail.getAndSet(Qnode); //加入队尾
      if (myPred== null || myPred.prev == AVAILABLE) {
      //前驱结点为空或者锁已释放
         return true;
      }
      long start = now(); //等待
      while (now()- start < timeout) {
```

```
            //在规定时间内自旋在 predPred 上
            Qnode predPred = myPred.prev;
            if (predPred == AVAILABLE) {//锁空闲
               return true;
            } else if (predPred != null) {
               myPred = predPred;
            }
         }
         if (!tail.compareAndSet(Qnode, myPred))
         //超时尝试删除 Qnode。若删除失败，表明存在后继结点
            Qnode.prev = myPred;
         return false;
      }
      public void unlock() {
         Qnode Qnode = myNode.get();
         if (!tail.compareAndSet(Qnode, null))
         //检查是否存在后继结点，若存在，则设置 prev，通知后继结点
          Qnode.prev = AVAILABLE;
   }}
```

标注：超时锁结合了 CLH 锁和 MCS 锁的特点，尤其是 unlock()方法。之所以说是 CLH 锁的扩展，主要原因在于超时锁也是忙等在前驱结点而不是自己的结点上，这是 CLH 锁最本质的特征。

超时锁通过复用 Available 结点，可节约内存开销。Available 结点可以用在每个 Qnode 结点中增加一个布尔域 locked 来代替。以下是无 Available 结点的版本。

```
   class Qnode {
        boolean locked = true;
        Qnode prev = null;
   }
   public class TOLock implements Lock {
      AtomicReference<Qnode> tail = new Qnode(); //队尾
      tail.locked = false; //队尾初始时，锁可用
      ThreadLocal<Qnode> myNode;
      public boolean lock(long timeout) {
         Qnode Qnode = new Qnode(); //创建并初始化
         myNode.set(Qnode);
         Qnode.prev = null;
         Qnode myPred = tail.getAndSet(Qnode); //交换，加入队尾
         if (myPred.prev == null && myPred.locked == false) {
         //前驱结点为空或者锁已释放
            return true;
         }
         long start = now();//等待
         while (now()- start < timeout) {
         //在规定时间内自旋在 predPred
```

```
            Qnode predPred = myPred.prev;
            if (predPred.prev == null && predPred.locked == false) { //锁已释放
               return true;
            } else if (predPred.prev != null) {
               myPred = predPred;
            }
         }
         if (!tail.compareAndSet(Qnode, myPred))
         //超时尝试删除 Qnode。若删除失败，表明存在后继结点
            Qnode.prev = myPred;
         return false;
      }
      public void unlock() {
         myNode.locked = false; //通知后继线程
   }}
```

八．总结

Spin Locks	MIMD	Deadlock -free	Starvation-free	Fairness	Space
Peterson/ Filter	✘	✓	✓	✘	$O(n)$
Bakery	✘	✓	✓	FIFO	$O(n)$
TAS/TTAS	✓	✓	✘	✘	$O(1)$
Anderson Queue	✓	✓	✓	FIFO	$O(n)$
CLH, MCS Queue	✓	✓	✓	FIFO	$O(n)$