并发算法与理论复习笔记

（一）概念

1. 并发算法的行为
2. 并发算法的性质
3. Linearizability：每个方法在它生效的那个地方指定一个可线性化点
    1. 可线性化点怎么找?
        1. 基于锁的实现：每个方法的临界区
        2. 不使用锁的实现：该方法调用的结果对其它方法调用可见的那个操作步

    例子：

        1. 带锁的queue：可线性化点在第10行释放锁的地方

```
1  public T deq() throwsEmptyException{
2      lock.lock();
3      try {
4          if (tail == head)
5              throw new EmptyException();
6          T x = items[head % items.length];
7          head++;
8          return x;
9      } finally {
10         lock.unlock();
11     }
12 }
```

        2. 无锁的queue：可线性化点分别在tail++和head++这里

```
1  public void enq(Item x) {
2      while(tail-head == capacity); // busy-wait
3      items[tail % capacity] = x;
4      tail++;
5  }
6  public Item deq() {
7      while(tail == head); // busy-wait
8      Item item = items[head % capacity];
9      head++;
10     return item;
11 }
```

    2. 可线性化点可以有多种情况，需要对方法的每个分支（情况）做分类讨论；也可能同一种分支有多个可线性化点，需要根据语句执行拓扑序分类讨论

    可线性化性定义：如果经历H存在一个扩展H'及一个合法的顺序经历S，并使得

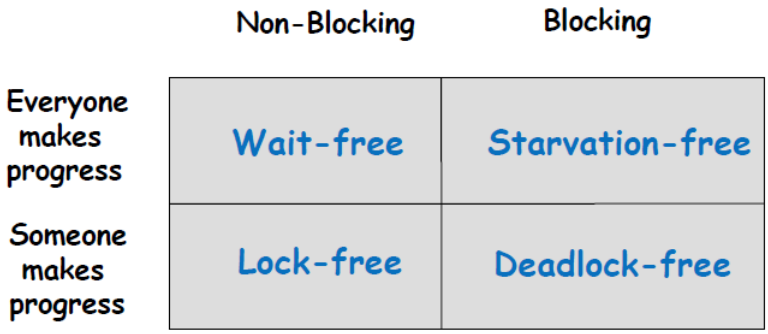
        1. complete(H')与S等价，且
        2. 若在H中的方法调用m_0先于m_1，那么在S中也成立

    那么称经历H是可线性化的

    注：

    1. 扩展：对0个或多个未决调用增加了响应
    2. complete(H')：只有匹配的调用和响应构成的H'的子序列
    3. 合法的顺序经历：每个对象的子经历对该对象都正确
    4. 经历G与S等价：例如若$\rightarrow_G = \{b \rightarrow c\}$，$\rightarrow_S = \{b \rightarrow a, a \rightarrow c, b \rightarrow c\}$，则$\rightarrow_G \subseteq \rightarrow_S$，即G和S等价
3. 特点：
    1. 复合性：对每个对象x，当且仅当H|x是可线性化的，H是可线性化的——重要的，可以对对象分别证明线性化性，从而证明经历的可线性化
    2. 非阻塞特性：一个完全方法的未决调用不需要等待另一个未决调用完成
4. 区别：顺序一致性：没有可复合性，但是同样非阻塞，相比可线性化去掉了$\rightarrow_G \subseteq \rightarrow_S$条件，即：
    1. 同一线程中的顺序不可以改，但是不同线程中不重叠的操作顺序可以改

4. 对于进程的四个Free
    1. Lock-free：**某个**调用**方法**的**线程**最终返回/保证某个方法的无限次调用都能在有限步内完成。需要假设公平调度
        1. Wait-free可以直接推Lock-free
    2. Wait-free：**每个**调用**方法**的**线程**最终都会返回/对一个并发对象的每一次方法调用都能在有限步内完成。需要假设公平调度
        1. 一个直接的判断标准：如果方法没有循环，那么就是wait-free的
    3. Deadlock-free：**某个**线程试图获取**锁**最终成功。不需要假设公平调度
    4. Starvation-free：**所有**试图获取**锁**的读取最终都成功了。不需要假设公平调度



5. Safety和Liveness
    1. Safety：永远不会有错误结果，如果有，那么可以在有限时间/步骤内发现（偏序性）
    2. Liveness：程序将会有结果，如果没有，那么我们也不知道什么时候可以观察到（可终止性）
    3. 例子：
        1. Mutual exclusion: safety，不可能有两个线程同时在临界区，如果违反，可以马上发现
        2. Bounded overtaking: safety，另一个线程至多占用N次临界区以后能进
        3. Starvation-freedom: liveness，想进临界区总能进
        4. Deadlock-freedom: liveness，想进临界区总能进且多个同时想进时总有一个能进
6. 对于锁的性质
    1. Mutual Exclusion: safety
        1. 定义：$CS_A \rightarrow CS_B$
        2. 证明思路：
            1. 方法一：反证法。假设不满足互斥，同时出现在临界区，利用反证法证明满足互斥
            2. 方法二：状态机法。证明初始状态满足互斥，并且任意一个状态向另一个状态的转换也满足互斥
    2. Deadlock-freedom: liveness

1. 定义：线程总能获得锁，如果调用`lock()`却不能获得锁，一定有其他线程正在无穷次执行临界区
2. 证明思路：
    1. 定义证明
    2. 无饥饿→无死锁
3. 定理：任意一种无死锁的Lock算法在最坏情况下至少需要读写n个不同的存储单元

3. Starvation-freedom: liveness
    1. 定义：每个获取锁的线程都能成功，每一个`lock()`调用都能返回
    2. 证明思路：
        1. 定义证明

7. Register——暂时说不考
    1. Safe Register：
        1. 如果没有重叠，那么返回旧值
        2. 如果读写重叠，返回任意一个可能的值
    2. Regular Register：
        1. 单写者
        2. 读者：
            1. 如果没有重叠，那么返回旧值
            2. 如果有重叠，那么返回旧值或其中一个新值
    3. Atomic Register：就是在顺序Safe Register中找到一个可线性化的结果的Register序列
8. Consensus——暂时说不考
    1. 定义
        1. 一致性Consistent：所有线程都决定同一个值
    2. 有效性Valid：这个共同的决定之是某个线程的输入
    3. 一致数：
        1. 原子寄存器的一致数为1
            1. 推论：对于任意一致数>1的对象，不可能用原子寄存器构造该对象的Wait-free实现
        2. 用一组原子寄存器不可能构造队列、栈、优先级队列、集合或链表的无等待实现
        3. FIFO队列实现双线程一致性：

```
public class QueueConsensus extendsConsensusProtocol {
    private Queue queue;
    public QueueConsensus() {
        queue = newQueue();
        queue.enq(Ball.RED);
        queue.enq(Ball.BLACK);
    }
    public decide(object value) {
        propose(value);
        Ball ball = this.queue.deq();
        if(ball == Ball.RED)
            returnproposed[i];
        else
            returnproposed[1-i];
    }
}
```

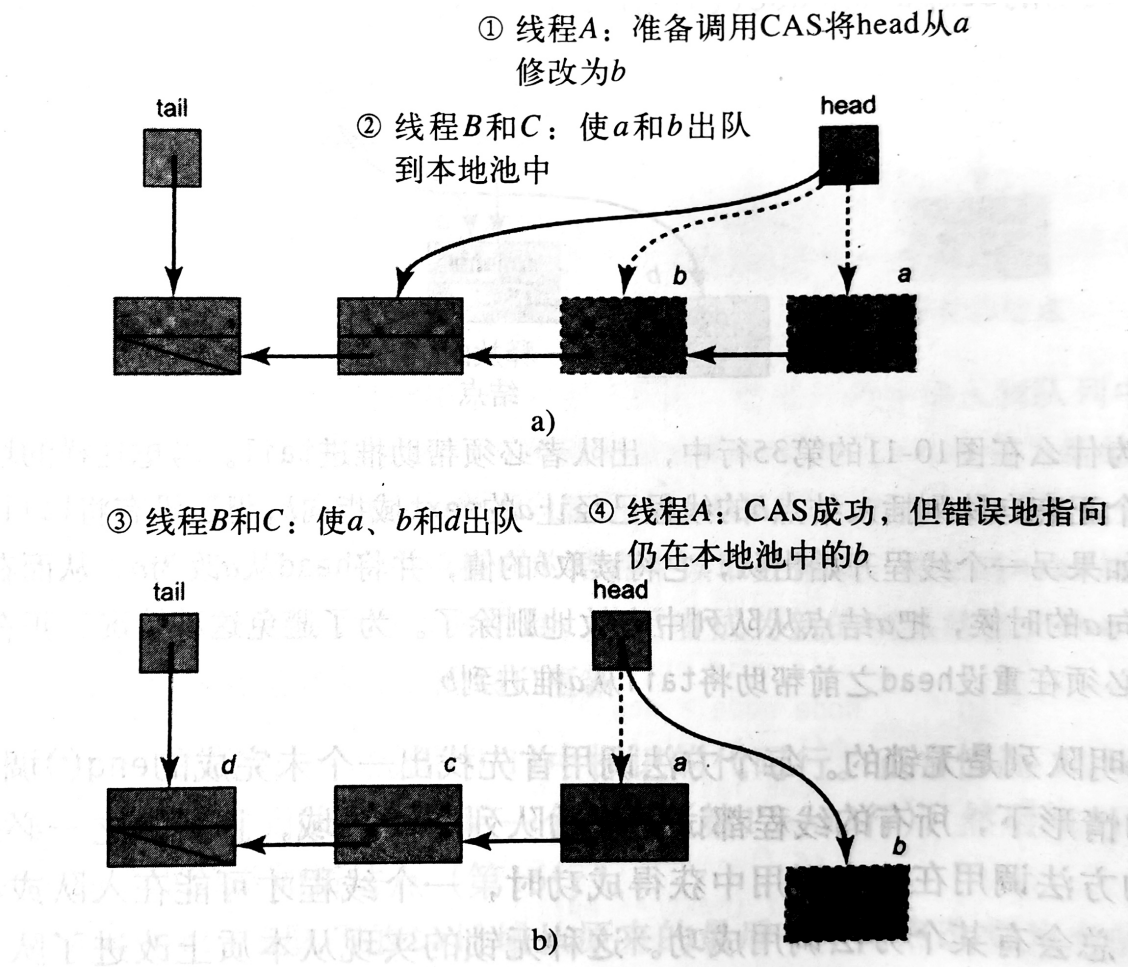
4. FIFO队列、集合、栈、链表、双端队列以及优先级队列的一致数均为2

9. Read Modify Write（RMW）：
    1. 定理：
        1. 任何非平凡的RMW寄存器的一致数至少为2
            1. 非平凡：RMW方法的函数集中至少包含一个非恒等函数
            2. 证明：如下，因为getAndMumble是非平凡的，所以mumble(v)≠v，因此可以让第一个线程get出v，但第二个线程get不出v，只能是自己。于是第一个get出v的线程决定自己的值，第二个get出自己的线程决定另一个线程的值。因此一致数至少为2

```
public class RMWConsensus extends ConsensusProtocol {
    private RMWRegister r = v;
    public Object decide(object value) {
        propose(value);
        if(r.getAndMumble() == v)
            return proposed[i];
        else
            return proposed[j];
    }
}
```

        2. 任意Common2 RMW寄存器的一致数都恰好为2
            1. Common2定义：令F是一个函数集，对于任意的值v以及F中的函数f_i和f_j，如果他们满足如下条件之一
                1. 可交换Commute：$f_i(f_j(v)) = f_j(f_i(v))$；
            2. 可重写Overwrite：$f_i(f_j(v)) = f_i(v)$或$f_j(f_i(v)) = f_j(v)$，

            那么函数集F属于Common2
    2. getAndSet(v)/swap(v)：用v原子地替换寄存器当前值并返回先前值。
        1. 满足Overwrite特性，一致数=2
    3. compareAndSet(e,u)：如果寄存器值=e，那么用u原子地替换它，否则不改变，同时返回一个布尔值说明寄存器值是否被改变。
        1. 严格上说不是一个对于$f_{e,u}$的RMW方法，既不Commute，也不Overwrite，可以证明一致数=∞
    4. getAndIncrement()：将寄存器的当前值原子地加1并返回先前值。
        1. 满足Commute特性，一致数=2
    5. getAndDecrement()：将寄存器的当前值原子地减1并返回先前值。
        1. 满足Commute特性，一致数=2
    6. getAndAdd(k)：将寄存器地当前值原子地加1并返回先前值。
        1. 满足Commute特性，一致数=2
    7. get()：返回寄存器的当前值。
        1. 该方法是对于恒等函数的RMW方法，非平凡，一致数=1
10. ABA问题



1. 解决办法是对每个原子引用附上唯一的时间戳，对象的引用和时间戳可以单独也可以同时原子地被修改。每次修改对象的时候，就将时间戳的值+1

（二）算法

一、锁算法

1、Peterson Lock

1. 算法

```
public void lock() {
    flag[i] =true;
    victim = i;
    while(flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] =false;
}
```

2. 特点：
    1. 互斥：用flag、victim的read、write顺序可以证明
    2. 无饥饿：若A阻塞，那么B可能：
        1. 反复进入临界区又离开，但B每次进入临界区要设置victim=B，此时A就会从lock()方法返回，矛盾；
        2. 陷入lock()方法调用，等待flag[A]变为false或victim=A，但victim不可能同时为A或B，矛盾。

因此无饥饿得证

3. 无死锁：无饥饿可以直接导出无死锁

2、Filter Lock

1. 算法

```
public void lock(){
    for (int L = 1; L < n; L++) {
        level[i]  = L;
        victim[L] = i;
        while ((∃k != i level[k] >= L) && victim[L] == i );
    }
}
public void unlock() {
    level[i] = 0;
}
```

2. 特点：
    1. 对于0~n-1中的整数j，层j上最多有n-j个线程：步骤为
        1. 归纳法：0显然成立，设层j-1中最多有n-j+1个线程
        2. 然后对演绎步骤应用反证法：假设层j中有n-j+1个线程，反推最后一个写victim[j]的线程一定会在while等待
        3. 得证
    2. 互斥性：进入临界区等价于进入层n-1（最多只能有一个线程进入）
    3. 无饥饿：步骤为
        1. 对层数进行反向归纳：对层n-1显然成立
        2. 演绎步骤：假设层j+1或更高层的线程最终都能进入并离开临界区；如果有线程被阻塞在层j，那么比j高的都没有线程了，而且比j低的线程因为自己还在第j层而进不来（阻塞在while循环的第一个条件，不会修改victim），因此第j层的线程一定会往上走
        3. 最后是第j层线程们，victim只有一个，其它都会往上走
    4. 无死锁：无饥饿可以直接导出无死锁

3、Lamport's Bakery Lock

1. 算法

```
public void lock() {
    flag[i] = true;//门廊区
    label[i] = max(label[0], …,label[n-1])+1;//门廊区
    while(∃k!=i flag[k] && (label[k],k) < (label[i],i));
public void unlock() {
    flag[i] = false;
}
```

2. 特点
    1. 无死锁：必定有一个线程具有最小的(label[k],k)，那么它不会等待其它线程
    2. 先来先服务：如果A的门廊区先于B的，即$D_A \rightarrow D_B$，那么根据read，write顺序（展开说），A的label必定小于B的label，所以flag[A]=true，B被封锁在外不能进入
        1. 先来先服务的定义：如果门廊区$D_A \rightarrow D_B$，则$CS_A \rightarrow CS_B$，即线程A必定不会被线程B赶超
    3. 无饥饿：无死锁且先来先服务可以直接导出无饥饿

4. 互斥性：反证法，假设(label[A],A)<(label[B],B)，B退出while循环时必定得到flag[A]=false或(label[B],B)<(label[A],A)，但是因为线程的id固定且label递增，所以只可能flag[A]=false，但是根据read，write顺序知道A一定不晚于B出门廊区，会先把flag[A]设置成true，所以矛盾。

4、Test-And-Set Lock（TAS自旋锁）

1. 算法

```
class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock() {
        while(state.getAndSet(true)) {}
    }
    public void unlock() {
        state.set(false);
    }
}
```

2. 特点

    1. N线程的自旋锁使用$O(1)$的空间，相对的是Peterson/Bakery锁的$O(n)$空间
        1. 我们怎么克服$\Omega(n)$的空间下界？因为我们使用了RMW操作
    2. 缺点：性能非常差，原因是
        1. 每个getAndSet()实质上是总线上的广播，每个getAndSet会延迟所有线程同步数据
        2. 更为糟糕的是，getAndSet()会迫使其它处理器丢弃他们自己cache中的锁副本，这样每个在自旋的线程几乎每次都会遇到cache miss，并且必须通过总线获取新的未被修改的值
        3. 更糟糕的是，当持有锁的线程试图释放锁的时候，总线被正在自旋的锁独占，该线程可能会被延迟
    3. 简单的总结：
        1. 自旋的线程：cache miss，会占用总线
        2. 释放锁的线程：等待使用总线，产生延迟
        3. 有饥饿，没有先来先服务的公平性

5、Test-Test-And-Set Lock（TTAS自旋锁）

1. 算法

```
class TTASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock() {
        while(true) {
            while(state.get()) {}//等待直到lock看起来false了
                if(!state.getAndSet(true))//调用TAS获取锁，成功则return，失败就重新等待
                    return;
        }
    }
    public void unlock(){
        state.set(false);
    }
}
```

2. 特点

1. 优点：
    1. 相对于TASLock而言，B第一次读锁的时候发生cache miss，从而阻塞等待值被载入它的cache中。此后，只要A持有锁，B就不停地重读该值，且每次都命中cache→这样不产生总线流量，也不会降低其它线程的访存速度
    2. 释放锁的线程不会被正在该锁上自旋的线程延迟
2. 缺点：
    1. 释放锁的时候锁的持有者将false写入锁变量，使得自旋线程的cache副本立刻失效，每个线程都会发生一次cache miss并重读该值，几乎同时调用getAndSet获取锁。第一个成功的线程将使其它线程再次失效，这些失效线程接下来又重读那个值→总线的流量风暴
3. 总结：
    1. 自旋线程：在本地cache上自旋，不会使用总线
    2. 释放锁：不会被自旋的线程延迟，但会有总线的流量风暴
    3. 有饥饿，没有先来先服务的公平性

6、Exponential Backoff Lock

1. 算法

```
public class BackoffLock implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while(true) {
            while(state.get()) {}
                if(!lock.getAndSet(true))//争用的时候赢了，就return
                    return;
                sleep(random() % delay);//后退
                if(delay < MAX_DELAY)
                    delay = 2 * delay;//指数扩大delay的值
        }
    }
}
```

2. 特点：
    1. 优点：易于实现、比TTAS Lock性能更好
    2. 缺点：
        1. 必须仔细选择MIN_DELAY和MAX_DELAY参数，且不同系统的参数不通用
        2. 每一次成功的锁访问都会产生cache一致性流量
        3. 线程延迟过长，导致临界区利用效率低下
        4. 有饥饿，没有先来先服务的公平性

7、Anderson’s Array-Based Queue Lock

1. 算法

```
class ALock implements Lock {
    boolean[] flags={true,false,…,false};//长度是线程个数
    AtomicInteger next = newAtomicInteger(0);//下一个用来等待的flag
    ThreadLocal<Integer>mySlot;//记录线程本身在flag的哪里等待
    public void lock() {
        mySlot = next.getAndIncrement();//获得下一个slot
        while(!flags[mySlot % n]) {};//自旋，直到自己等的那个坑变成True
        flags[mySlot % n] = false;//for reuse
    }
    public void unlock() {
        flags[(mySlot+1) % n] = true;//通知下一个线程可以开始了
    }
}
```

2. 特点：
    1. 优点：
        1. 每个线程在不同的存储单元上旋转，从而降低cache一致性流量。相比于backoff有更低的移交时间
        2. 提高临界区利用率
        3. 弹性的表现，可以适用不同线程数和系统
        4. 无饥饿、提供先来先服务的公平性b
    2. 缺点
        1. 空间不有效：要求知道并发线程的最大个数n，同时每个线程需要分配一个bit的数组。同步L个对象需要$O(Ln)$大小的空间

8、CLH Queue Lock

1. 算法

```
class Qnode{
    AtomicBooleanlocked = new AtomicBoolean(false);//true:锁还未释放，false:锁空闲
}
class CLHLock implements Lock {
    AtomicReference<Qnode> tail = new AtomicReference<Qnode>(new Qnode());
    ThreadLocal<Qnode> myNode = new Qnode();//初始化线程本地Qnode
    ThreadLocal<Qnode> pred = null;
    public void lock() {
        myNode.locked.set(true);//锁还没有被释放
        pred = tail.getAndSet(myNode);//用myNode来和tail交换
        while(pred.locked) {}//自旋直到前驱释放锁的时候
    }
    public void unlock() {
        myNode.locked.set(false);//notify后继
        myNode = pred;//复用前驱节点
    }
}
```

2. 特点：隐式链表
    1. 优点：
        1. L个锁/对象，n个线程的时候，ALock需要$O(Ln)$的空间，而CLHLock仅需$O(L+n)$的空间。每个线程仅需常数的空间

2. 在不同的存储单元上自旋，当线程释放锁的时候，只能使后继的cache无效
3. 不需要知道请求锁的线程的数量
4. 无饥饿，提供先来先服务的公平性

2. 缺点：在无cache的NUMA系统中性能较差，每个线程都需要等待前驱的lock域变为false，如果内存位置教员，那么性能将会损失

9、MCS Queue Lock

1. 算法

```
class Qnode {
    boolean locked = false;
    Qnode next = null;
}
class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        Qnode qnode = newQnode();
        Qnode pred = tail.getAndSet(qnode);//把自己的Qnode添加到链表尾部
        if(pred != null) {//如果pred为空，自己就是第一个节点，直接获取锁，否则执行
            qnode.locked = true;//自己锁住，可以自旋
            pred.next = qnode;//把前驱Qnode的next域指向自己的Qnode
            while(qnode.locked) {}//自旋直到前驱把自己的qnode.locked变为false
        }
    }
    public void unlock() {
        if(qnode.next == null) {//null有两种情况
            if(tail.CAS(qnode, null)//tail还是指向自己的qnode，那么置tail=null返回
                return;
            while(qnode.next == null) {}//tail指向下一个新的qnode，但自己的qnode.next还没指向下一个新的，就自旋直到自己的qnode.next指向下一个新的qnode
        }
        qnode.next.locked = false;//qnode.next非空了，就可以让next.locked=false了
    }
}
```

2. 特点：显式链表

1. 优点：具有CLH Lock的优点，包括

1. 先来先服务
2. 只在本地内存上自旋，并且每个锁释放仅能使后继的cache项无效
3. 每个线程仅需常数的空间
4. 适用于无cache的NUMA系统结构
5. 节点可以被重复使用
6. 空间复杂度为$O(L+n)$

2. 缺点：

1. 释放锁的时候需要旋转
2. 比CLH Lock算法的读、写和compareAndSet()调用次数多

二、List-Based Sets算法

> 因为本次考试不考，所以本内容略过，但以下稍做记录

1. 同步技术
    1. 细粒度同步：分解成同步组件，确保只有多个方法调用试图同时访问同一个组件的时候才发生冲突
    2. 乐观同步：查找时无需获取任何锁。如果方法找到所需的部分，就锁住该组件，然后确认该组件在被检测和上锁期间没有发生变化。只有当成功次数>失败次数时才有价值
    3. 惰性同步：将较难的工作推迟完成。例如删除操作先逻辑删除，在物理删除
    4. 非阻塞同步：有时可以完全消除锁，依靠类似compapreAndSet()的内置原子操作进行同步
        1. 优点：没有对调度的假设
        2. 缺点：复杂，可能有高的开销

1、FineList/Lock-Coupling List

1. 特点：注意可线性化点的确定需要分调用是否成功、待插入/删除/检测的节点在不在链表中讨论
    1. 优点：手把手上锁，无饥饿，无死锁
    2. 缺点：
        1. 可能出现长的获取锁和释放锁的序列
        2. 访问链表的不同部分的线程可能相互阻塞

2、Optimistic List

1. 特点：
    1. 优点：
        1. 无干扰，但依赖于垃圾回收来保证正在被遍历的节点不能重用
        2. 是无死锁的
        3. 不用锁遍历两次链表的代价比使用锁遍历一次链表的代价小需多时效果好
    2. 缺点：
        1. 不是无饥饿的：如果不断添加或删除节点，那么一个线程将会被永远阻塞
        2. contains()方法需要获得锁，并且实际上contains方法调用更频繁

3、Lazy List算法

1. 特点
    1. 优点：
        1. contains()无等待，add()和remove()仅需遍历一次链表
    2. 缺点
        1. add()和remove()不是无饥饿的
        2. add()和remove()是阻塞的，如果一个线程延迟，那么其他线程也将延迟

4、Lock-Free List算法

1. 特点
    1. 优点：
        1. add()，remove()无锁，contains()无等待
        2. 保证面对任意的延迟的时候，线程可以继续演进
    2. 缺点：

1. 强演进保证需要额外代价：对引用和布尔标记的原子修改需要额外的性能损耗
2. add()和remove()遍历链表的时候需要参与对已删除的结点的并发清理，从而导致线程之间发生争用

三、Queue算法

1. 池Pool：只提供get和set方法
    1. 可以有界或无界
    2. 可以是完全、部分或同步的
        1. 完全：一个方法的调用不需要等待某个条件成立（non-blocking）
            1. 空池删除元素的get()调用、满的有界池添加元素的set()调用将立刻返回错误码或异常
        2. 部分：一个方法的调用需要等待某个条件成立（blocking）
            1. 空池删除元素的get()调用会阻塞，直到池中有可用元素才返回
            2. 满的有界池的添加元素的set()调用会阻塞，直到池中有一个可用的空槽
        3. 同步：一个方法需要等待另一个方法与它的调用间隔重叠
            1. 同步池中向池中添加元素的方法调用将被阻塞直到该增加的元素被另一个方法调用取走
            2. 同步池中删除元素的方法调用将被阻塞直到另一个方法调用添加了这个可用于删除的元素
    3. 提供公平性保证，如先进先出、后进后出等

1、Two-Lock Queue（Bounded, Blocking）

1. 算法

```
public class BoundedQueue<T> {
    ReentrantLock enqLock, deqLock;
    Condition notEmptyCondition, notFullCondition;
    AtomicInteger permits;//0 to capacity
    Node head;
    Node tail;
    int capacity;
    enqLock = new ReentrantLock();
    notFullCondition = enqLock.newCondition();
    deqLock = new ReentrantLock();
    notEmptyCondition = deqLock.newCondition();
    public BoundedQueue(int _capacity){
        ...
    }
    public void enq(T x) {
        boolean mustWakeDequeuers = false;
        enqLock.lock();
        try {
            while (permits.get() == 0)//看一看队列是否满，满就await，不满就可以入队
                notFullCondition.await();//注意这里permits可能会变，但我们持有enqLock，所以permits只会变大不会变小，因此await是没有问题的
            Node e = new Node(x);
            tail.next = e;
            tail = e;
            if (permits.getAndDecrement() == capacity)//看一看队列是否空，空就要通知所有deque线程
```

```
                mustWakeDequeuers = true;
            } finally {
                enqLock.unlock();
            }
            if (mustWakeDequeuers) {//如果要通知所有deque线程则通知
                deqLock.lock();
                try {
                    notEmptyCondition.signalAll();//应该使用signalAll
                } finally {
                    deqLock.unlock();
                }
            }
        }
        public T deq() {
            T result;
            boolean mustWakeEnqueuers = false;
            deqLock.lock();
            try {
                while (permits.get() == capacity)
                    notEmptyCondition.await();//这里permits也会变，但实现正确，
证明同enq理
                result = head.next.value;
                head = head.next;
                if (permits.getAndIncrement() == 0)
                    mustWakeEnqueuers = true;
            } finally {
                deqLock.unlock();
            }
            if (mustWakeEnqueuers) {
                enqLock.lock();
                try {
                    notFullCondition.signalAll();
                } finally {
                    enqLock.unlock();
                }
            }
            return result;
        }
}

```

2. 特点
    1. 优点：
        1. 没有唤醒丢失问题
    2. 缺点：
        1. 并发的enq()和deq()相互干扰，但又不通过锁。所有方法对size域调用getAndIncrement()和getAndDecrement()。这种方法比通常的读写开销大，且能引起顺序瓶颈
            1. 解决方案：切分size为两个计数器，分别负责enqSize和deqSize
3. 注意：
    1. 唤醒丢失问题：考虑如下问题
        1. 若唤醒采用仅当队列实际上从空变为非空才给条件发出信号的方式：消费者A、B都希望在一个空队列中出队元素，他们检测到队列空，于是在nonEmpty条件上阻塞。生产者C将缓冲区一个数据项入队，给nonEmpty发信号，唤醒A，然而在A获得锁之前，另一个

生产者D把第二个数据项放入队列中。由于队列非空，所以它不对nonEmpty产生信号。于是A获得锁，移走第一个数据项，B却要永远等待，即使缓冲区有一个等待消费的数据项。

但是本算法不会产生唤醒丢失问题，因为signalAll了

2、Lock-Free Queue（Unbounded, Non-Blocking)

1. 算法

```
public void enq(T x) {
    Node e = new Node(x);
    while (true) {
        Node t = tail.get();
        Node n = t.next.get();
        if (t == tail.get()) {//这个判断不是必要的
            if (n == null) {//说明是真正的尾部
                if (t.next.compareAndSet(n, e)) {//一旦逻辑连接成功，就最终一定会被连接进来，所以这里是enq的可线性化点
                    tail.compareAndSet(t, e);
                    return;
                }
            } else {
                tail.compareAndSet(t, n);//帮助其它线程连接了尾部
            }
        }
    }
}
public T deq() throws EmptyException{
    while (true) {
        Node h = head.get();
        Node n = h.next.get();
        Node t = tail.get();
        if (h == head.get()) {
            if (h == t) {//判断需不需要
                if (n == null)
                    throws new EmptyException();//失败的时候的可线性化点
                tail.compareAndSet(t, n);//帮助其它线程设置tail
            } else {
                T result = n.value;
                if (head.compareAndSet(h, n)) //成功的时候的可线性化点，真正连接进来了，如果有人修改head就返回false，重新从while循环来过
                    return result;//没人修改head，成功返回
            }
        }
    }
}
```

2. 特点

    1. 无锁，效率高

3. ABA问题解决：compareAndSet同时比较引用和时间戳，并且每次使用compareAndSet更新引用的时候都会把时间戳+1。算法整体就是照搬LockFree Queue的，改了compareAndSet而已。

```
public T deq() throws EmptyException{
    int[1] hStamp, nStamp, tStamp;
    while (true) {
```

```
        Node h = head.get(hStamp);
        Node n = h.next.get(nStamp);
        Node t = tail.get(tStamp);
        if (h == t) {//如果头等于尾，那么需要确定尾需不需要帮忙往后
            if (n == null) //确实为空就抛出异常
                throw new EmptyException();
            tail.compareAndSet(t, n, tStamp[0], tStamp[0] + 1);//需要帮忙就把尾往后移动，然后进入下一轮循环
        } else {
            T result = n.value;
            if (head.compareAndSet(h, n, hStamp[0], hStamp[0] + 1)) {//
                free(h);
                return result;
            }
        }
    }
}
```

四、Stack算法

1、Lock-Free Stack（Exponential Backoff)

1. 算法

```
public class LockFreeStack {
    private AtomicReference top = new AtomicReference(null);
    public boolean tryPush(Node node){
        Node oldTop = top.get();
        node.next = oldTop;
        return(top.compareAndSet(oldTop, node))
    }
    public void push(T value) {
        Node node = new Node(value);
        while (true) {
            if (tryPush(node)) {//tryPush成功就返回，失败则指数后退
                return;
            }
            else
                backoff.backoff();
        }
    }
    public Node tryPop() throws EmptyException{
        Node oldTop = top.get();
        if(oldTop == null){
            throw new EmptyException();
        }
        Node newTop = oldTop.next;
        if(top.compareAndSet(oldTop, newTop)){
            return oldTop;
        } else {
            return null;
        }
    }
    public T pop() throws EmptyException{
        while(true){
            Node returnNode = tryPop();
            if(returnNode != null) {
```

```
                return returnNode.value;
            } else {
                backoff.backoff();
            }
        }
    }
}
```

2. 特点
    1. 优点：无锁
    2. 可线性化点：push和pop方法都是成功的compareAndSet调用，或对空栈调用pop()方法抛出异常的时刻
    3. pop()的compareAndSet不会出现ABA问题，是因为Java的垃圾回收器能确保只要一个结点对另一个结点可达则不会被同一个线程重用
        1. 如果没有垃圾回收，怎么设计避免ABA问题的无锁栈?
    4. 缺点：
        1. 害怕ABA问题
        2. 栈的top域是一个争用源
        3. 这种栈是一个顺序瓶颈：方法调用只能按照对top域的compareAndSet()的成功调用次序来排序，任何情况都无法并行

2、Elimination Backoff Stack

1. 无锁交换机的Exchanger算法

```
public T Exchange(T myItem, long nanos) throws TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {EMPTY};//储存timestamp的数组
    while (true) {
        if (System.nanoTime() > timeBound)//循环直到超时
            throw new TimeoutException();
        T herItem = slot.get(stampHolder);//获得他人的item
        int stamp = stampHolder[0];//获得他人的stampHolder
        switch(stamp) {
            case EMPTY:// slot is free
                if (slot.compareAndSet(herItem, myItem, EMPTY, WAITING))
{
                    // slot空闲，尝试插入我的item，并设置成WAITING，否则重试
                    while (System.nanoTime() < timeBound){//循环直到超时
                        herItem = slot.get(stampHolder);
                        if (stampHolder[0] == BUSY) {//他人设置state成
BUSY了吗
                            slot.set(null, EMPTY);//成功了就重设slot回null
和EMPTY
                            return herItem;
                        }
                    }
                    //我们超时了，重设我们回EMPTY并抛出Timeout异常，要看看是不是
真的没希望了
                    if (slot.compareAndSet(myItem, null, WAITING,
EMPTY))
                        throw new TimeoutException();
                    herItem = slot.get(stampHolder);//如果失败，意味着还是
有人来了!
```

```
                    slot.set(null, EMPTY);//成功了就重设slot回null和EMPTY
                    return herItem;
                }
                break;//retry
            case WAITING:// someone waiting for me
                //意味着有人在等我们，所以尝试把自己的数据项CAS进去，并设置WATING为BUSY
                if (slot.compareAndSet(herItem, myItem, WAITING, BUSY))
                    return herItem;//成功就返回别人的item，否则有人拿走了她，就重试
                break;
            case BUSY: // others in middle of exchanging
                break;
            default: // impossible
                break;
        }
    }
}
```

1. 不存在ABA问题，因为改变状态的compareAndSet()调用绝不会检查数据项。
2. 可线性化点：
    1. 成功的交换：第二个到达的线程将状态从WAITING改为BUSY的时刻
    2. 不成功交换：抛出超时异常 的时刻
3. 无锁：因为只有当其它交换不断成功（例如太短的交换时间、或者相互重叠的交换时间太长等）或没人与我配对的时候，才会失败。

2. 消除数组的Elimination Array算法

```
public T visit(T value, int Range) throws TimeoutException{
    int slot = random.nextInt(Range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur));
}
```

3. Elimination Backoff Stack算法：

```
public void push(T value) {
    ...
    while (true) {
        if (tryPush(node)) {//先尝试push，成功就返回
            return;
        } else {
            try {//push失败了，不再指数后退，而是改用Elimination的方法
                T otherValue = eliminationArray.visit(value,policy.Range);//压栈的值和准备尝试的range
                if (otherValue == null) {
                    return;//pop成功匹配
                }
            } catch (TimeoutException e) {
                policy.recordEliminationTimeout();
            }
        }
    }
}
public T pop() {
    ...
```

```
    while (true) {
        if (tryPop()) {
            return returnNode.value;
        } else{
            try {
                T otherValue =
eliminationArray.visit(null,policy.Range);
                if (otherValue != null) {
                    return otherValue;
                }
            } catch (TimeoutException e) {
                policy.recordEliminationTimeout();
            }
        }
    }
}
```

4. 特点
    1. 优点：
        1. 无锁
        2. 可线性化（因为共享栈原来就是可线性化的，并且可以直接排序被消除的push、pop调用）
            1. 任何访问LockFreeStack成功返回的push()和pop()调用都可以在访问LockFreeStack时被线性化
            2. 每一对被消除的push()和pop()调用可以在他们冲突时被线性化
        3. 负载增加的时候，成功消除的个数会变大，从而允许很多操作并行执行
        4. 因为被消除的操作不会访问栈，所以减少了在栈上对top的争用，减少了线程访问无锁栈的机会