Chapter2:

10. 互斥是通过每个线程看到的各自的view得到关于global的关于critical area的owner的一致看法实现的。根据2.8的证明，锁的实现必须有写的动作，如果第一条指令是读，且只依据这一条指令是不能区分先后的；如果写了之后没有读，线程不能得到view，和没写一样；如果又写又读，并得到某些顺序则它实际就是个gate。

11. 满足互斥。假设不成立。假设 CS(A)-->CS(B) => R(A)(turn=A) --> R(B)(turn=B) && W(A)(turn=A)-->W(B)(turn=B) && R(A)(turn=A)-->W(B)(turn=B)；否则turn由B改变后不能再变成A。所以有 W(A)(busy=true)-->R(A)(turn=A)-->W(B)(turn=B)-->R(B)(busy=false) => W(A)(busy=true)->R(B)(busy=false). 矛盾。

不满足无饥饿，因为某个线程A执行完turn=A之后，等待busy = false的时候，别的线程可能无限次的turn=X-->busy==false-->busy=true。

不满足无死锁。可能有 W(A)(turn=A)-->W(B)(turn=B)-->R(A)(busy=false)-->W(A)(busy=true)-->R(B)(busy=false). A waits turn==A, B waits busy == false.

12. 假设线程A在第k层停住，只要有>2(除A外)个线程进入第k层，到得早的线程必然有 victim != me。

13. 用归纳法。假设k层的这种锁满足互斥，当树高增加到k+1层时，因为Peterson锁满足互斥，使得k+1层上的线程只能在每个k层节点上推举出一个线程，结果是构成一个k层的树，根据假设它满足互斥。同理它也无饥饿。

满足无死锁，因为获得锁的过程是二叉树从叶子节点到跟节点的方向，整个图不能找出任一个有向环，不满足死锁的条件。

上界为n。首先上界>=n(包括自己这一次)，否则不满足互斥。假设h=k(根节点h=0)时上界为n=(2^k)，当叶节点为2n，h=k+1时，某个叶节点A可能兄弟节点赢，兄弟节点在h=k层上等2^k次加解锁；根据Peterson锁的性质，下次兄弟节点参与竞争是必然是A赢，则A在h=k层上再等待2^k次；即2^(k+1)次，必然得到锁。或者这样考虑，如果一个线程A抢锁成功，则它下一次再参与的时候必然比其它在它第二次抢锁开始之前参与的线程B（包括第一次抢锁）要后得到锁。因为叶结点为n，所以最多有n个线程在它之前。同时，放锁之后下一次抢锁必然输给A。所以界是n。

14. 将filter锁减少l层。根据section 2.4的证明第l层的过程，得知第l层满足l-Exclusion和l-Starvation-Freedom的性质。

15. 不满足互斥。比如2个线程以这个顺序执行lock：W(A)(x=A)-->W(B)(x=B)-->R(A)(y==-1)-->R(B)(y==-1)-->W(A)(y=A)-->W(B)(y=B)-->R(B)(x==B)-->CS(B)-->R(A)(x!=A)-->（A）(lock.lock())-->CS(A)。因为B进入CS时没有经过lock => lock不知道还有一个竞争者 => lock同意A进入CS。而且因为B没有经过lock，它在执行unlock-->lock.unlock()的时候也可能会出问题。

16. 因为W(A)(last = A) -->R(A)(goRight == false)-->W(A)(goRIght = true) --> R(A)(last == A); W(B)(last = B) -->R(B)(goRight == false)-->W(B)(goRIght = true) --> R(B)(last == B)；假设A先，有 W(A)(last = A) --> R(A)(last == A)-->W(B)(last = B) --> R(B)(last == B)，也有 W(A)(goRight = true) --> R(A)(last == A) --> W(B)(last = B)--> R(B)(goRight == false)。矛盾。这个过程类似于11题。所以最多只有一个线程获得STOP。

因为任意的线程X都有W(X)(last = X)-->R(X)(last != X)，即对所有的线程X都存在Y使得W(X)(last = X) --> W(Y)(last = Y)。因为在W(X)(last = X)这个点上可以将所有线程按照时间排成完全有序的序列，则随后一个线程找不到比它后来的线程，所以最多只有n-1个得到DOWN。

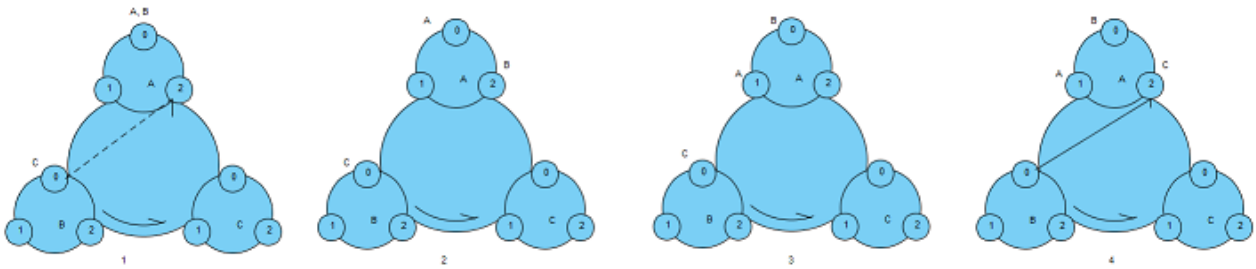
因为有R(X)(goRIght == true) --> RIGHT(X)，而且goRight初始值为false，必然有线程Y: R(Y)(goRight == false) --> W(Y)(goRight = true)。即R(Y)(goRight == false) --> W(Y)(goRIght = true) --> STOP(Y) || DOWN(Y)。所以最多只有一个n-1个线程得到RIGHT。

17. 因为每个Bouncer对象都最多有n-1个线程得到RIGHT，最多n-1个线程得到DOWN，即数组中任意Bouncer对象的右边对象和下面的对象都要比该Bouncer对象少至少一个竞争者。所以线程沿着Bouncer的计算结果移动时，或者得到STOP，或者是剩下的最后一个参与者，也得到STOP。所以必然停在某一个Bouncer。

因为每一步最多有n-1个遗留的线程，但不能确定是往DOWN还是RIGHT，所以得到的布局是如图2-18所示n * n的三角形。

18. 如图所示：

- A, B在A0上，C在B0上。C观察后准备移到A2，以决定A，B。然后C休眠。
- B移到A2上
- A移到A1上
- A，B在圈A上转，直到A在A1，B在A0。C醒来，转移到A2。A上形成了一个circle。

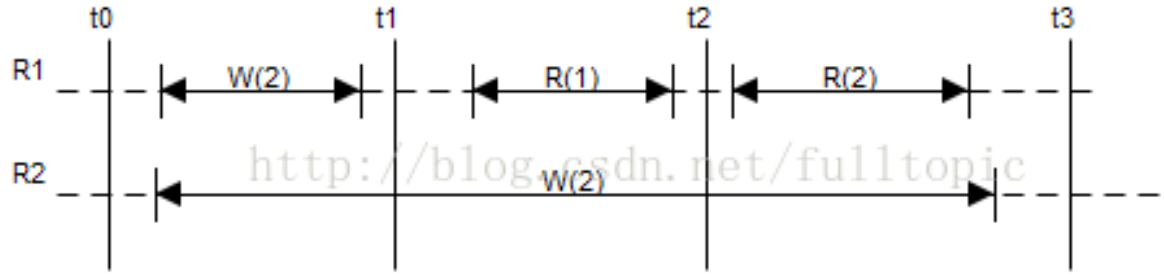


19. 一个n位2进制的每一位代表一个参与者，1为active，0为inactive。需要一个新label时，假设为线程j，扫描其他n-1个n为2进制数，如果第i个数的第i位为1，表示i为active，并将第j个2进制数的第i位置1，最后将自己的第j位置1。j退出时要将自己的第j位置0，并将其他数的第j位置0。

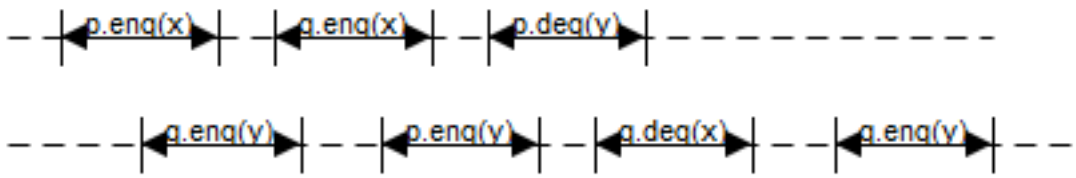
Chapter3

21 A，B复合之后的任一静止状态必然同时是A和B的静止状态，则由这个静止状态分开的任何方法都满足 原理3.3.2

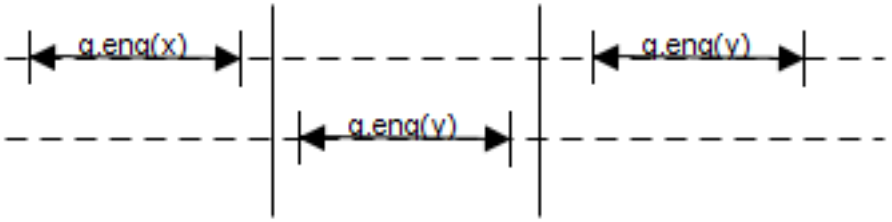
22. 否。如图示: R1一个时间单位返回,但是效果需要3个时间单位才能被看到; R2需要3个时钟单位返回，并且立刻能被看到。对于组合，t3为静止态，读操作能得到合理的结果；但是对于R1，t1为静止态，不满足静态一致性。



23. 静态一致非顺序一致：



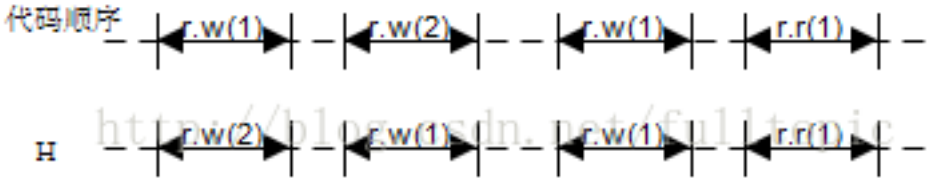
顺序一致非静态一致：



24.1 是静态一致: r.write(2) --> r.read(2)。 是可线性化：r.w(1) --> r.r(1) --> r.w(2) --> r.r(2)。所以也是顺序一致的。

24.2 是静态一致: r.write(1) --> r.read(1)。 是可线性化：r.w(2) --> r.w(1) --> r.r(1) --> r.r(1)。 所以也是顺序一致的。

25. 因为合法不代表遵循单线程中的代码顺序，所以去掉L2不能保证顺序一致。比如：



26. 假设有 H | x不可线性化， 必然违反定义3.6.1中的一条。因为S是响应紧邻调用的事件组合，对于H等价的S去掉所有其他H| y对应的S | y的事件后，剩下的事件必然构成一个H | x的顺序经历，所以不可能违反L1。如果违反L2，所有能找到的S(x)都至少有2个事件m2 --> m1但是在H | x中有m1 --> m2；对于满足L2的S，在m2与m1之间插入任意的事件都不能改变这个事实，即如果没有S | x满足L2，也不可能有S满足L2。

27. 见参考答案。

28. 根据我所知道的c的内存模型，如果代码进行了优化，可能会出现volatile的v比非volatile的x先在多核之间同步的情况，即y可能被0除；不清楚java的模型中happens before是不是消除了这种情况。

29. 假如完成了无限个方法调用执行了无限的平方个操作步，每个调用仍然执行了无限个操作步，不是无锁的。

30. 如果x中某一个方法调用不是无锁的，即进行了无限次(n， n趋近于无限)操作，则任意选取只包含了这个调用的H，例如， |H| = O(2n)，n趋近无限大。则H是个无限经历，但只有2个操作完成。矛盾。所以是无锁的。
31. 如果i有限则2^i有限，所以无等待。但是2^i不收敛，不是有界无等待。

Chapter4

42. 因为只有一个write重叠，从后往前读。
case 1: b2 == b1，write已经更新完了；返回b2。
case 2: b2 == b1，write正在更新b0；返回b2。这时write可能已经更新了b[2*N + i]的值，但是原来的值已经记录在b2中，而且是上一次更新的值。
case 2: b2 != b1，write正在更新b1；返回b0。因为只有一个write重叠，所以更新过的b0已经是一个有效值，而且read返回前不会改变。
case 4: b2 != b1，write正在更新b2；返回b0。
从b2到b0读的关键在于，在整个读的过程中只会读到(b2, b1, b0中的）最多一个正在被改变的值。
所以不能从b0往b2读。假如判断是 return (b0 == b1)? b0: b2; 则可能读b2值的时候write正在更新而且没有完成，会返回一个不安全的值。

```
1  1 class AcmeRegister implements Register{
2  2 // N is the total number of threads
3  3 // Atomic multi-reader single-writer registers
4  4 private BoolRegister[] b = new BoolMRSWRegister[3 * N];
5  5 public void write(int x) {
6  6 boolean[] v = intToBooleanArray(x);
7  7 // copy v[i] to b[i] in ascending order of i
8  8 for (int i = 0; i < N; i++)
9  9 b[i].write(v[i]);
10 10 // copy v[i] to b[N+i] in ascending order of i
11 11 for (int i = 0; i < N; i++)
12 12 b[N+i].write(v[i]);
13 13 // copy v[i] to b[2N+i] in ascending order of i
14 14 for (int i = 0; i < N; i++)
15 15 b[(2*N)+i].write(v[i]);
16 16 }
17 17 public int read() {
18 18  int b2 = booleanArrayToInt(b[2*N], N); //psudo codes
19 19  int b1 = booleanArrayToInt(b[N], N);
20 20  int b0 = booleanArrayToInt(b[0], N);
21 21
22 22  return (b2 == b1)? b2: b0;
23 23 }
24 24 }
```

43. True
44. Obstruction-Free 实现:
根据题中定理4.5.3， read()中的c0 >= cl; c1 <= cl; 因为 (c0 == c1) ==> (c0 <= cl && c0 >= cl) == > c0 == cl, 即这是一个规则的实现。根据Note：
If a read of c obtains traces of version cj, j >= 0, then:
The beginning of the read preceded the end of write c(j + 1).
The end of read followed the beginning of the write of cj;
所以如果A不可能读到未来的值，也不会读到过期的值；如果有A.read -> C.read。C只能读到c(j+m)(m>=0)。所以这是一个swmr的原子实现。

[java] view plain copy

```
1  class Counter
2  {
3    private boolean c[2][M];
```

```
    private int convert(boolean[] c)
    {
      //convert c1c2...cm into counter;
      return 0;
    }

    private boolean[] convert(int v)
    {
      //convert counter value into c array
      return boolean[2];
    }

    public void update(int v)
    {
      boolean[] cv = convert(v);
      for(int i = 0; i < M; i ++)
      {
        c[0][i] = cv[i];
      }
      for(int i = M - 1; i >= 0; i --)
      {
        c[1][i] = cv[i];
      }

      return;
    }

    public int scan()
    {
      boolean  b[2][M];

      do
      {
        for(int i = M - 1; i >=0; i --)
        {
          b[0][i] = c[0][i];
        }
        for(int i = 0; i < M; i ++)
        {
          b[1][i] = c[1][i];
        }
      }while(!Arrays.equals(b[0],b[1]);

      return convert(b[0]);
    }
  }
```

[java] view plain copy

```
1
```

Wait-Free 实现

在obstruction-free的证明的基础上。

根据section 4.3中的证明，如果A观察到B udpate了2次，则可以用B的scan作为A scan的结果。因为counter值一直增加，所以如果B保存的值 > A collect一次的值，则B有update，因为根据这个实现，B只有update时才会更新它保存的scan值。

[java] view plain copy

```
1   import java.util.*;
2
3   public abstract class Counter
4   {
5       public static final int SIZE = 8;
6       public static final int N = 8;
7       public ThreadLocal<Integer> ThreadId;
8
9       private int[] c1 = new int[SIZE];
10      private int[] c2 = new int[SIZE];
11
12      private int[] cValueTable = new int[N];
13
14      public Counter()
15      {
16      }
17
18      private int[] convert(int value)
19      {
20          return new int[SIZE];
21      }
22
23      private int convert(int[] cValue)
24      {
25          return 0;
26      }
27
28      public int scan()
29      {
30          int[] scanC1 = new int[SIZE];
31          int[] scanC2 = new int[SIZE];
32          boolean[] moved = new boolean[N];
33
34          while(true)
35          {
36              for(int i = 0; i < c1.length; i ++)
37              {
38                  scanC1[i] = c1[i];
39              }
40              for(int i = c2.length; i >= 0; i --)
41              {
42                  scanC2[i] = c2[i];
43              }
44
45              int value = convert(scanC1);
46              if(Arrays.equals(scanC1, scanC2))
47              {
48                  return value;
49              }else
50              {
51                  for(int i = 0; i < N; i ++)
52                  {
53                      if(cValueTable[i] > value && moved[i])
54                      {
55                          return cValueTable[i];
56                      }else
```

```
57                  {
58                      moved[i] =  true;
59                  }
60              }
61          }
62      }
63  }
64
65
66  public void update(int value)
67  {
68      int me = ThreadId.get().intValue();
69
70      int[] cv = convert(value);
71      for(int i = c1.length; i >=0; i --)
72      {
73          c1[i] = cv[i];
74      }
75
76      for(int i = 0; i < c2.length; i ++)
77      {
78          c2[i] = cv[i];
79      }
80
81      cValueTable[me] = value;
82  }
83  }
```

45. (1). 因为有W(C2(k)) --> W(C1(k)) --> R(C1(k)) --> R(C2(k)) ==> 即对任意k，读过c1再读c2，得到的c2版本号一定 >= c1的版本号。所以有 l1 < k2。
(2). 令 c(k, l) = c(i1)c(i2)...c(im)。根据4.5.1，有i1 <= i2 <= ... <= im <= l；根据4.5.2有 c(i1)c(i2)...c(im) <= cl。

Chapter5

47. 如果n个线程刚好可以分成2组, 第一组的行为i与lemma5.1的行为一样，第二组的行为与Lemma5.1的行为一样，则它们有一个2值初始状态。
49. 因为critical state必须是bivalent，所以2个后续只能 1-valent + 0-valent或者 x + bivalent。
因为if any thread moves ,the protocol has a critical state，所以后续只能是univalent。所以只能是 1-valent + 0-valent。
50.将 n个线程分成2组，每组的线程都完全同步并行，并且行为与2线程protocol的2个线程完全一样；则如果2个线程不能decide，n个线程也不能。
51. 将k值与2值做映射，比如
[java] view plain copy

```
1   if(k >= MAXK / 2)
2   {
3       k = 1;
4   }else
5   {
6       k = 0;
7   }
```

则n线程decide on k后再映射成2值。

52.
以58题答案为例
53.

consensus >= 2:

```
1  package jokes;
2
3
4  import java.util.Stack;
5
6  public class StackProtocol
7  {
8      private int[] proposes;
9      static ThreadLocal<Integer> ThreadId = new ThreadLocal<Integer>()
10     {
11         protected Integer initialValue()
12         {
13             return new Integer(0);
14         }
15     };
16     static final int WIN = 1;
17     static final int LOSE = 2;
18     private Stack<Integer> stack;
19
20     public StackProtocol()
21     {
22         proposes = new int[2];
23         stack = new Stack<Integer>();
24         stack.push(LOSE);
25         stack.push(WIN);
26     }
27
28     private void propose(int v)
29     {
30         proposes[ThreadId.get()] = v;
31     }
32
33     public int decide(int v)
34     {
35         propose(v);
36         if(stack.pop() == WIN)
37         {
38             return proposes[ThreadId.get()];
39         }else
40         {
41             return proposes[1 - ThreadId.get()];
42         }
43     }
44 }
```

consensus < 3:
Both A & B pop(): 不论A先或B先，2个pop()结束后，C solo的时候无法区分谁是winner
A push & B pop: A.push() --> B.pop() 与AB都没有动作不能区分； B.pop() --> A.push() 与只有A.push()无法区分。
A push & B push: 2种顺序Stack状态一样，无法区分。
54.
如下所示：

[java] view plain copy

```
1  package jokes;
2
3  import java.util.concurrent.LinkedTransferQueue;
4
5  public class QueuePeekConsensus
6  {
7      private LinkedTransferQueue<Integer> q;
8
9      public QueuePeekConsensus(int n)
10     {
11         q = new LinkedTransferQueue<Integer>();
12     }
13
14     private void propose(int v)
15     {
16         q.put(v);
17     }
18
19     public int decide(int v)
20     {
21
22         propose(v);
23         return q.peek();
24     }
25 }
```

55.

不能。因为R(AB)只能被A和B访问，即只有2个现成参与。可以将CompareAndSet用getAndIncrement实现。则这个协议是由Common2寄存器和原子寄存器构成的，一致数为2。

一个反例：

A.CAS(RAB) --> B.CAS(RAB) --> B.CAS(RBC) --> C.CAS(RBC) --> C.CAS(RAB) --> A.CAS(RBC)

==> A-->B-->C-->A

56.

如下所示：

[java] view plain copy

```
1  package jokes;
2
3  import java.util.concurrent.atomic.*;
4
5  public class CAS32Protocol
6  {
7      public final int THREADNUM = 3;
8      public final int THREADA = 0;
9      public final int THREADB = 1;
10     public final int THREADC = 2;
11     public final int INITIAL = -1;
12
13     private AtomicInteger[] casObjs;
14
15     private int[] proposes;
16     public CAS32Protocol()
17     {
18         proposes = new int[THREADNUM];
19         casObjs = new AtomicInteger[THREADNUM]; //ab, bc, ac
```

```
20          for(int i = 0; i < THREADNUM; i ++)
21          {
22              casObjs[i] = new AtomicInteger(INITIAL);
23          }
24      }
25
26
27      private synchronized boolean  propose(int index, int exp1, int exp2, int v1, int v2)
28      {
29          int a = index;
30          int c = (index + 2) % THREADNUM;
31
32          if(casObjs[a].compareAndSet(exp1, v1)
33                  && casObjs[c].compareAndSet(exp2, v2))
34          {
35              return true;
36          }
37
38          return false;
39      }
40
41      //It dumps when champion and runner-up both halted
42      public int decide(int threadId, int v)
43      {
44          int a = threadId;
45          int b = (threadId + 1) % 3;
46          int c = (threadId + 2) % 3;
47
48          proposes[a] = v;
49
50          if(propose(a, INITIAL, INITIAL, THREADA, THREADA))
51          {
52              proposes[b] = proposes[c] = proposes[a];
53              //A wins
54          }else if(propose(a, THREADB, INITIAL, THREADB, THREADA))
55          {
56              // B-->A-->C
57              proposes[a] = proposes[b];
58          }else if(propose(a, INITIAL, THREADC, THREADA, THREADC))
59          {
60              //C-->A-->B
61              proposes[a] = proposes[c];
62          }else
63          {
64              //Waiting for winner
65              while(proposes[b] != proposes[c]);
66              proposes[a] = proposes[b];
67          }
68
69          return proposes[a];
70      }
71  }
```

57.

只要能够在loser知道自己是loser之前就能获得announce的值，就有infinite consensus

58.
1)如下所示：

[java] view plain copy

```
package jokes;

import java.util.concurrent.atomic.*;
public class Sticky
{
    public enum StickyValueEnum
    {
        INIT(-1),
        ZERO(0),
        ONE(1);

        private int v;
        private StickyValueEnum(int v)
        {
            this.v = v;
        }

        public int get()
        {
            return v;
        }
    }

    private AtomicInteger v;

    public Sticky()
    {
        v = new AtomicInteger(StickyValueEnum.INIT.get());
    }

    public boolean write(StickyValueEnum v)
    {
        return this.v.compareAndSet(StickyValueEnum.INIT.get(), v.get());
    }

    public int read()
    {
        return v.get();
    }

}
```

[java] view plain copy

```
package jokes;

import jokes.Sticky;
import jokes.Sticky.StickyValueEnum;


public class BinarySticky
{
```

```
    private Sticky sticky;

    public BinarySticky()
    {
        sticky = new Sticky();
    }

    public int decide(StickyValueEnum v)
    {
        sticky.write(v);

        return sticky.read();
    }
}
```

2)

[java] view plain copy

```
package jokes;

import jokes.Sticky;
import jokes.Sticky.StickyValueEnum;

//import java.nio.ByteBuffer;
import java.util.concurrent.atomic.*;
import java.util.BitSet;

public class MValentSticky
{
    private static ThreadLocal<Integer> ThreadId = new ThreadLocal<Integer>()
    {
        protected Integer initialValue()
        {
            return new Integer(0);
        }
    };
    private AtomicInteger[] proposes;
    private Sticky[] stickys;

    private final int stickyNum;
    private final int threadNum;

    public MValentSticky(int vNum,  int threadNum)
    {
        this.threadNum = threadNum;
        proposes = new AtomicInteger[this.threadNum];
        for(int i = 0; i < threadNum; i ++)
        {
            proposes[i] = new AtomicInteger(Sticky.StickyValueEnum.INIT.get());
        }

        stickyNum = (int)Math.ceil(Math.log(vNum) / Math.log(2));
        stickys = new Sticky[stickyNum];

        for(int i = 0; i < stickyNum; i ++)
```

```
38          {
39              stickys[i] = new Sticky();
40          }
41      }
42
43      private static StickyValueEnum GetStickyValue(boolean bit)
44      {
45          if(bit)
46          {
47              return StickyValueEnum.ONE;
48          }else
49          {
50              return StickyValueEnum.ZERO;
51          }
52      }
53
54      private BitSet getBitSet(int v)
55      {
56
57          BitSet bits = new BitSet(stickyNum);
58
59          int i = 0;
60          while(v != 0)
61          {
62              if(v % 2 == 1)
63              {
64                  bits.set(i);
65              }
66              i ++;
67              v >>= 1;
68          }
69
70          return bits;
71      }
72
73      //maxBitIndex: first bit not matched
74      private BitSet foundMatchWinner(BitSet me, int maxBitIndex)
75      {
76          if(maxBitIndex >= stickyNum)
77          {
78              return me;
79          }
80
81          //In same order, losers may find same winner
82          //While it is not important
83          for(int i = 0; i < threadNum; i ++)
84          {
85              if(i == ThreadId.get())
86              {
87                  continue;
88              }
89
90              BitSet other = getBitSet(proposes[i].get());
91              int j = 0;
92              while(j < maxBitIndex && me.get(j) == other.get(j));
93
```

```
94              if(j >= maxBitIndex)
95              {
96                  return other;
97              }
98          }
99
100         return me;
101     }
102
103     private int convert(BitSet bits)
104     {
105         int v = 0;
106         for(int i = bits.length(); i >= 0; i --)
107         {
108             v <<= 1;
109             if(bits.get(i))
110             {
111                 v |= 1;
112             }
113         }
114
115         return v;
116     }
117
118     public int decide(int proposeV)
119     {
120         int v = proposeV;
121         int index = ThreadId.get();
122         proposes[index].set(v);
123
124         BitSet bits = getBitSet(v);
125
126
127         for(int i = 0; i < stickyNum; i ++)
128         {
129             if((stickys[i].write(GetStickyValue(bits.get(i)))))
130             {
131                 continue;
132             }else
133             {
134                 bits = foundMatchWinner(bits, i);
135                 //Not necessary to update proposes[]
136                 //as candidates can get value from sticky
137             }
138         }
139
140         return convert(bits);
141     }
142
143     public static void main(String[] args)
144     {
145         MValentSticky toy = new MValentSticky(4, 2);
146         int rc = toy.decide(3);
147         System.out.println("End of test " + rc);
148     }
149 }
```

59.
一个用原子寄存器实现的SetAgree(2)如下所示。因为它可以用AtomicRegister实现，所以一致数为1。
但是当 k < n (线程数), consensus number = infinite。

[java] view plain copy

```
1  package jokes;
2
3  import java.util.concurrent.atomic.*;
4
5  public class SetAgree2
6  {
7      private AtomicBoolean[] proposes;
8
9      public final int ThreadNum;
10
11     public SetAgree2(int threadNum)
12     {
13         this.ThreadNum = threadNum;
14         proposes = new AtomicBoolean[this.ThreadNum];
15         for(int i = 0; i < this.ThreadNum; i ++)
16         {
17             proposes[i] = new AtomicBoolean();
18         }
19     }
20
21     public void propose(boolean v, int threadId)
22     {
23         if(threadId < ThreadNum && threadId >= 0)
24         {
25             proposes[threadId].set(v);
26         }
27     }
28
29     public boolean decide(int threadId)
30     {
31         if(threadId < ThreadNum && threadId >= 0)
32         {
33             return proposes[threadId].get();
34         }
35
36         return false;
37     }
38
39 }
```

60.
1
61
A类：不能。将收到消息看成对一个对象进行读操作的返回。因为这个对象之提供读和写操作，即相当于原子寄存器。
则A类型的实现相当于对一组原子寄存器进行读写实现的，所以consensus number =1
B类：类似于访问一个提供peek方法的queue，consensus number = infinite

62.
如果A.propose == 1， A decides 1； B可以随意决定都是正确的。

如果A.propose == 0 而且B还没有propose，则A可以决定一个合理的值，B 在决定的时候因为可以看到A，所以可以配合这个决定。
因为如果B后来propose了0则A必须决定0，所以这种情况下A决0。
如果A decides时候B已经有了propose，即B决定在A之前。如果B.propose == 0，则A必须决定0，否则A必须决定1，因为B已经决定了1。

[java] view plain copy

```
1   package jokes;
2
3   public class QuasiConsensus
4   {
5     public static final int THREADNUM = 2;
6
7     int[] proposes;
8
9     public QuasiConsensus()
10    {
11      proposes = new int[THREADNUM];
12      for(int i = 0; i < THREADNUM;i ++)
13      {
14        proposes[i] = -1;
15      }
16    }
17
18    //threadId: A--> true, B-->false
19    public int decide(boolean threadId, int v)
20    {
21      if(threadId) // thread A
22      {
23        proposes[1] = v;
24
25        if(v == 1)
26        {
27          return 1;
28        }else
29        {
30          if(proposes[0] != 1)
31          {
32            return 0;
33          }else
34          {
35            return 1;
36          }
37        }
38      }
39      else // thread B
40      {
41        proposes[0] = v;
42
43        if(v == 0)
44        {
45          return 0;
46        }else
47        {
```

```
48                  if(proposes[1] != 0)
49                  {
50                      return 1;
51                  }else
52                  {
53                      return 0;
54                  }
55              }
56          }
57      }
58  }
```

63.
因为一致对象已经有决定的能力，即只要输入状态时合法的，一致性就是可能的。
64.
同48题?
65.
[java] view plain copy

```
1   package jokes;
2
3   import java.util.concurrent.atomic.*;
4
5   import jokes.NoDefineException;
6
7   public class TeamConsensusObject
8   {
9       private AtomicInteger[] proposes;
10      public final static int INIT = -1;
11
12      public TeamConsensusObject()
13      {
14          proposes = new AtomicInteger[2];
15          for(int i = 0; i < proposes.length; i ++)
16          {
17              proposes[i] = new AtomicInteger(INIT);
18          }
19      }
20
21      public void propose(int v)
22      {
23          if(!proposes[0].compareAndSet(INIT, v))
24          {
25              proposes[1].compareAndSet(INIT, v);
26          }
27      }
28
29      public int decide(int v) throws NoDefineException
30      {
31          if(v == proposes[0].get())
32          {
33              return v;
34          }else if(v == proposes[1].get())
35          {
36              return proposes[0].get();
37          }else
```

```
38          {
39              throw new NoDefineException("");
40          }
41      }
42  }
```

[java] view plain copy

```
1   package jokes;
2
3   import jokes.NoDefineException;
4   import jokes.TeamConsensusObject;
5
6   public class TeamConsensusProtocol
7   {
8       private TeamConsensusObject[] nodes;
9       private int threadNum;
10
11      public TeamConsensusProtocol(int threadNum)
12      {
13          this.threadNum =  (int) Math.pow(2, (Math.ceil((Math.log(threadNum)/ Math.log(2)))));
14          System.out.println("Get threadNum " + this.threadNum);
15          nodes = new TeamConsensusObject[this.threadNum * 2];
16          for(int i = 0; i < nodes.length; i ++)
17          {
18              nodes[i] = new TeamConsensusObject();
19          }
20      }
21
22      public int decide(int threadId, int proposeV) throws NoDefineException
23      {
24          if(threadId >= this.threadNum || threadId < 0)
25          {
26              throw new NoDefineException("");
27          }
28
29          int nodeId = threadId;
30          int v = proposeV;
31          int step = this.threadNum;
32          int base = 0;
33
34          while(step > 0)
35          {
36              nodes[nodeId].propose(v);
37              v = nodes[nodeId].decide(v);
38
39              threadId = (threadId + 1) / 2;
40              base += step;
41              nodeId = base + threadId;
42              step /= 2;
43          }
44
45          return v;
46      }
47
48      public static void main(String[] args)
49      {
```

```
50          TeamConsensusProtocol tester = new TeamConsensusProtocol(7);
51
52          try
53          {
54              System.out.println("Decide " + tester.decide(6, 5));
55          }catch(Exception e)
56          {
57              e.printStackTrace();
58          }
59      }
60  }
```

66.
58题解法，将值换成thread id
67.
如果一个算法的执行的步数是有限的，则无锁和无等待是等价的。对于一个给定的一致性对象，既然它的decide()方法只被每个线程执行一次，则它的无锁实现也是无等待的，反之亦然。
所以没有原子寄存器的无等待实现也就没有它的无锁实现。
68.
只用原子寄存器实现的scan实现的peek()操作的一致数为1。一旦有deq被调用，peek()返回的在deq调用前和调用后的得到的结果是不一样的，不满足一致的条件。
如果没有线程调用dep，因为enq不是一个原子操作，虽然getAndIncrement()的一致数为2，但是items[slot]=x是一个原子寄存器的操作，一致数只有1。所以即使不用deq实现的protocol一致数也是1。

69.

[java] view plain copy

```
1   package jokes;
2
3   import java.util.concurrent.atomic.*;
4
5   public class NewCASObj
6   {
7       private AtomicInteger v;
8
9       public NewCASObj(int v)
10      {
11          this.v = new AtomicInteger(v);
12      }
13
14      public int newCompareAndSet(int exp, int replace)
15      {
16          v.compareAndSet(exp, replace);
17          do
18          {
19              exp = v.get();
20          }while(!v.compareAndSet(exp, exp));
21          return exp;
22      }
23  }
```

70.

根据5.8.1.的证明，n界CAS的一致数>=n。如果有n+1线程第n+1次调用这个对象，返回值为⊥，则前n次调用的任何状态都不能被区分，n+k (k > 0)次调用都不能得到winner的信息。

71.

[java] view plain copy

```
1  package jokes;
2
3  import java.util.concurrent.atomic.*;
4
5  public class CASAssign32
6  {
7     AtomicInteger[] rs;
8
9     public CASAssign32(int initValue)
10    {
11       rs = new AtomicInteger[3];
12       for(int i = 0; i < rs.length; i ++)
13       {
14          rs[i] = new AtomicInteger(initValue);
15       }
16    }
17
18    //Assign in by increasing index
19    public void assign(int i0, int i1, int v0, int v1)
20    {
21       if(i0 >= 3 || i1 >= 3 || i0 < 0 || i1 < 0 || i0 == i1)
22       {
23          return;
24       }
25
26       if(i0 > i1)
27       {
28          int tmp = i0;
29          i0 = i1;
30          i1 = tmp;
31       }
32
33       while(true)
34       {
35          int origV0 = rs[i0].get();
36          int origV1 = rs[i1].get();
37
38          if(rs[i0].compareAndSet(origV0, v0))
39          {
40             if(rs[i1].compareAndSet(origV1, v1))
41             {
42                return;
43             }
44          }
45       }
46    }
47
48    /*
49     * Check by decreasing index.
50     * For example, when rs2 modified in read rs0 in assign(1, 2, x, x)
51     * The returned value is valid as r1, r2 had been checked
52     */
```

```
53      public int read(int i) throws NoDefineException
54      {
55          if(i >= 3 || i < 0)
56          {
57              throw new NoDefineException("");
58          }
59
60          while(true)
61          {
62              int v0 = rs[0].get();
63              int v1 = rs[1].get();
64              int v2 = rs[2].get();
65
66              if(rs[2].compareAndSet(v2, v2))
67                  if(rs[1].compareAndSet(v1, v1))
68                      if(rs[0].compareAndSet(v0, v0))
69                      {
70                          return rs[i].get();
71                      }
72          }
73      }
74  }
```

72.
73.

[java] view plain copy

```
1   package jokes;
2
3   import java.util.concurrent.atomic.*;
4
5   public class CASConsensus
6   {
7       private AtomicInteger proposeId;
8       private int[] proposes;
9       private int threadNum;
10
11      public CASConsensus(int threadNum)
12      {
13          this.threadNum = threadNum;
14          proposes = new int[this.threadNum];
15          proposeId = new AtomicInteger(-1);
16      }
17
18      public int decide(int threadId, int v)
19      {
20          proposes[threadId] = v;
21
22          if(proposeId.compareAndSet(-1, threadId))
23          {
24              return v;
25          }else
26          {
27              for(int i = 0; i < threadNum; i ++)
28              {
```

```
29                  //Suppose 0 is invalid/initial value
30                  if(proposes[i] != 0)
31                  {
32                      if(proposeId.compareAndSet(i, i))
33                      {
34                          return proposes[i];
35                      }
36                  }
37              }
38          }
39          //Make compiler happy
40          return v;
41      }
42  }
```

74.
如果2个线程input不一样，调度器每当探测到其中一个线程（线程i）flip() == true时，在i读到prefer[j]之后写prefer[i]之前停下线程i；线程j solo直到线程j flip() == true，j读到prefer[i]之后写prefer[j]之前，启动线程i；即2个线程同时换到对方的prefer值。这样decide将没有返回。

Chapter6

77. 将确定顺序规范的对象变成确定顺序规范:
做不确定顺序规范到确定顺序规范的映射或者将Node变成ConsensusNode,即给Node一个consensus wrapper，保证不确定的输出将在各个线程中的道一致的确定的输出。
78. 无等待的算法会出错。seq == 0会被当作需要加到list当中的Node ==> tail->next = tail.
79. 不明白“不用通用构造”又“改造这个算法”是什么意思。还是用的通用构造

[java] view plain copy

```
1   package p79;
2
3   import java.util.concurrent.atomic.AtomicReference;
4
5   public class Consensus<T>
6   {
7       private AtomicReference<T> curObj;
8
9       public Consensus()
10      {
11          this.curObj = new AtomicReference<T>();
12          curObj.set(null);
13      }
14
15      public T decide(T next)
16      {
17          curObj.compareAndSet(null, next);
18
19          return curObj.get();
20      }
21  }
```

[java] view plain copy

```
1   package p79;
2
```

```java
public class Node
{
    public Node next;
    public int seq;
    public int exp;
    public int v;
    public Consensus<Node> decideNext;

    public Node()
    {
        this.seq = 0;
        this.next = null;
        decideNext = new Consensus<Node>();
    }

    public Node(int exp, int replace)
    {
        this();
        this.exp = exp;
        this.v = replace;
    }

    public static Node max(Node[] array)
    {
        Node max = array[0];

        for(int i = 1; i < array.length; i ++)
        {
            if(max.seq < array[i].seq)
            {
                max = array[i];
            }
        }

        return max;
    }
}
```

[java] view plain copy

```java
package p79;


public class Universal_CAS
{
    private Node[] announce;
    private Node[] head;
    private Node tail;
    private int n;

    private static ThreadLocal<Integer> ThreadId = new ThreadLocal<Integer>()
    {
        protected Integer initialValue()
        {
            return new Integer(0);
        }
    };
```

```
18
19      public Universal_CAS(int n)
20      {
21          this.n = n;
22          tail = new Node();
23          tail.seq = 1;
24
25          announce = new Node[n];
26          head = new Node[n];
27
28          for(int i = 0; i < n; i ++)
29          {
30              announce[i] = tail;
31              head[i] = tail;
32          }
33      }
34
35      public int read()
36      {
37          Node max = Node.max(head);
38          return max.v;
39      }
40
41      public boolean compareAndSet(int exp, int replace)
42      {
43          int i = ThreadId.get();
44          announce[i] = new Node(exp, replace);
45          head[i] = Node.max(head);
46          Node before = null;
47          while(announce[i].seq == 0)
48          {
49              before = head[i];
50              Node help = announce[(before.seq + 1) % n];
51              Node prefer = help;
52              if(help.seq != 0)
53              {
54                  prefer = announce[i];
55              }
56
57              Node after = before.decideNext.decide(prefer);
58              before.next = after;
59              after.seq = before.seq + 1;
60              if(before.v != after.exp)
61              {
62                  after.v = before.v;
63              }
64
65              head[i] = after;
66          }
67
68          boolean rc = false;
69          if(before.v == announce[i].exp)
70          {
71              announce[i].v = replace;
72              rc = true;
73          }else
```

```
74              {
75                  announce[i].v = before.v;
76              }
77              head[i] = announce[i];
78
79              return rc;
80          }
81      }
```

80. 如果每个线程首先尝试加入自己的结点，在announce[i].seq !=0 后就推出apply()方法，仍然会出现有的线程一直成功而有的线程饿死的状况。如果线程成功的加入自己的结点后开始尝试帮助其它的线程，例如将While()中的帮助循环n-1次，应该是可以的。
81. 将max方法改成下面这样，然后用max(tail)来得到head[i]

[java] view plain copy

```
1   package p79;
2
3   public class Node
4   {
5       public Node next;
6       public int seq;
7       public int exp;
8       public int v;
9       public Consensus<Node> decideNext;
10
11      public Node()
12      {
13          this.seq = 0;
14          this.next = null;
15          decideNext = new Consensus<Node>();
16      }
17
18      public Node(int exp, int replace)
19      {
20          this();
21          this.exp = exp;
22          this.v = replace;
23      }
24
25      public static Node max(Node tail)
26      {
27          Node pre = tail;
28          while(pre.next != null)
29          {
30              pre = pre.next;
31          }
32
33          return pre;
34      }
35  }
```

82.
1. 因为有可能节点 seq已经设置，但是节点没有放到head[]中，
所以实际上的start(A) = m + 1，但是head[A].seq用Node.max()计算出来为m，
使6.4.4不成立。
2.仍然能够正常工作。
因为这种情况下有(head[A].seq - start(A)) >= -1。
假设(head[A].seq - start(A)) < -1，比如 = -2；则必然在head[A]后至少有2个节点增加到链表中。假设他们的seq为m + 1, m + 2。

如果设置 m +2的线程从 head[]中得到m + 1的值，则这个不等式成立；
如果从链表中得到m + 1的值，则这个线程必然已经在while()循环中完成了after.seq = m + 1，head[i] = after的一轮操作，即m + 1已经被这个线程加到head中。
所以这个不等式总是成立。
根据这个不等式，和Theorem 6.4.1的推导方法，线程经过了最多(n + 1)次迭代后仍然能将自己的announce设置到head中。
83.
利用新添加的before和新添加的calculated flag。和java 垃圾搜集。tail的calculated初始化为true。
当用before.next = after构建链表时，也用 after.before = before构建双向链表。
完成将announce添加到链表的操作后，计算log值时，不从tail开始，而是从current向before利用反向链表，直到找到一个before.calculated = true的node，作为初始值开始计算本线程的local log。
当head[i]的值计算完毕，calculated设置完毕后head[i].before = null；使节点脱离反向链表。
但是如果一个线程中途退出了，它将由head[i] hold住无限长的 next链表。所以其他线程也要帮助它脱离next链表。
准备一个全局的dummy node，它将作为next的填充物，防止before.next.decide在脱链后又被加入新的节点。
每个线程在计算完自己的log后帮助其他线程的过时head脱链：
检查每个head[j].seq。如果(current.seq - head[j].seq) >(n + 1)；假设head[j].seq = m，则必然已经有一个node.seq = (m + 1)的calculated = true。即这个node已经不需要了。所以将head[j].next = dummy。
但是其他的线程在沿着链表遍历的时候可能会遇到这个节点。如果使用的是这个节点本身，或者它的before/next节点值，线程都能够发现这个节点的异常，比如after = before.next.decide() ; after == dummy。线程可以重新计算max重来。
因为这个处理并不影响线程的seq关系，所以无等待没有影响。
这个过时的节点可能会被附上新的before节点，但是当before关系被用到时，这个节点自身的calculated已经被置位，所以不会有被反向链表用到的时候。而新被链入的before节点在这个过时节点本身从next链中摘除并回收的时候，新加入的before节点也会被解除被引用的关系。

Chapter7

85. 因为线程自己的节点还被后续节点和tail引用。
被后续节点引用的case: 2个线程A，B
A<--B
B.lock() --> B.pred=A --> B.pred.locked=true --> A.unlock() --> A.locked = false
==> A wants lock again : A.lock() --> A.locked=true --> A.pred = B --> A.pred.locked()=true
==> B wakes up: B.pred.locked =true --> A.pred.locked=true
==> deaklock
被tail引用：
tail=A --> A.lock() --> A.unlock() --> A.lock() --> A.locked=true --> A.pred=tail.getAndSet(A)
==> A.pred=A --> deadlock

86. 第二种比较好。
第一种实现在最好的情况下，比如说同步程度非常高，则每个线程占用总线实现cache miss --> read --> update --> write；然后每个线程重新load得到已经更新到过路障的值，总共有n次回写，n次cache miss。一般的情况下，每一次更新都可以类比于tta锁的unlock，带来n个cache miss，即n^2次cache miss。
第二种情况下，最坏的情况下每个更新只引起一个回写和一个cache miss；加上最后一个线程引起n个cache miss；也是n次回写，2n次cache miss。

87. 互斥
如果已经有一个线程拥有fastpath锁，与它竞争的线程在竞争fp锁的时候或者(oldStamp & FASTPATH) != 0 或者tail.compareAndSet会失败。继续慢速通道的时候会有pred需要等，或者有fastPathWait()需要等fp锁释放。
如果已经有一个现成拥有慢速锁，与他竞争的线程在试图得到fp锁的时候会发现 qnode != null; 继续走慢速通道时根据普通compositeLock互斥。

饥饿
CompositeLock只有在抢到了队列中的节点并加入tail链之后才会开始排队等锁，在抢队列节点和加入tail的时候是没有公平性的。所以在acquireQNode和spliceQNode时候的CAS都可能引起饥饿。
同时，fastPathUnlock()中的CAS也可能引起所有其他线程的饥饿，因为不断有线程timeout，不断有现成开始抢锁，导致tail不断被更新，使这个cas退不出来。

88. 如果这个算法不正确，错误的case应该是这个关键节点同时使localqueue和globalqueue的preNode，如果localqueue一直不知道自己是master，localqueue就不能被加入globalqueue中，最后的关系如下所示：
node<-- ... <-- current node(A) <-- master of another cluster(B) <-- other nodes of another cluster
/ | ____ (master) of local queue(C) <-- other nodes of my cluster
则当current node释放锁的时候，可能会有2个master同时进入critical section，破坏互斥。
但是这是不可能发生的，因为C将自己加到localqueue以后，要执行myPred.waitForGrantOrClusterMaster()。这个方法使C等到A放锁或者发现自己是local master。但是A在放锁之前一定会先执行 localTail.setTailWhenSpliced(true)，即B在发现A.isSuccessorMustWait() == false之前一定有A.isTailWhenSpliced() == true；所以在进入critical region之前会先履行master的职责。

89. 会出现每次每一个cluster只有一个node被加到globalqueue中的情况，即完全没有体现层次锁的优点。可以加一个timer，如果master发现localqueue上的节点太少，时间也充分就等待一段时间。

90. 有一种case可能会出问题：即 ClusterA中的NodeA1已经是localqueue(A)中的tail，加入到globalQueue后被ClusterB中的NodeB回收；同时NodeA2要加到localqueue(A)中。
如果NodeB执行NodeA1.unlock()时，不是原子的同时更改clusterId, SMW和TWS，而是先更改TWS=false。则NodeA2.waitForGrantOrClusterMaster()会发现
getClusterID() == myCluster && !isTailWhenSpliced() && !isuccessorMustWait()
==〉NodeA2会获得锁，即使当前globalqueue上还有其他等待的Node。
如果NodeA1.unlock()时先设置clusterId，除非cpu决定乱序执行，否则应该可以不用原子的修改state。因为如果NodeA1.unlock()是设置的clusterId等于原clusterId，则NodeA2从localqueue(A)中得到的myPred != NodeA1; 如果不等于原clusterId，NodeA2.waitForGrantOrClusterMaster()会发现NodeA2是master，然后从globalQueue去锁。

91.
TAS

[java] view plain copy

```
1    boolean isLocked()
2    {
3        return state.get();
4    }
```

CLH:

[java] view plain copy

```
1    boolean isLocked()
2    {
3        return tail.get().locked;
4    }
```

MCS:

[java] view plain copy

```
1    boolean isLocked()
2    {
```

```
3           QNode node = queue.get();
4           if(node == null)
5           {
6             return false;
7           }else
8           {
9             return node.locked;
10          }
11      }
```

92. 原来的证明过程中现成必须先写某个寄存器然后判断全局状态，如果用R-M-W操作，比如CAS，可以先读然后决定是否写，所以可以避免覆盖的情况。85. 因为线程自己的节点还被后续节点和tail引用。
被后续节点引用的case: 2个线程A，B
A<--B
B.lock() --> B.pred=A --> B.pred.locked=true --> A.unlock() --> A.locked = false
==> A wants lock again : A.lock() --> A.locked=true --> A.pred = B --> A.pred.locked()=true
==> B wakes up: B.pred.locked =true --> A.pred.locked=true
==> deaklock
被tail引用：
tail=A --> A.lock() --> A.unlock() --> A.lock() --> A.locked=true --> A.pred=tail.getAndSet(A)
==> A.pred=A --> deadlock

86. 第二种比较好。
第一种实现在最好的情况下，比如说同步程度非常高，则每个线程占用总线实现cache miss --> read --> update --> write；然后每个线程重新load得到已经更新到过路障的值，总共有n次回写，n次cache miss。一般的情况下，每一次更新都可以类比于tta锁的unlock，带来n个cache miss，即n^2次cache miss。
第二种情况下，最坏的情况下每个更新只引起一个回写和一个cache miss；加上最后一个线程引起n个cache miss；也是n次回写，2n次cache miss。

87. 互斥
如果已经有一个线程拥有fastpath锁，与它竞争的线程在竞争fp锁的时候或者(oldStamp & FASTPATH) != 0 或者tail.compareAndSet会失败。继续慢速通道的时候会有pred需要等，或者有fastPathWait()需要等fp锁释放。
如果已经有一个现成拥有慢速锁，与他竞争的线程在试图得到fp锁的时候会发现 qnode != null; 继续走慢速通道时根据普通compositeLock互斥。
饥饿
CompositeLock只有在抢到了队列中的节点并加入tail链之后才会开始排队等锁，在抢队列节点和加入tail的时候是没有公平性的。所以在acquireQNode和spliceQNode时候的CAS都可能引起饥饿。
同时，fastPathUnlock()中的CAS也可能引起所有其他线程的饥饿，因为不断有线程timeout，不断有现成开始抢锁，导致tail不断被更新，使这个cas退不出来。

88. 如果这个算法不正确，错误的case应该是这个关键节点同时使localqueue和globalqueue的preNode，如果localqueue一直不知道自己是master，localqueue就不能被加入globalqueue中，最后的关系如下所示：
node<-- ... <-- current node(A) <-- master of another cluster(B) <-- other nodes of another cluster
                /|____ (master) of local queue(C) <-- other nodes of my cluster
则当current node释放锁的时候，可能会有2个master同时进入critical section，破坏互斥。
但是这是不可能发生的，因为C将自己加到localqueue以后，要执行myPred.waitForGrantOrClusterMaster()。这个方法使C等到A放锁或者发现自己是local master。但是A在放锁之前一定会先执行 localTail.setTailWhenSpliced(true)，即B在发现A.isSuccessorMustWait() == false之前一定有A.isTailWhenSpliced() == true；所以在进入critical region之前会先履行master的职责。

89. 会出现每次每一个cluster只有一个node被加到globalqueue中的情况，即完全没有体现层次锁的优点。可以加一个timer，如果master发现localqueue上的节点太少，时间也充分就等待一段时间。

90. 有一种case可能会出问题：即 ClusterA中的NodeA1已经是localqueue(A)中的tail，加入到globalQueue后被ClusterB中的NodeB回收；同时NodeA2要加到localqueue(A)中。

如果NodeB执行NodeA1.unlock()时，不是原子的同时更改clusterId, SMW和TWS，而是先更改TWS=false。则NodeA2.waitForGrantOrClusterMaster()会发现
getClusterID() == myCluster && !isTailWhenSpliced() && !isuccessorMustWait()
==〉NodeA2会获得锁，即使当前globalqueue上还有其他等待的Node。
如果NodeA1.unlock()时先设置clusterId，除非cpu决定乱序执行，否则应该可以不用原子的修改state。因为如果NodeA1.unlock()是设置的clusterId等于原clusterId，则NodeA2从localqueue(A)中得到的myPred != NodeA1; 如果不等于原clusterId，NodeA2.waitForGrantOrClusterMaster()会发现NodeA2是master，然后从globalQueue去锁。

91.
TAS

[java] view plain copy

```
boolean isLocked()
{
   return state.get();
}
```

CLH:

[java] view plain copy

```
boolean isLocked()
{
   return tail.get().locked;
}
```

MCS:

[java] view plain copy

```
boolean isLocked()
{
   QNode node = queue.get();
   if(node == null)
   {
    return false;
   }else
   {
    return node.locked;
   }
}
```

92. 原来的证明过程中现成必须先写某个寄存器然后判断全局状态，如果用R-M-W操作，比如CAS，可以先读然后决定是否写，所以可以避免覆盖的情况。

Chapter8

93.

[java] view plain copy

```
package p93;

```

```
3   import java.util.concurrent.locks.Lock;
4
5   public class SimpleRWLock
6   {
7      private class ReadLock implements Lock
8      {
9         public void lock()
10        {
11           try
12           {
13              synchronized(lockObj)
14              {
15                 while(writer)
16                 {
17                    lockObj.wait();
18                 }
19
20                 readers ++;
21              }
22           }catch(Exception e)
23           {
24              e.printStackTrace();
25           }
26        }
27
28        public void unlock()
29        {
30           synchronized(lockObj)
31           {
32              if(readers <= 0)
33              {
34                 lockObj.notifyAll();
35              }
36           }
37        }
38
39        public void lockInterruptibly() throws java.lang.InterruptedException
40        {
41           throw new InterruptedException();
42        }
43
44        public boolean tryLock()
45        {
46           return false;
47        }
48
49        public boolean tryLock(long arg0, java.util.concurrent.TimeUnit arg1) throws java.lang.InterruptedException
50        {
51           throw new InterruptedException();
52        }
53
54        public java.util.concurrent.locks.Condition newCondition()
55        {
56           return null;
57        }
58     }
```

```
59
60      private class WriteLock implements Lock
61      {
62          public void lock()
63          {
64              try
65              {
66                  synchronized(lockObj)
67                  {
68                      while(readers > 0 || writer)
69                      {
70                          lockObj.wait();
71                      }
72
73                      writer =  true;
74                  }
75              }catch(Exception e)
76              {
77                  e.printStackTrace();
78              }
79          }
80
81          public void unlock()
82          {
83              synchronized(lockObj)
84              {
85                  writer = false;
86                  lockObj.notifyAll();
87              }
88          }
89
90          public void lockInterruptibly() throws java.lang.InterruptedException
91          {
92              throw new InterruptedException();
93          }
94
95          public boolean tryLock()
96          {
97              return false;
98          }
99
100         public boolean tryLock(long arg0, java.util.concurrent.TimeUnit arg1) throws java.lang.InterruptedException
101         {
102             throw new InterruptedException();
103         }
104
105         public java.util.concurrent.locks.Condition newCondition()
106         {
107             return null;
108         }
109     }
110
111     private int readers;
112     private boolean writer;
113     private Object lockObj;
114     private ReadLock readLock;
```

```
115     private WriteLock writeLock;
116
117     public SimpleRWLock()
118     {
119         lockObj = new Object();
120         readLock = new ReadLock();
121         writeLock = new WriteLock();
122     }
123
124     public Lock readLock()
125     {
126         return readLock;
127     }
128
129     public Lock writeLock()
130     {
131         return writeLock;
132     }
133
134     public static void main(String[] args)
135     {
136         SimpleRWLock lock = new SimpleRWLock();
137
138         lock.readLock().lock();
139         lock.readLock().unlock();
140         lock.writeLock().lock();
141         lock.writeLock().unlock();
142     }
143 }
```

94.
会有 readlock.lock() --> readwait > readrelease --> writelock.lock() --> write = true --> waiting for readwait == readrelease ==> deadlock
其他的线程会有 write == true ==> block

95. 不一定全部会返回，因为可能在别人的cond上死锁，boss notify自己的锁线程感应不到。比如下面的case:

[java] view plain copy

```
1   package p95;
2
3
4   public class Prefer
5   {
6       private int balance;
7       private int preferRequests;
8
9       public static final int PREFER = 1;
10      public static final int NORMAL = 2;
11
12      public Prefer()
13      {
14          balance = 0;
15          preferRequests = 0;
16      }
```

```
17
18      public Prefer(int initial)
19      {
20          this();
21          balance = initial;
22      }
23
24      private void deposit_nolock(int k)
25      {
26          balance += k;
27      }
28      public synchronized void deposit(int k)
29      {
30          deposit_nolock(k);
31          this.notifyAll();
32      }
33
34      public synchronized void withdraw(int k, int style)
35      {
36          try
37          {
38              switch(style)
39              {
40              case NORMAL:
41                  while(preferRequests > 0 || balance < k)
42                  {
43                      this.wait();
44                  }
45                  balance -= k;
46                  break;
47              case PREFER:
48                  preferRequests ++;
49                  while(balance < k)
50                  {
51                      this.wait();
52                  }
53                  balance -= k;
54                  preferRequests --;
55                  this.notifyAll();
56                  break;
57              default:
58                  throw new InterruptedException();
59              }
60          }catch(InterruptedException e)
61          {
62              this.notifyAll();
63          }
64      }
65
66      public synchronized void transfer(Prefer other, int k)
67      {
68          other.withdraw(k, NORMAL);
69          deposit_nolock(k);
70      }
71
72  }
```

[java] view plain copy

```java
package p95;

public class TestThread extends Thread
{
    private Prefer myAccount;
    private Prefer otherAccount;
    public final int WITHDRAW = 200;
    public final int BOSS = 1000;

    public TestThread(Prefer me, Prefer other)
    {
        this.myAccount = me;
        this.otherAccount = other;
    }

    public void run()
    {
        System.out.println("To transfer ");
        myAccount.transfer(otherAccount, WITHDRAW);
        System.out.println("End of transfer ");
    }

    public void boss()
    {
        myAccount.deposit(BOSS);
    }

    public static void main(String[] args)
    {
        Prefer[] accounts = new Prefer[2];
        for(int i = 0; i < accounts.length; i ++)
        {
            accounts[i] = new Prefer(0);
        }

        TestThread tester1 = new TestThread(accounts[0], accounts[1]);
        TestThread tester2 = new TestThread(accounts[1], accounts[0]);

        tester1.start();
        tester2.start();

        try
        {
            Thread.sleep(1000);
        }catch(Exception e)
        {
            e.printStackTrace();
        }
        System.out.println("After sleep ");

        tester1.boss();
        tester2.boss();
    }
```

```
    }
```

96.

[java] view plain copy

```
package p96;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Bathroom_1
{
    private Lock lock;
    private Condition maleCond;
    private Condition femaleCond;
    private int maleAcq;
    private int maleRel;
    private int femaleAcq;
    private int femaleRel;
    private int maleWait;
    private int femaleWait;

    public static final int MALE = 1;
    public static final int FEMALE = 2;

    public Bathroom_1()
    {
        lock = new ReentrantLock();
        maleCond = lock.newCondition();
        femaleCond = lock.newCondition();
    }

    public void maleEnter()
    {
        lock.lock();

        try
        {
            maleWait ++;
            while((femaleWait != femaleAcq && maleAcq != maleRel)
                    || (femaleAcq != femaleRel))
            {
                maleCond.await();
            }

            maleAcq ++;
        }catch(Exception e)
        {
            e.printStackTrace();
            maleWait --;
            maleCond.signalAll();
            femaleCond.signalAll();
        }finally
        {
            lock.unlock();
```

```
52          }
53      }
54
55      public void maleExit()
56      {
57          lock.lock();
58
59          maleRel ++;
60          if(maleRel == maleAcq)
61          {
62              femaleCond.signalAll();
63          }
64
65          lock.unlock();
66      }
67
68      public void femaleEnter()
69      {
70          lock.lock();
71
72          try
73          {
74              femaleWait ++;
75              while((maleWait != maleAcq && femaleAcq != femaleRel)
76                      || (maleAcq != maleRel))
77              {
78                  femaleCond.await();
79              }
80              femaleAcq ++;
81          }catch(Exception e)
82          {
83              e.printStackTrace();
84              femaleWait --;
85              maleCond.signalAll();
86              femaleCond.signalAll();
87          }finally
88          {
89              lock.unlock();
90          }
91      }
92
93      public void femaleExit()
94      {
95          lock.lock();
96
97          femaleRel ++;
98          if(femaleRel == femaleAcq)
99          {
100             maleCond.signalAll();
101         }
102
103         lock.unlock();
104     }
105 }
```

97.

[java] view plain copy

```java
package p97;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Rooms
{
    public interface Handler
    {
        void onEmpty();
    }

    private Lock lock;
    private Condition[] conds;
    private int waiting[];
    private int acquire[];
    private int release[];
    private Handler[] handlers;
    private final int m;
    private int currRoom;

    public Rooms(int m)
    {
        this.m = m;
        lock = new ReentrantLock();
        conds = new Condition[m + 1];
        for(int i = 0; i < conds.length; i ++)
        {
            conds[i] = lock.newCondition();
        }
        waiting = new int[m + 1];
        acquire = new int[m + 1];
        release = new int[m + 1];
        handlers = new Handler[m + 1];
        currRoom = -1;
    }

    private boolean toWait(int index)
    {
        if(currRoom == -1)
        {
            return false;
        }
        if(currRoom != index)
        {
            return true;
        }
        for(int i = 1; i <= m; i ++)
        {
            int other = (i + index) % (m + 1);
            if(waiting[other] != acquire[other])
            {
                return true;
```

```
55              }
56          }
57          return false;
58      }
59
60      public void enter(int i)
61      {
62          if(i < 0 || i > m)
63          {
64              return;
65          }
66
67          lock.lock();
68
69          try
70          {
71              waiting[i] ++;
72
73              while(toWait(i))
74              {
75                  conds[i].await();
76              }
77
78              acquire[i] ++;
79              currRoom = i;
80          }catch(Exception e)
81          {
82              e.printStackTrace();
83          }finally
84          {
85              lock.unlock();
86          }
87      }
88
89      private int notifyWho()
90      {
91          for(int i = 1; i <= m; i ++)
92          {
93              int other = (i + currRoom) % (m + 1);
94              if(waiting[other] != acquire[other])
95              {
96                  return other;
97              }
98          }
99
100         return -1;
101     }
102
103     public boolean exit()
104     {
105         lock.lock();
106
107         release[currRoom] ++;
108         if(release[currRoom] == acquire[currRoom])
109         {
110             if(handlers[currRoom] != null)
```

```
111                 {
112                     handlers[currRoom].onEmpty();
113                 }
114 
115                 int other = notifyWho();
116                 if(other >= 0)
117                 {
118                     conds[other].signalAll();
119                 }
120                 currRoom = -1;
121             }
122 
123             lock.unlock();
124             return true;
125         }
126 
127         public void setExitHandler(int i, Rooms.Handler handler)
128         {
129             if(i < 0 || i > m)
130             {
131                 return;
132             }
133 
134             lock.lock();
135             handlers[i] = handler;
136             lock.unlock();
137         }
138     }
```

98.

[java] view plain copy

```
1   package p98;
2   
3   import java.util.concurrent.locks.Condition;
4   import java.util.concurrent.locks.Lock;
5   import java.util.concurrent.locks.ReentrantLock;
6   
7   
8   public class CountDownLatch
9   {
10      private Lock lock;
11      private Condition cond;
12      private int m;
13  
14      public CountDownLatch(int m)
15      {
16          this.m = m;
17          lock = new ReentrantLock();
18          cond = lock.newCondition();
19      }
20  
21      public void countDown()
22      {
```

```
23            lock.lock();
24            m --;
25            if(m <= 0)
26            {
27                cond.signalAll();
28            }
29            lock.unlock();
30        }
31
32        public void await()
33        {
34            lock.lock();
35
36            try
37            {
38                while(m > 0)
39                {
40                    cond.await();
41                }
42            }catch(Exception e)
43            {
44                e.printStackTrace();
45            }finally
46            {
47                lock.unlock();
48            }
49        }
50
51        /*
52         * Lock is not necessary as:
53         * 1.   set m at anytime can get matching history
54         * 2.   User should take care of reset since I don't want to block reset() method
55         */
56        public void reset(int m)
57        {
58            lock.lock();
59            this.m = m;
60            lock.unlock();
61        }
62    }
```

Chapter9

100. 将add()方法的 if(cur.key == key) return false去掉，改为找到 pred.key <= key && curr.key > key为插入的位置。remove()方法和contains()方法都是找到任意一个key == key时结束。

101. 所有的锁都按照升序获得，不会有循环，所以无死锁。

102. 当线程拥有pred.lock和curr.lock的时候，与这2个节点无关的操作并行操作，结果与这个add()方法无关，可以在串行化的历史中任意安排这2个操作的顺序；同时与这2个节点相关的操作，包括在pred后插入节点，删除pred和删除curr都将在获得2个锁的时候串行执行。这是add方法的可线性化点。

103. 没有对获得锁的循环。

104. 只要有pred和curr的关系不断变化就可以。比如说当前节点 head--> ... --> A --> B--> ... --> tail ; Threada想要remove B，Threadb不断在AB之间插入节点：

Ta.pred = A, Ta.curr = B; ==> Tb.pred = A, Tb.curr = B; Tb.pred.lock(), Tb.curr.lock(); ==> Tb inserts A1 ==> A-->A1-->B ==> Ta.pred.lock(), Ta.curr.lock() ==> Ta.validate ==> Ta.pred.next (A.next) != Ta.curr (B) ==> Ta tries remove again. ==> Tb inserts A2 ==> Ta failed ==> ...

或者重复的插入和删除：

Ta.pred = A, Ta.curr = B ==> Tb.pred = A, Tb.curr = B; Tb.pred.lock(), Tb.curr.lock() ==> Tb inserts A1 ==> A-->A1-->B
==> Ta.pred.lock(), Ta.curr.lock() ==> Ta.validate ==> Ta.pred.next != Ta.curr ==> Ta tries remove again.
==> Ta.pred = A1, Ta.curr = B; Tb.pred = A, Tb.curr = A1; Tb.pred.lock, Tb.curr.lock; ==> Tb.remove(curr) ==> A-->B
==> Ta.pred.lock, Ta.curr.lock ==> Ta.validate ==> Ta.pred not reachable ==> Ta tries remove again
==> Repeat
105.

[java] view plain copy

```
1       public boolean contains(T item) {
2         Node last = null, pred = null, curr = null;
3         int key = item.hashCode();
4         head.lock();
5         try {
6           pred = head;
7           curr = pred.next;
8           curr.lock();
9           try {
10            while (curr.key < key) {
11              pred.unlock();
12              pred = curr;
13              curr = curr.next;
14              curr.lock();
15            }
16            return (curr.key == key);
17          } finally {
18            curr.unlock();
19          }
20        } finally {
21          pred.unlock();
22        }
```

106. 如果只有add交换顺序，remove不交换，会死锁。如果remove也交换，因为validation总是在获得2个锁之后，则算法仍然正确。
107. 如果pred可达，则它肯定能达到tail。
否则，可以用归纳法证明如果predA != null, 而且predA不可达，则predA所在的list一定能够达到一个可达的并能够达到Tail的节点（包括tail）。
当pred最开始从list删除的时候，因为有curr.lock（此时curr = predA），所以被删除的predA肯定有predA.next可达。
假设predA不可达，根据假设它所在的list一定能够达到一个可达的而且能够到达tail的节点（R）；因为不可能对不可达的节点进行操作，所以增加节点并不会改变这个性质。如果删除节点R, 根据上面的证明，R一定有R-->R.next-->...-->tail ==〉predA --> ... --> R -->R.next --> ... --> tail
108. 常规的锁法之所以需要2个锁，是因为要确定pred->curr的关系，即确保pred->next在add/remove操作过程中不会改变。因为add操作只涉及到pred->next和一个将成为pred->next的新节点，所以add之需要锁pred。具体来说，考虑下面的case：

1. ThreadA.add && ThreadB.add。ThreadA.pred.lock ==> ThreadB waiting for pred.lock ==> ThreadA.pred.next = new, new->next = ThreadA.curr ==> ThreadB.pred.lock() ==> ThreadB.valiation ==> Thread.B.pred.next != ThreadB.curr。
2. ThreadA.add && ThreadB.remove(pred)。ThreadA.pred.lock ==> ThreadB.pred.pred.lock(), ThreadB.pred waiting for lock ==> ThreadA.pred.next = new, Thread.new.next = curr ==> ThreadB.pred.lock() ==> ThreadB.pred.pred.next = new
3. ThreadA.add && ThreadB.remove(pred)。ThreadB.pred.pred.lock(), ThreadB.pred.lock() ==> ThreadA.pred waiting for lock ==> ThreadB.pred.pred.next = ThreadB.cur; ==> ThreadA.pred.lock() ==> ThreadA.validate ==> pred is not reachable
4. ThreadA.add && ThreadB.remove(curr)。ThreadA.pred.lock ==> ThreadB.pred waiting for lock ==> ThreadA.pred.next = new, ThreadA.pred.next.next = ThreadA.cur; ==> ThreadB.pred.lock() ==> ThreadB.validate ==> ThreadB.pred.next != ThreadB.cur

5 ThreadA.add && ThreadB.remove(curr)。ThreadB.pred.lock(), ThreadB.cur.lock() ==> ThreadA.pred waiting for lock ==> ThreadB.pred.next = ThreadB.cur.next ==> ThreadA.pred.lock() ==> ThreadA.validate ==> ThreadA.pred.next != ThreadA.curr

109. contains()是一个有明确定义的方法，它的起作用的时间点在方法返回的时候，所以如果contains的返回在remove之后则contains必须返回false。所以题目中的alternative不是可线性化的，因为它不能覆盖这样的case：
ThreadA.pred.lock, ThreadA.cur.lock ==> ThreadA.pred.next = new, Thread.new.next = Thread.cur; ==> A-->A1-->B ==> ThreadB.contains(A1) ==> ThreadB.validate = true, ThreadB.cur = A1 ==> ThreadA.remove(A1) ==> A-->B ==> ThreadB.cur.key == A1 ==> ThreadB.contains(A1) return true.

110. 不能。
head-->A-->B-->C-->tail ==> ThreadA.remove(B) ==> ThreadA.B.next = null ==> ThreadB.contains(C) ==> ThreadB.B.next == null ==> C is not reachable.
lock-less也不能。

111. 如图9.20 (a)所示。

112. 不能。因为add()会破坏这种关系：
ThreadA.pred = A, ThreadA.cur = B ==> A-->B ==> ThreadB.add() ==> A-->A1-->B ==> ThreadA inserts ==> A-->A2-->B

113. 不能。因为remove会对pred和curr做modify，但只锁一个node，比如说ThreadA.pred == ThreadB.cur && 只锁pred，则ThreadB可以在没有锁保护的情况下modifyThreadA.pred，即Thread.pred的锁并没有起作用。具体说来其结果如图9.22 (b)所示。

114. 好处：分担add/remove的复杂性，尽快清除节点。
缺点：承担add/remove的复杂性，contains性能抖动大。

115. 不需要。因为pred.marked==false, 且pred.key < key；所以要找的curr肯定在pred节点的后面，而且可达。

116. 不能。因为这2种算法是利用排好序的链表找目标节点可能在的位置。

117. 如下所示。 其中find()返回除了pred/curr之外的可以重用的Node。
因为 snip = pred.next.compareAndSet(curr, succ, false, false); 所以只有一个线程能够物理删除可重用的节点，之后这个节点就不可达了，而且处于被标记的状态。所以只有一个线程能够得到这个可重用的节点，包括运行contains()的节点。因此在add()方法中对这个节点的赋值不需要同步机制，所以只要这个节点到达了add()就可以像new Node那样使用了。
[java] view plain copy

```java
package p117;

/*
 * LockFreeList.java
 *
 * Created on January 4, 2006, 2:41 PM
 *
 * From "Multiprocessor Synchronization and Concurrent Data Structures",
 * by Maurice Herlihy and Nir Shavit.
 * Copyright 2006 Elsevier Inc. All rights reserved.
 */
import java.util.concurrent.atomic.AtomicMarkableReference;

/**
 * Lock-free List based on M. Michael's algorithm.
 * @param T Item type.
 * @author Maurice Herlihy
 */
public class LockFreeList<T> {
  /**
```

```
21      * First list node
22      */
23     Node head;
24     /**
25      * Constructor
26      */
27     public LockFreeList() {
28       this.head  = new Node(Integer.MIN_VALUE);
29       Node tail = new Node(Integer.MAX_VALUE);
30       while (!head.next.compareAndSet(null, tail, false, false));
31     }
32     /**
33      * Add an element.
34      * @param item element to add
35      * @return true iff element was not there already
36      */
37    //fulltopic: Adjusted to reuse marked node
38     public boolean add(T item) {
39       int key = item.hashCode();
40       Node node = null;
41       while (true) {
42         // find predecessor and curren entries
43         Window window = find(head, key);
44         Node pred = window.pred, curr = window.curr;
45         // is the key present?
46         if (curr.key == key) {
47           return false;
48         } else {
49           // splice in new node
50           if(node == null)
51           {
52              if(window.reuseNode != null)
53              {
54                 node = window.reuseNode;
55                 node.item = item;
56              }else
57              {
58                 node = new Node(item);
59              }
60           }
61           node.next = new AtomicMarkableReference<LockFreeList<T>.Node>(curr, false);
62           if (pred.next.compareAndSet(curr, node, false, false)) {
63             return true;
64           }
65         }
66       }
67     }
68     /**
69      * Remove an element.
70      * @param item element to remove
71      * @return true iff element was present
72      */
73     public boolean remove(T item) {
74       int key = item.hashCode();
75       boolean snip;
76       while (true) {
```

```
77          // find predecessor and curren entries
78          Window window = find(head, key);
79          Node pred = window.pred, curr = window.curr;
80          // is the key present?
81          if (curr.key != key) {
82            return false;
83          } else {
84            // snip out matching node
85            Node succ = curr.next.getReference();
86            snip = curr.next.attemptMark(succ, true);
87            if (!snip)
88              continue;
89            pred.next.compareAndSet(curr, succ, false, false);
90            return true;
91          }
92        }
93      }
94      /**
95       * Test whether element is present
96       * @param item element to test
97       * @return true iff element is present
98       */
99      public boolean contains(T item) {
100       int key = item.hashCode();
101       // find predecessor and curren entries
102       Window window = find(head, key);
103       Node curr = window.curr;
104       return (curr.key == key);
105     }
106     /**
107      * list node
108      */
109     private class Node {
110       /**
111        * actual item
112        */
113       T item;
114       /**
115        * item's hash code
116        */
117       int key;
118       /**
119        * next node in list
120        */
121       AtomicMarkableReference<Node> next;
122       /**
123        * Constructor for usual node
124        * @param item element in list
125        */
126       Node(T item) {      // usual constructor
127         this.item = item;
128         this.key = item.hashCode();
129         this.next = new AtomicMarkableReference<Node>(null, false);
130       }
131       /**
132        * Constructor for sentinel node
```

```
133         * @param key should be min or max int value
134         */
135        Node(int key) { // sentinel constructor
136          this.item = null;
137          this.key = key;
138          this.next = new AtomicMarkableReference<Node>(null, false);
139        }
140      }
141
142      /**
143       * Pair of adjacent list entries.
144       */
145      class Window {
146        /**
147         * Earlier node.
148         */
149        public Node pred;
150        /**
151         * Later node.
152         */
153        public Node curr;
154
155        public Node reuseNode;
156        /**
157         * Constructor.
158         */
159        Window(Node pred, Node curr) {
160          this.pred = pred; this.curr = curr;
161        }
162
163        //fulltopic: Set userNdoe
164        Window(Node pred, Node curr, Node reuseNode)
165        {
166           this(pred, curr);
167           this.reuseNode = reuseNode;
168        }
169      }
170
171      /**
172       * If element is present, returns node and predecessor. If absent, returns
173       * node with least larger key.
174       * @param head start of list
175       * @param key key to search for
176       * @return If element is present, returns node and predecessor. If absent, returns
177       * node with least larger key.
178       */
179      public Window find(Node head, int key) {
180        Node pred = null, curr = null, succ = null, reuseNode = null;
181        boolean[] marked = {false}; // is curr marked?
182        boolean snip;
183        retry: while (true) {
184          pred = head;
185          curr = pred.next.getReference();
186          while (true) {
187            succ = curr.next.get(marked);
188            while (marked[0]) {          // replace curr if marked
```

```
189             snip = pred.next.compareAndSet(curr, succ, false, false);
190             if (!snip) continue retry;
191             //fulltopic: get use node
192             if(curr.key == key)
193             {
194                 reuseNode = curr;
195             }
196             curr = pred.next.getReference();
197             succ = curr.next.get(marked);
198           }
199           if (curr.key >= key)
200             return new Window(pred, curr, reuseNode);
201           pred = curr;
202           curr = succ;
203         }
204       }
205     }
206   }
```

118. 因为contains() --> find() 会忽略所有marked的节点。如果线性化的顺序是remove-->add-->contains，contains().find()会发现curr.key == key而不是停止在marked removed node with key == key。

Chapter10

119.

[java] view plain copy

```
1     private enum NodeType {ITEM, RESERVATION, SENTINEL};
2
3   public class SynchronousDualQueue<T> {
4     AtomicReference<Node> head;
5     AtomicReference<Node> tail;
6     private final Node NullNode = new Node(null, NodeType.SENTINEL);
7
8   /* Replace all null with NullNode */
9   }
```

120.

[java] view plain copy

```
1   package p120;
2
3   public class TwoThreadLockFreeQueue<T>
4   {
5     int head = 0, tail = 0;
6     T[] items;
7
8     @SuppressWarnings("unchecked")
9     public TwoThreadLockFreeQueue(int capacity)
10    {
11      head = tail = 0;
12      items = (T[]) new Object[capacity];
13    }
```

```
14
15      public void enq(T x)
16      {
17          //rmb
18          while(tail - head == items.length)
19          {
20              //rmb
21          }
22
23          items[tail % items.length] = x;
24          tail ++;
25          //wmb
26      }
27
28      public Object deq()
29      {
30          //rmb
31          while(tail == head)
32          {
33              //rmb
34          }
35
36          Object x = items[head % items.length];
37          head ++;
38          //wmb
39          return x;
40      }
41  }
```

121.

[java] view plain copy

```
1   package p121;
2
3   import java.util.concurrent.locks.Condition;
4   import java.util.concurrent.locks.Lock;
5   import java.util.concurrent.locks.ReentrantLock;
6
7   public class TwoLockArrayQueue
8   {
9       private final int capacity;
10      private int enqSideSize;
11      private int deqSideSize;
12      private Lock enqLock;
13      private Condition enqCond;
14      private Lock deqLock;
15      private Condition deqCond;
16      private int head;
17      private int tail;
18      private T[] datas;
19
20      @SuppressWarnings("unchecked")
21      public TwoLockArrayQueue(int cap)
22      {
23          capacity = cap;
```

```
24          enqSideSize = 0;
25          deqSideSize = 0;
26          head = 0;
27          tail = 0;
28          enqLock = new ReentrantLock();
29          deqLock = new ReentrantLock();
30          enqCond = enqLock.newCondition();
31          deqCond = deqLock.newCondition();
32
33          datas = (T[]) new Object[capacity];
34      }
35
36      public void enq(T x)
37      {
38          boolean notifyDeq = false;
39
40          enqLock.lock();
41          try
42          {
43              while(enqSideSize == capacity)
44              {
45                  deqLock.lock();
46                  try
47                  {
48                      enqSideSize = capacity - deqSideSize;
49                      deqSideSize = 0;
50                  }finally
51                  {
52                      deqLock.unlock();
53                  }
54
55                  if(enqSideSize == capacity)
56                  {
57                      enqCond.await();
58                  }
59              }
60
61              datas[tail % capacity] = x;
62              tail ++;
63              enqSideSize ++;
64
65              //No notification lost as deqSideSize can only increase and still fit the condition
66              if(enqSideSize - deqSideSize <= 1)
67              {
68                  notifyDeq = true;
69              }
70
71          }catch(InterruptedException e)
72          {
73              e.printStackTrace();
74          }finally
75          {
76              enqLock.unlock();
77          }
78
79          if(!notifyDeq) return;
```

```
80
81          deqLock.lock();
82          try
83          {
84              deqCond.signalAll();
85          }finally
86          {
87              deqLock.unlock();
88          }
89      }
90
91      public T deq()
92      {
93          T x = null;
94          boolean notifyEnq = false;
95
96          deqLock.lock();
97
98          try
99          {
100             /*
101              * No notification lost as when enqSideSize increased in between while and await,
102              * there would always an enq thread knows enqSideSize - deqSideSize <= 1
103              * and requires deqLock for notification.
104              */
105             while(deqSideSize >= enqSideSize)
106             {
107                 deqCond.await();
108             }
109
110             x = datas[head % capacity];
111             head ++;
112             deqSideSize ++;
113
114             if(enqSideSize >= capacity)
115             {
116                 notifyEnq = true;
117             }
118         }catch(InterruptedException e)
119         {
120             e.printStackTrace();
121         }finally
122         {
123             deqLock.unlock();
124         }
125
126
127         if(notifyEnq)
128         {
129             enqLock.lock();
130             try
131             {
132                 enqCond.signalAll();
133             }finally
134             {
135                 enqLock.unlock();
```

```
136             }
137           }
138           return x;
139       }
140   }
```

122. 必须。head.next != null --> deq 必须是原子的，否则可能有这样的case：queue中只有一个元素，2个线程deq，2个现成都得到head.next != null，然后2个现成分别在拿到锁之后deq，错误。

123.

使用SynchronousDualQueue, 在enq或deq消除（而不是加入list）的时候排除不符合要求的case。是分散的，低征用的，随机的。

[java] view plain copy

```
1   package p123;
2
3   import java.util.concurrent.atomic.AtomicReference;
4
5   public class FeedOthers {
6     AtomicReference head;
7     AtomicReference tail;
8
9     public FeedOthers() {
10      Node sentinel = new Node(null, NodeType.ITEM);
11      head = new AtomicReference(sentinel);
12      tail = new AtomicReference(sentinel);
13    }
14
15    public void enq(T e) {
16      Node offer = new Node(e, NodeType.ITEM);
17      while (true) {
18        Node t = tail.get();
19        Node h = head.get();
20        if (h == t || t.type == NodeType.ITEM) {
21          Node n = t.next.get();
22          if (t == tail.get()) {
23            if (n != null) {
24              tail.compareAndSet(t, n);
25            } else if (t.next.compareAndSet(n, offer)) {
26              tail.compareAndSet(t, offer);
27              while (offer.item.get() == e); // spin
28              h = head.get();
29              if (offer == h.next.get()) {
30                head.compareAndSet(h, offer);
31              }
32              return;
33            }
34          }
35        } else {
36          Node n = h.next.get();
37          if (t != tail.get() || h != head.get() || n == null) {
38            continue; // inconsistent snapshot
39          }
40
41          //fulltopic
42          else if(n.type != NodeType.RESERVATION)
```

```
43          {
44              continue;
45          }
46          T other = n.item.get();
47          if(other.equals(e))
48          {
49  //          System.out.println("Can not eat from self ");
50              continue;
51          }
52
53          boolean success = n.item.compareAndSet(other, e);
54          head.compareAndSet(h, n);
55          if (success) {
56            return;
57          }
58        }
59      }
60    }
61
62    public T deq(T me) {
63      Node offer = new Node(me, NodeType.RESERVATION); //fulltopic
64
65      while (true) {
66        Node t = tail.get();
67        Node h = head.get();
68
69        if (h == t || t.type == NodeType.RESERVATION) {
70          Node n = t.next.get();
71          if (t == tail.get()) {
72            if (n != null) {
73              tail.compareAndSet(t, n);
74            } else if (t.next.compareAndSet(n, offer)) {
75              tail.compareAndSet(t, offer);
76              while (offer.item.get() == me); // spin
77              h = head.get();
78              if (offer == h.next.get()) {
79                head.compareAndSet(h, offer);
80              }
81              return offer.item.get();
82            }
83          }
84        } else {
85          Node n = h.next.get();
86          if (t != tail.get() || h != head.get() || n == null) {
87            continue; // inconsistent snapshot
88          }
89
90          //fulltopic
91          else if(n.type != NodeType.ITEM)
92          {
93              continue;
94          }
95          T item = n.item.get();
96          if(item == null || item.equals(me))
97          {
98  //          System.out.println("Can not feed self ");
```

```
99                continue;
100             }
101
102             boolean success = n.item.compareAndSet(item, null);
103             head.compareAndSet(h, n);
104             if (success) {
105               return item;
106             }
107           }
108         }
109       }
110
111       private enum NodeType {ITEM, RESERVATION};
112       private class Node {
113         volatile NodeType type;
114         volatile AtomicReference item;
115         volatile AtomicReference next;
116         Node(T item, NodeType type) {
117           this.item = new AtomicReference(item);
118           this.next = new AtomicReference(null);
119           this.type = type;
120         }
121         public String toString() {
122           return "Node[" + type +
123               ", item: " + item +
124               ", next: " + next +
125               "]";
126         }
127       }
128     }
```

124.
1. 应该以38行成功为线性化点，否则，
假设queue的状况是 head(A)-->B-->C-->sentinel
2个线程deq的执行顺序是 ThreadA.CAS --> ThreadB.CAS --> ThreadB.return --> ThreadA.return
以return为线性化点，返回结果应该是ThreadB得到A，ThreadA得到B; 但是算法的结果是 ThreadA得到A，ThreadB得到B。
2. enq的线性化点位16行成功。如果以tail更新为可线性化点，假设一个线程16行执行部成功，但是21行更新tail成功，则这个方法在没有实际enq的情况下使是方法的执行状态可见，这没有意义。
125.
1. enq是无等待的，因为每个调用能够在有限步完成。deq不是无锁的，假设只有一个deq线程，没有enq，这个线程运行无限步也不能完成一个调用。
2. 大概是这样：
deq()的可线性化点在13行成功地时刻。
对于enq和deq的操作，enq的可线性化点在第7行执行结束。
对于enq和enq的操作，enq的可线性化点在它们enq的值被deq成功的时刻，即第13行操作成功的时刻。

Chapter11

126.

[java] view plain copy

```
1   package p126;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
```

```
5
6   public class UnboundListStack
7   {
8       public class Node
9       {
10          private T item;
11          public Node next;
12
13          public Node(T item)
14          {
15              this.item = item;
16              this.next = null;
17          }
18
19          public T get()
20          {
21              return item;
22          }
23      }
24
25      private Lock lock;
26      private Node head;
27
28      public UnboundListStack()
29      {
30          this.lock = new ReentrantLock();
31          head = new Node(null);
32      }
33
34      public void push(T x)
35      {
36          Node node = new Node(x);
37
38          lock.lock();
39
40          try
41          {
42              node.next = head.next;
43              head.next = node;
44          }finally
45          {
46              lock.unlock();
47          }
48      }
49
50      public T pop() throws Exception
51      {
52          lock.lock();
53
54          try
55          {
56              if(head.next == null)
57              {
58                  throw new Exception("Empty");
59              }else
60              {
```

```
61                  T item = head.next.get();
62                  head.next = head.next.next;
63                  return item;
64              }
65          }finally
66          {
67              lock.unlock();
68          }
69      }
70
71  }
127.
```

[java] view plain copy

```
1   package p126;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
5
6   public class UnboundListStack
7   {
8       public class Node
9       {
10          private T item;
11          public Node next;
12
13          public Node(T item)
14          {
15              this.item = item;
16              this.next = null;
17          }
18
19          public T get()
20          {
21              return item;
22          }
23      }
24
25      private Lock lock;
26      private Node head;
27
28      public UnboundListStack()
29      {
30          this.lock = new ReentrantLock();
31          head = new Node(null);
32      }
33
34      public void push(T x)
35      {
36          Node node = new Node(x);
37
38          lock.lock();
39
40          try
```

```
41              {
42                  node.next = head.next;
43                  head.next = node;
44              }finally
45              {
46                  lock.unlock();
47              }
48          }
49
50          public T pop() throws Exception
51          {
52              lock.lock();
53
54              try
55              {
56                  if(head.next == null)
57                  {
58                      throw new Exception("Empty");
59                  }else
60                  {
61                      T item = head.next.get();
62                      head.next = head.next.next;
63                      return item;
64                  }
65              }finally
66              {
67                  lock.unlock();
68              }
69          }
70
71      }
```

128.

[java] view plain copy

```
1   package p128;
2
3   import java.util.concurrent.atomic.AtomicStampedReference;
4
5   import utils.EmptyException;
6   import ch7.Backoff;
7
8   public class LockFreeStack
9   {
10      public class Node
11      {
12          public T value;
13          public Node next;
14          public Node(T value)
15          {
16              this.value = value;
17              next = null;
18          }
19      }
20
```

```
21      static final int INITCAPACITY = 2;
22      ThreadLocal nodeList = new ThreadLocal()
23      {
24          protected Node initialValue()
25          {
26              Node head = new Node(null);
27
28              for(int i = 0; i < INITCAPACITY; i ++)
29              {
30                  Node node = new Node(null);
31                  node.next = head.next;
32                  head.next = node;
33              }
34
35              return head;
36          }
37      };
38
39      private Node allocNode()
40      {
41          Node head = nodeList.get();
42          if(head.next == null)
43          {
44              return new Node(null);
45          }else
46          {
47              Node node = head.next;
48              head.next = head.next.next;
49              node.next = null;
50              return node;
51          }
52      }
53
54      private void freeNode(Node node)
55      {
56          if(node == null)
57          {
58              return;
59          }
60          node.value = null;
61          Node head = nodeList.get();
62          node.next = head.next;
63          head.next = node;
64      }
65
66      AtomicStampedReference top = new AtomicStampedReference(null, 0);
67      static final int MIN_DELAY = 1;
68      static final int MAX_DELAY = 10;
69      Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
70
71      protected boolean tryPush(Node node)
72      {
73          int[] stamp = new int[1];
74          Node oldTop = top.get(stamp);
75          node.next = oldTop;
76          return (top.compareAndSet(oldTop, node, stamp[0], stamp[0] + 1));
```

```
77          }
78
79          public void push(T value)
80          {
81              Node node = allocNode();
82              node.value = value;
83              while(true)
84              {
85                  if(tryPush(node))
86                  {
87                      return;
88                  }else
89                  {
90                      try
91                      {
92                          backoff.backoff();
93                      }catch(Exception e)
94                      {
95                          e.printStackTrace();
96                      }
97                  }
98              }
99          }
100
101         protected Node tryPop() throws EmptyException
102         {
103             int[] stamp = new int[1];
104             Node oldTop = top.get(stamp);
105             if(oldTop == null)
106             {
107                 throw new EmptyException();
108             }
109             Node newTop = oldTop.next;
110             if(top.compareAndSet(oldTop, newTop, stamp[0], stamp[0] + 1))
111             {
112                 return oldTop;
113             }else
114             {
115                 return null;
116             }
117         }
118
119         public T pop() throws EmptyException
120         {
121             while(true)
122             {
123                 Node returnNode = tryPop();
124                 if(returnNode != null)
125                 {
126                     T value = returnNode.value;
127                     freeNode(returnNode);
128                     return value;
129                 }else
130                 {
131                     try
132                     {
```

```
133                     backoff.backoff();
134                 }catch(Exception e)
135                 {
136                     e.printStackTrace();
137                 }
138             }
139         }
140     }
141 }
```

129.

1. 意义在于push和pop都是在top上的冲突，所以他们共享同一个backoff才能比较好的减少冲突。

2. 根据：add additional backoff delays before accessing the shared stack, and control whether to access the shared stack or the array dynamically。在policy中增加关于tryPush()和tryPop()成功和失败的计数。假设成功次数为s，失败次数为f，则成功率为 s / (s + f)；则在tryPush()之前生成随机数r，如果 r % ((s + f) / s) == 0，则直接尝试tryPush()，否则直接从exchangers 入栈。

130.

将EliminationBackoffStack去掉tryPush和tryPop的部分，并将exchangers' capacity设为题目要求的界限。

131.

有类似于ABA的问题，比如：

1. top = 2, --> T1.pop() --> top = 1 --> T1.pop.i = 2, stack[2].value = value.T0

==> T2.pop() --> top = 0

==> T3.push() --> top = 1

==> T4.push() --> top = 2--> stack[2].value = value.T4; stack[2].full = true

==> T1.stack[2].full == true --> T1.pop() = value.T4

即T0写入的值被覆盖，丢失了。

2. top = 2 --> T1.pop --> top = 1 --> T1.i = 2

==> T2.push --> top = 2 --> T2.pushed

==> T3.pop --> top = 1, T3.i = 2

==> T3 && T1 pop the same item

问题在于push线程和pop线程之间没有同步，push线程并不知道当前操作的slot是否已经被pop了，即不能做到1对1。多个线程可能得到同一个index，因为取得index和读写操作并不原子，所以写/读的时候没有互斥；另外如果pop和push的速度太失衡，比如top.increase的速度总是赶不上pop.decrease的速度，会活锁。

[java] view plain copy

```
1   package p131;
2
3   import java.util.concurrent.atomic.AtomicInteger;
4   import java.util.concurrent.atomic.AtomicStampedReference;
5
6   import utils.EmptyException;
7   import utils.FullException;
8
9   public class DualStack
10  {
11      public class Slot
12      {
13          public static final int EMPTY = 0;
14          public static final int FULL = 1;
15          public static final int BUSY = 2;
16
17          volatile T value = null;
18          AtomicStampedReference state = new AtomicStampedReference(EMPTY, 0);
19      }
20
```

```
21      Slot[] stack;
22      int capacity;
23
24      private AtomicInteger top = new AtomicInteger(0);
25
26      @SuppressWarnings("unchecked")
27      public DualStack(int myCapacity)
28      {
29          this.capacity = myCapacity;
30
31          stack = new DualStack.Slot[capacity];
32  //      stack = (Slot[]) new Object[capacity];
33
34          for(int i = 0; i < capacity; i ++)
35          {
36              stack[i] = new Slot();
37          }
38      }
39
40      public void push(T value) throws FullException
41      {
42          while(true)
43          {
44              int i = top.getAndIncrement();
45              if(i > capacity - 1)
46              {
47                  top.compareAndSet(i, capacity);
48                  throw new FullException("" + i);
49              }else if(i >= 0)
50              {
51                  while(stack[i].state.getReference().intValue() != Slot.EMPTY) {}
52                  int[] stamp = new int[1];
53                  @SuppressWarnings("unused")
54                  int state = stack[i].state.get(stamp);
55                  if(!stack[i].state.compareAndSet(Slot.EMPTY, Slot.BUSY, stamp[0], stamp[0] + 1))
56                  {
57                      continue;
58                  }
59                  stack[i].value = value;
60                  stack[i].state.set(Slot.FULL, stamp[0] + 1);
61  //                System.out.println("stack " + i + " set full");
62                  return;
63              }
64          }
65      }
66
67      public T pop() throws EmptyException
68      {
69          while(true)
70          {
71              int i = top.getAndDecrement() - 1;
72              if(i < 0)
73              {
74                  top.compareAndSet(i + 1, 0);
75                  throw new EmptyException("" + (i + 1));
76              }else if(i < capacity)
```

```
77              {
78  //              System.out.println("Try to pop " + i);
79                  while(stack[i].state.getReference().intValue() != Slot.FULL) {}
80                  int[] stamp = new int[1];
81                  @SuppressWarnings("unused")
82                  int state = stack[i].state.get(stamp);
83                  T value = stack[i].value;
84                  if(!stack[i].state.compareAndSet(Slot.FULL, Slot.EMPTY, stamp[0], stamp[0]))
85                  {
86                      continue;
87                  }
88                  return value;
89              }
90          }
91      }
92
93      public int get()
94      {
95          return top.get();
96      }
97  }
```

132.

1. 有131题类似的问题；还有越界的问题。比如：

T1.pop, top = -1 --> T2.pop, top = -2 --> T3.push, top = -1, items[-1] = x;

2. 如下所示。Rooms保证在所有的线程在由push和pop工作的转换中是静态一致的，所以保证操作的i都是有效的。同时如果2个push线程得到同一个index，他们之间必然已经隔了一段pop的执行；而在这个pop之前，第一个push线程必然已经完成items[i]=x的操作。如果这个slot没有被pop，则在第二次push阶段因为i = top.getAndIncrement()也不会被覆盖。

[java] view plain copy

```
1   package p132;
2
3   import java.util.concurrent.atomic.AtomicInteger;
4
5   import p97.Rooms;
6   import utils.EmptyException;
7   import utils.FullException;
8
9   public class RoomsStack
10  {
11      private AtomicInteger top;
12      private T[] items;
13      private Rooms rooms;
14      public static final int PUSH = 0;
15      public static final int POP = 1;
16
17      @SuppressWarnings("unchecked")
18      public RoomsStack(int capacity)
19      {
20          top = new AtomicInteger(0);
21          items = (T[]) new Object[capacity];
22          rooms = new Rooms(2);
23      }
24
```

```
25      public void push(T x) throws FullException
26      {
27          rooms.enter(PUSH);
28          try
29          {
30              int i = top.getAndIncrement();
31              if(i >= items.length)
32              {
33                  top.getAndDecrement();
34                  throw new FullException();
35              }
36              items[i] = x;
37          }finally
38          {
39              rooms.exit();
40          }
41      }
42
43      public T pop() throws EmptyException
44      {
45          rooms.enter(POP);
46          try
47          {
48              int i = top.getAndDecrement();
49              if(i < 0)
50              {
51                  top.getAndIncrement();
52                  throw new EmptyException();
53              }
54              return items[i];
55          }finally
56          {
57              rooms.exit();
58          }
59      }
60  }
```

133.

[java] view plain copy

```
1   private class ExitHandler implements Rooms.Handler
2   {
3       public void onEmpty()
4       {
5           int newSize = items.length;
6           @SuppressWarnings("unchecked")
7           T[] newItems = (T[]) new Object[newSize];
8           for(int i = 0 ; i < items.length; i ++)
9           {
10              newItems[i] = items[i];
11          }
12          items = newItems;
13      }
14  }
```

```
15
16      @SuppressWarnings("unchecked")
17    public RoomsStack(int capacity)
18    {
19      top = new AtomicInteger(0);
20      items = (T[]) new Object[capacity];
21      rooms = new Rooms(2);
22      rooms.setExitHandler(PUSH, new ExitHandler());
23    }
24
25    public void push(T x)
26    {
27      while(true)
28      {
29        rooms.enter(PUSH);
30        int i = top.getAndIncrement();
31        if(i >= items.length)
32        {
33          top.getAndDecrement();
34          rooms.exit();
35        }else
36        {
37          items[i] = x;
38          return;
39        }
40      }
41    }
```

Chapter12

134.

135,136题我认为应该是这样，但是测试失败了，还在查。

135.

[java] view plain copy

```
1    package p135;
2
3    public class Node
4    {
5      enum CStatus{IDLE, FIRST, SECOND, THIRD, RESULT, ROOT};
6      boolean locked = false;
7      CStatus status;
8      int firstValue;
9      int secondValue;
10     int thirdValue;
11     int result;
12     Node parent;
13     int drained = 0;
14
15     public Node()
16     {
17       status = CStatus.ROOT;
18       parent = null;
19       locked =false;
20     }
```

```
21
22          public Node(Node parent)
23          {
24              this.parent = parent;
25              status = CStatus.IDLE;
26              locked = false;
27          }
28
29          synchronized CStatus precombine() throws InterruptedException, UnexpectedStatusException
30          {
31              while(drained == 2 || locked) wait();
32
33              switch(status)
34              {
35              //Return IDLE or FIRST is of no significance
36              case IDLE:
37                  status = CStatus.FIRST;
38                  return CStatus.IDLE;
39              case FIRST:
40                  drained = 1;
41                  status = CStatus.SECOND;
42                  return CStatus.SECOND;
43              case SECOND:
44                  drained = 2;
45                  status = CStatus.THIRD;
46                  return CStatus.THIRD;
47              case ROOT:
48                  return CStatus.ROOT;
49              default:
50                throw new UnexpectedStatusException();
51              }
52          }
53
54          synchronized int combine(int combined) throws InterruptedException, UnexpectedStatusException
55          {
56              while(drained > 0) wait();
57              locked = true;
58              firstValue = combined;
59
60              switch(status)
61              {
62              case FIRST:
63                  return firstValue;
64              case SECOND:
65                  return firstValue + secondValue;
66              case THIRD:
67                  return firstValue + secondValue + thirdValue;
68              default:
69                  throw new UnexpectedStatusException();
70              }
71          }
72
73          synchronized int op(int combined, CStatus myStatus) throws InterruptedException, UnexpectedStatusException
74          {
75              switch(myStatus)
76              {
```

```
77              case ROOT:
78                  int oldValue = result;
79                  result += combined;
80                  return oldValue;
81              case SECOND:
82                  secondValue = combined;
83                  drained = 0;
84                  notifyAll();
85                  while(status != CStatus.RESULT) wait();
86                  locked = false;
87                  notifyAll();
88                  status = CStatus.IDLE;
89                  return secondValue;
90              case THIRD:
91                  switch(myStatus)
92                  {
93                  case SECOND:
94                      secondValue = combined;
95                      break;
96                  case THIRD:
97                      thirdValue = combined;
98                      break;
99                  default:
100                   break;
101                 }
102                 if(--drained == 0)
103                 {
104                     notifyAll();
105                 }
106                 while(status != CStatus.RESULT) wait();
107                 locked = false;
108                 notifyAll();
109                 status = CStatus.IDLE;
110                 return thirdValue;
111             default:
112                 throw new UnexpectedStatusException();
113             }
114         }
115
116     synchronized void distribute(int prior) throws UnexpectedStatusException
117     {
118         switch(status)
119         {
120         case FIRST:
121             status = CStatus.IDLE;
122             locked = false;
123             break;
124         case SECOND:
125             secondValue = prior + firstValue;
126             status = CStatus.RESULT;
127             break;
128         case THIRD:
129             thirdValue = prior + firstValue + secondValue;
130             status = CStatus.RESULT;
131             break;
132         default:
```

```
133                 throw new UnexpectedStatusException();
134             }
135             notifyAll();
136         }
137     }
138 package p135;
139 
140 import java.util.Stack;
141 
142 import p135.Node.CStatus;
143 
144 public class Tree
145 {
146     Node[] leaf;
147     public ThreadLocal ThreadId = new ThreadLocal()
148     {
149         protected Integer initialValue()
150         {
151             return new Integer(0);
152         }
153     };
154 
155     public Tree(int y)
156     {
157         int size = ((int)Math.pow(3, (y + 1)) - 1) / (3 - 1);
158         Node[] nodes = new Node[size];
159         nodes[0] = new Node();
160         for(int i = 1; i < size; i ++)
161         {
162             nodes[i] = new Node(nodes[(i - 1) / 3]);
163         }
164 
165         int leafSize = (int)Math.pow(3, y);
166         leaf = new Node[leafSize];
167         for(int i = 0; i < leaf.length; i ++)
168         {
169             leaf[i] = nodes[nodes.length - i - 1];
170         }
171     }
172 
173     public int getAndIncrement()
174     {
175 //      System.out.println("Try to increment by " + ThreadId.get());
176         CStatus myStatus = CStatus.IDLE;
177         Stack stack = new Stack();
178         Node myLeaf = leaf[ThreadId.get()];
179         Node node = myLeaf;
180         int prior = 0;
181         try
182         {
183         while((myStatus = node.precombine()) == CStatus.IDLE)
184         {
185             node = node.parent;
186         }
187         Node stop = node;
188 
```

```
189             node = myLeaf;
190             int combined = 1;
191             while(node != stop)
192             {
193                 combined = node.combine(combined);
194                 stack.push(node);
195                 node = node.parent;
196             }
197
198             prior = node.op(combined, myStatus);
199
200             while(!stack.empty())
201             {
202                 node = stack.pop();
203                 node.distribute(prior);
204             }
205             }catch(Exception e)
206             {
207                 e.printStackTrace();
208             }
209
210             return prior;
211         }
212     }
213     package p135;
214
215     public class UnexpectedStatusException extends Exception
216     {
217         private static final long serialVersionUID = -5317568791061327777L;
218
219         public UnexpectedStatusException()
220         {
221             super();
222         }
223     }
```

136.

[java] view plain copy

```
1   package p136;
2
3
4   import java.util.concurrent.Exchanger;
5   import p135.UnexpectedStatusException;
6
7   /*
8    * Node.java
9    *
10   * Created on October 29, 2005, 8:59 AM
11   *
12   * From "The Art of Multiprocessor Programming",
13   * by Maurice Herlihy and Nir Shavit.
14   * Copyright 2006 Elsevier Inc. All rights reserved.
15   */
```

```
16
17  /**
18   * Node declaration for software combining tree.
19   * @author Maurice Herlihy
20   */
21  public class Node {
22    enum CStatus{IDLE, FIRST, SECOND, RESULT, ROOT};
23    boolean locked;   // is node locked?
24    CStatus cStatus;  // combining status
25    int firstValue, secondValue; // values to be combined
26    int result;       // result of combining
27    Node parent;      // reference to parent
28    Exchanger sendUp;
29    Exchanger sendDown;
30    int index;
31
32    /** Creates a root Node */
33    public Node() {
34      cStatus = CStatus.ROOT;
35      locked = false;
36      index = 0;
37    }
38    /** Create a non-root Node */
39    public Node(Node _parent, int index) {
40      parent = _parent;
41      cStatus = CStatus.IDLE;
42      locked = false;
43      sendUp = new Exchanger();
44      sendDown = new Exchanger();
45      this.index = index;
46    }
47
48    synchronized boolean precombine() throws InterruptedException, UnexpectedStatusException
49    {
50      while (locked) wait();
51      switch (cStatus) {
52        case IDLE:
53          cStatus = CStatus.FIRST;
54          return true;
55        case FIRST:
56  //        locked = true;
57          cStatus = CStatus.SECOND;
58          System.out.println("SECOND in precombine");
59          return false;
60        case ROOT:
61          return false;
62        default:
63          throw new UnexpectedStatusException();
64      }
65    }
66
67    synchronized int combine(int combined) throws InterruptedException, UnexpectedStatusException
68    {
69  //    while (locked) wait();
70      locked = true;
71      firstValue = combined;
```

```
72        switch (cStatus) {
73          case FIRST:
74            return firstValue;
75          case SECOND:
76              System.out.println("SECOND in combine");
77            int rc = firstValue + sendUp.exchange(null);
78            System.out.println("SECOND after combine");
79            return rc;
80          default:
81            throw new UnexpectedStatusException();
82        }
83      }
84
85      synchronized int op(int combined) throws InterruptedException, UnexpectedStatusException
86      {
87        switch (cStatus) {
88          case ROOT:
89            int oldValue = result;
90            result += combined;
91            return oldValue;
92          case SECOND:
93      //        secondValue = combined;
94              System.out.println("SECOND before exchange up");
95
96            sendUp.exchange(combined);
97      //        locked = false;
98      //        notifyAll();
99      //        while (cStatus != CStatus.RESULT) wait();
100           System.out.println("SECOND before exchange down");
101           result = sendDown.exchange(null);
102           locked = false;
103           cStatus = CStatus.IDLE;
104           notifyAll();
105           return result;
106         default:
107           throw new UnexpectedStatusException();
108       }
109     }
110     synchronized void distribute(int prior) throws InterruptedException, UnexpectedStatusException
111     {
112       switch (cStatus) {
113         case FIRST:
114           cStatus = CStatus.IDLE;
115           locked = false;
116           break;
117         case SECOND:
118     //        result = prior + firstValue;
119           sendDown.exchange(prior + firstValue);
120           cStatus = CStatus.RESULT;
121           break;
122         default:
123           throw new UnexpectedStatusException();
124       }
125     //    notifyAll();
126     }
127
```

```
128     public int get()
129     {
130         return index;
131     }
132 
133 }
```

Refer for 135/136

137.
同queue

138.
139.

[java] view plain copy

```
1   package p139;
2   
3   import java.util.concurrent.atomic.AtomicInteger;
4   
5   public class Balancer
6   {
7       AtomicInteger counter;
8   
9       public Balancer()
10      {
11          counter = new AtomicInteger(0);
12      }
13  
14      public int traverse()
15      {
16          int count = counter.getAndIncrement();
17  
18          if(count % 2 == 0)
19          {
20              return 0;
21          }else
22          {
23              return 1;
24          }
25      }
26  }
```

140.
采用归纳法证明：假设 T[k]有布进特性，用T[k]构成T[2k]网后，进入 T[0][k]和进入T[1][k]的令牌最多相差1个。则根据引理12.5.1.2，另T[0].j = c，则T[1].j = c || T[1].j = c - 1
case 1: T[0].j = c && T[1].j = c && (c % 2) == 0
令T[0][i] = a (i <= j)，则T[1][i] = a (i <= j)。经过输出线重定向得到：T[1] = T[0][1] = a, T[2] = T[1][1] = a, ... T[(c - 1) * 2] = T[0][c - 1] = a, T[c * 2] = T[1][c - 1] = a, ... , T[i] = (a - 1)..., T[2k] = (a - 1)。
case 2: T[0].j = c && T[1].j = (c - 1) && (c % 2) == 0
case 3: T[0].j = c && T[1].j = c && (c % 2) == 1
case 4: T[0].j = c && T[1].j = (c - 1) && (c % 2) == 1
case 5: j 不存在

都做类似证明。

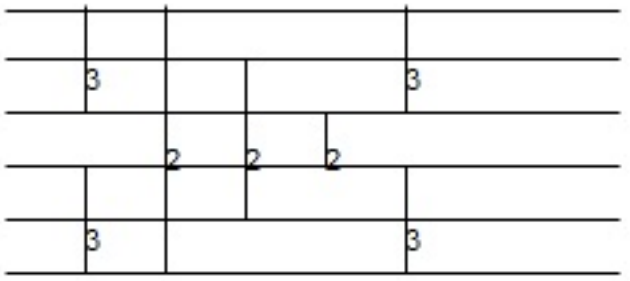
141.
用归纳法证明：
假设 m = 2 ^ (d - 1)个令牌从一个输出线上穿过深度为d - 1的网络后B在令牌离开网络后的状态与进入前的状态相同，则第m+1个令牌通过d-1层网络后到达的第d层的输入线与第1个令牌到达输入线相同。如果第d层的这个balancer的状态为1（被翻转），则第m+1个令牌将它的状态恢复，如果状态为0（已经被恢复），则m+2 ~ n个令牌中的某些会像1 ~m个令牌的行为一样将其恢复。

142.
令X的输出为x，Y的输出为y；min(x) = ux, min(y) = uy;
经过匹配后的Z的输出为z，则 max(z) = ceil(((max(x) + max(y)) / 2), min(z) = floor(((min(x) + min(y)) / 2)
==> max(z) - min(z) = ceil(((max(x) + max(y)) / 2) - floor((ux + uy) / 2)
<= ceil((ux + k + uy + k) / 2) - floor((ux + uy) / 2)
= ceil((ux + uy) / 2 + k) - floor((ux + uy) / 2) <= k + 1

143.
Block[k]的深度为logk，每一个深度对应一个Balancer，每一个balancer都是1-光滑的，经过logk层后，有142知是logk-光滑的

144.
因为S是光滑网，所以令从S的输出线上输出的令牌数为k或者k+1，输出线上的前k个令牌经过P后输出仍然是每线输出为k，平衡。剩下的令牌分散在P的输入线上，有的输入线输入为1（个令牌），有的输入线输入为0；因为P是布尔排序网的同构网，所以可以对1/0输入进行排序，即值为1的输入将在值为0的输入之前输出；将输入为1与有令牌输入对应，输入为0与无令牌输入对应；即P将对剩下的令牌排序。

145.



如图所示。

146.
令n为距离w最近而且>w的2的幂，构建宽度为n的BitomicSort，将w与n的gap用最小值补齐

147.
1.以设置filter[v]为可线性化点，题目中说了，每个线程等待其它具有较小值的线程赶上，设置自己的filter相，然后返回。即可以根据v值得到全序，并且v值较小的在前。等待比自己小的线程到达实际上是等待自己前一个线程将filter[v-1]值翻转。如果翻转，表示确实等到同一轮v-1的到来。如果已经翻转，但不是同一轮的v-1，则翻转这个值的必然是v-1+2*w或者v-1-2*w。如果由v-1-2*w翻转的，则表示v-1-w线程还没有到达，则v-w线程不能返回，即v线程并不能开始，矛盾；所以不可能是v-1-2*w线程翻转的，类似的也不是v-1+2*w线程翻转的。
2.
3. 有更高的吞吐量。首先filter网采用spin last anticipator的方案，类似于CLH队列锁，在SMP上比所有线程spin在同一个变量上效率高；而且在进入filter之前先经过吞吐量由于单个锁的排序网，缩短了在前一个变量上spin的时间，减小了需要等待的几率。

148.
首先证明k-光滑的X穿过深度为1的平衡网后仍然是k-光滑的。
假设x[i]穿过了平衡器，则必然与某一个x[j]平衡，得到的输出y[i]=ceil((x[i] + x[j]) / 2), y[j] = floor((x[i] + x[j]) / 2)

==> max(X) >= y[i] >= y[j] >= min(X)
否则如果x[i]没有经过平衡器，则仍然有
==> max(X) >= y[i] >= min(X)
所以有任何 |y[k] - y[i]| <= k，即仍然是k-光滑序列。
然后可以用归纳法证明，如果X经过了深度为d的平衡网仍然是k-光滑的，则经过d+1的平衡网后也是k-光滑的。

149.
1. 因为M(2w) = M(w) + 1 ==> M(w) = log(w) + 1;
又B(2w) = B(w) + M(2w)
= log(w) * (log(w) + 1) / 2 + log(w) + 1
= (log(w) * log(w) + 3log(w) + 2) / 2
= (1 + log(w)) * (2 + log(w)) / 2
= log(2*w) * (log(2 * w) + 1) / 2
= B(2w)
2.
M(2w) = 2 * M(w) + 2 ==> M(w) = w * log(w) / 2
B(2w) = 2 * B(w) + M(2w)
= (2*w*log(w) 2 * w * (log(w) ^ 2)) / 4 + wlog(w) + w
= (2 * w * log(2w)) * (1 + log(2w)) / 2
归纳法得证。

150.
如果Balancer是无锁的，DiffractingBalancer就是无锁的?

151.
类似并发栈的消除策略：prism成功的次数增加则增加slot的范围，失败的次数增加则减小slot的范围，当slot减小到一定程度可以取消prism，直接走balancer。

152.
一个反例：
3个token从1线输入它们的路径分别是
1->1->1
1->2->2
1->1->1

153.
假设深度为d的网络N满足这个性质，令在后面加一层的网络为N'。则第 (2 ^ d) + 1个令牌通过N后与第1个令牌的输出线相同，当它通过第d+1层的平衡器时，无论令牌是否反令牌，(2 ^ d) + 1个令牌都将恢复第1个令牌改变的第d + 1层平衡器。所以经过 (2 ^ d) * 2个令牌，即(2 ^ (d + 1))个令牌后 N'也将恢复成初始状态。

154.
因为反令牌的行为是翻转toggle并返回未翻转的值，所以这个反令牌必然跟踪原来的令牌所走的相同路线并将令牌翻转过的平衡器恢复。

155.
一个反令牌根据平衡器的当前值移动，并恢复平衡器的状态，而且权重为-1。因此可以将一个反令牌的行为看成一个令牌的行为的反：一个反令牌之所以经过某一条路径而不是另一条是因为有一个令牌已经经过了这条路径（这个令牌的动作可能发生在反令牌通过网络之后）。即一个反令牌在网络达到静态后可以抵消某个令牌。所以如果有a个令牌和b各反令牌通过了网络，其行为相当于a - b个令牌经过了这个网络，由于网络是计数网，网络将对a - b个令牌平衡，即对输出线上的令牌和反令牌权重和平衡，符合有反令牌的平衡网的定义。

156.
Adding Networks
这篇论文是本书作者关于adding network的论文，比题目的描述要详细。
大意是如果按照题目描述的token的顺序和行为，t将在adding-network的出线返回a，t1返回a + b，t2返回a + 2*b ...
本书题目中说的ti可以在S的某个交换器上终结，意思是ti必然经过S中的某个交换器。

论文的第7页给了正式的证明，没有完全看懂，求指教。
大概的意思是用归纳法：
令s为S中的任一交换器。
起始：
因为t1要返回a + b，t1的路径必须经过s：因为adding-network并不能产生数值和token，如果t1不经过s，a不能凭空产生。
归纳：反证
如果ti在t之后穿过adding-network，ti在出线上应该得到a + i * b。
但是因为ti经过的路径没有经过s，整个交换网在 t经过然后（t1, t2, ..., ti）经过 与 只有(t1, t2, ..., ti)然后t经过 没有区别。但是前者ti的出线值是a + i * b；后者的出线值是 i * b。无法区分。
所以ti必然经过s
得证。

157.
计数网是加法网的一种，即每次加a = b = 1的加法网，所以深度也至少为n。

Chapter13

158.

[java] view plain copy

```
1    public class StripedHashSet extends BaseHashSet
2    {
3        final ReadWriteLock[] locks;
4        //...
5
6        public final void acquire(T x)
7        {
8            locks[x.hashCode() % locks.length].readLock().lock();
9        }
10
11       public void release(T x)
12       {
13           locks[x.hashCode() % locks.length].readLock().unlock();
14       }
15
16       public void resize()
17       {
18           for(Lock lock : locks)
19           {
20               lock.writeLock().lock();
21           }
22
23           //...
24
25           for(Lock lock: locks)
26           {
27               lock.writeLock().unlock();
28           }
29       }
30   }
```

159.
因为bucket要为hashset提供shortcut，但是对list的remove和对bucket的值remove不能原子的实现，所以会有问题。
如果先从list中删除，再处理bucket指向，则通过bucket访问hashset的线程有可能得到已经被删除的节点；

如果先处理bucket指向，再处理list，则考虑这样的case：
bucket = a，在list中 a-->b;
线程A remove a，首先找到b，并认为b合法，应该有bucket=b;
线程B remove b，没有corner case, remove b结束;
线程A 令bucket=b, 在list中remove a，结束。

160.
table的个数为N，并且pow（2，i）=N。当N=2，bucket的个数从一个变成两个的时候就是要递归的初始化O(logN)个父桶。由于是pow（2，i）=N，求对数的话预期程度是常数。
161.
如果允许一个比较大的但是flexible的数据结构，可以用14章介绍的SkipList。
如果要求任意大小，因为resize只考虑增加不考虑减少，所以可以采用这样的二叉树：

[java] view plain copy

```
1   package p161;
2
3   import java.util.BitSet;
4   import java.util.concurrent.atomic.AtomicReference;
5
6
7   //First bit of key(BitSet) set as 1 to calculate length of bitset
8   public class Node
9   {
10     private final int key;
11     private final BitSet bits;
12     private AtomicReference> left;
13     private AtomicReference> right;
14
15     private AtomicReference sentinal;
16
17     private void init()
18     {
19        left = new AtomicReference>(null);
20        right = new AtomicReference>(null);
21        sentinal = new AtomicReference(null);
22     }
23
24     private Node(int key, BitSet bits)
25     {
26        this.key = key;
27        this.bits = bits;
28
29        init();
30     }
31
32
33     public Node()
34     {
35        this.key = 0;
36        this.bits = new BitSet(1);
37        this.bits.set(0, true);
38
39        init();
40     }
41
```

```
42      private Node createNode(boolean rChild)
43      {
44          int bitLen = bits.length();
45          int newValue = rChild? 1: 0;
46          BitSet newBits = new BitSet(bitLen + 1);
47          int i = 0;
48          for(; i < bitLen - 1; i ++)
49          {
50              newBits.set(i, bits.get(i));
51          }
52          newBits.set(i, rChild);
53          newBits.set(i + 1, true);
54
55          int newKey = key | (newValue << (bitLen - 1));
56
57          return new Node(newKey, newBits);
58      }
59
60      public Node getChild(boolean isRight)
61      {
62          AtomicReference> child = isRight? right: left;
63
64          if(child.get() != null)
65          {
66              return child.get();
67          }else
68          {
69              Node newChild  = createNode(isRight);
70
71              child.compareAndSet(null, newChild);
72              return child.get();
73          }
74      }
75
76      public static BitSet IntToBits(int key)
77      {
78          long MASK = Long.MAX_VALUE;
79          int BITMASK = 1;
80          int index = 0;
81
82          //64bits enough for int
83          BitSet bits = new BitSet();
84          while((key & MASK) != 0)
85          {
86              boolean keyBit = (key & BITMASK) == 0? false: true;
87              bits.set(index, keyBit);
88
89              MASK <<= 1;
90              BITMASK <<= 1;
91              index ++;
92          }
93
94          bits.set(index, true);
95
96          return bits;
97      }
```

```
98
99      public static int BitsToInt(BitSet bits)
100     {
101        int key = 0;
102        for(int i = 0; i < bits.length() - 1; i ++)
103        {
104           int bitV = bits.get(i)? 1: 0;
105           key |= (bitV << i);
106        }
107
108        return key;
109     }
110
111     public Node getLeft()
112     {
113        return getChild(false);
114     }
115
116     public Node getRight()
117     {
118        return getChild(true);
119     }
120
121     public Node getLeftChild()
122     {
123        return left.get();
124     }
125
126     public Node getRightChild()
127     {
128        return right.get();
129     }
130
131     public int getKey()
132     {
133        return key;
134     }
135
136     public T getSentinal()
137     {
138        return sentinal.get();
139     }
140
141     public void setSentinal(T x)
142     {
143        sentinal.set(x);;
144     }
145
146     public BitSet getKeySet()
147     {
148        return bits;
149     }
150  }
151
152  package p161;
153
```

```
154
155
156 import java.util.BitSet;
157
158 public class LocklessBinaryTree
159 {
160    private Node root;
161
162    public LocklessBinaryTree(T initV)
163    {
164       root = new Node();
165       root.setSentinal(initV);
166    }
167
168    public Node getRoot()
169    {
170       return root;
171    }
172
173    public Node getNearestAncestor(BitSet key)
174    {
175       int len = key.length();
176
177       Node node = root;
178       Node child = node;
179       for(int i = 0; i < len - 1; i ++)
180       {
181          if(key.get(i))
182          {
183             child = node.getRight();
184          }else
185          {
186             child = node.getLeft();
187          }
188
189          //Make sure parent has sentinal set
190          if(child == null || child.getSentinal() == null)
191          {
192             return node;
193          }else
194          {
195             node = child;
196          }
197       }
198
199       return node;
200    }
201
202    public Node getChild(Node parent, boolean isRight)
203    {
204       if(parent == null)
205       {
206          return null;
207       }
208
209       Node node = parent.getChild(isRight);
```

```
210             return node;
211         }
212     }
```

[java] view plain copy

```
1   package p161;
2
3
4   /*
5    * LockFreeHashSet.java
6    *
7    * Created on December 30, 2005, 12:48 AM
8    *
9    * From "Multiprocessor Synchronization and Concurrent Data Structures",
10   * by Maurice Herlihy and Nir Shavit.
11   * Copyright 2006 Elsevier Inc. All rights reserved.
12   */
13
14
15  //import ch13.Hash.src.hash.*;
16
17
18  import java.util.BitSet;
19  import java.util.concurrent.atomic.AtomicInteger;
20
21  /**
22   * @param T item type
23   * @author Maurice Herlihy
24   */
25  public class LockFreeHashSet {
26
27    protected LocklessBinaryTree > tree;
28
29    protected AtomicInteger bucketSize;
30    protected AtomicInteger setSize;
31    private static final double THRESHOLD = 4.0;
32
33    public LockFreeHashSet()
34    {
35      BucketList list = new BucketList();
36      tree = new LocklessBinaryTree >(list);
37      bucketSize = new AtomicInteger(2);
38      setSize = new AtomicInteger(0);
39    }
40
41
42    public boolean add(T x) {
43      int myBucket = Math.abs(BucketList.hashCode(x) % bucketSize.get());
44      BucketList b = getBucketList(myBucket);
45      if (!b.add(x))
46        return false;
47      int setSizeNow = setSize.getAndIncrement();
48      int bucketSizeNow = bucketSize.get();
49      if (setSizeNow / (double)bucketSizeNow > THRESHOLD)
50        bucketSize.compareAndSet(bucketSizeNow, 2 * bucketSizeNow);
```

```
51        return true;
52      }
53
54      public boolean remove(T x) {
55        int myBucket = Math.abs(BucketList.hashCode(x) % bucketSize.get());
56        BucketList b = getBucketList(myBucket);
57        if (!b.remove(x)) {
58          return false;    // she's not there
59        }
60        return true;
61      }
62
63      public boolean contains(T x) {
64        int myBucket = Math.abs(BucketList.hashCode(x) % bucketSize.get());
65        BucketList b = getBucketList(myBucket);
66        return b.contains(x);
67      }
68
69      private BucketList getBucketList(int myBucket)
70      {
71          BitSet bits = Node.IntToBits(myBucket);
72          Node> parent = tree.getNearestAncestor(bits);
73
74          if(parent.getKey() == myBucket)
75          {
76              return parent.getSentinal();
77          }
78
79          BitSet parentBits = parent.getKeySet();
80
81          int index = parentBits.length() - 1;
82          Node> node = null;
83          for(; index < (bits.length() - 1); index ++)
84          {
85              node = tree.getChild(parent, bits.get(index));
86              if(node.getSentinal() == null)
87              {
88                  BucketList b = parent.getSentinal().getSentinel(node.getKey());
89
90                  if(b != null) //Is that possible?
91                  {
92                      node.setSentinal(b);
93                  }
94              }
95
96              parent = node;
97          }
98
99          return node.getSentinal();
100     }
101
102 }
```

Chapter14

163.

每个节点有相同的高度，并且是一次失败的查找。

164.
N * pow(p, n)

165.

[java] view plain copy

```
1   public final class LazySkipList
2   {
3       static final int MAX_LEVEL = ...;
4
5       int[] levelCounters;
6       volatile int highestLevel;
7       Lock levelCounterLock;
8       ...
9
10      public LazySkipList()
11      {
12          ...
13          levelCounters = new int[MAX_LEVEL + 1];
14          levelCounterLock = new ReentrantLock();
15          highestLevel = 0;
16      }
17
18      private int updateHighestLevel(boolean addAction, int updatedLevel)
19      {
20          levelCounterLock.lock();
21          try
22          {
23              if(addAction)
24              {
25                  levelCounters[updatedLevel] ++;
26                  if(updatedLevel > highestLevel)
27                  {
28                      highestLevel = updatedLevel;
29                  }
30              }else
31              {
32                  levelCounters[updatedLevel] --;
33
34                  if(updatedLevel == highestLevel
35                          && levelCounters[updatedLevel] == 0)
36                  {
37                      for(int i = highestLevel - 1; i >= 0; i --)
38                      {
39                          if(levelCounters[i] > 0)
40                          {
41                              highestLevel = i;
42                              break;
43                          }
44                      }
45                  }
46              }
47          }finally
```

```
48          {
49              levelCounterLock.unlock();
50          }
51      }
52
53      int find(T x, Node[] preds, Node[] succs)
54      {
55          ...
56          //No error if highestLevel is being upated.
57          //If highestLevel raised, current highestLevel can cover the subset
58          //If highestLevel lowered, find() reaches TAIL on upper layers
59          for(int level = highestLevel; level >= 0; level --)
60          {
61              ...
62          }
63          ...
64      }
65
66      boolean add(T x)
67      {
68          ...
69          newNode.fullyLinked = true;
70          updateHighestLevel(true, topLevel);
71          return true;
72          ...
73      }
74
75      boolean remove(T x)
76      {
77          ...
78          victim.lock.unlock();
79          updateHighestLevel(false, victim.topLevel);
80          return true;
81          ...
82      }
83  }
```

166.
key可以不唯一，item唯一，因为add/remove/find都是以x(item)为参数。
所以find方法改为找到该节点 或者 找到该key值的末尾。
add方法也不用改变，因为add是从0层开始链入；如果2个add试图以同一个节点为尾节点加入新节点，则肯定有一个add会抢锁失败然后valid失败；如果以不同的节点为尾节点，表示某一个尾节点的next已经被改变，valid也会失败。如果一个节点链入了0层，则另一个的find会失败。

[java] view plain copy

```
1   int find(T x, Node[] preds, Node[] succs)
2   {
3       int key = x.hashCode();
4       int lFound = -1;
5       Node pred = head;
6
7       for(int level = MAX_LEVEL; level >= 0; level --)
8       {
9           Node curr = pred.next[level];
10
```

```
11          while(key > curr.key || (curr.key == key && curr.item != x))
12          {
13              pred = curr;
14              curr = curr.next[level];
15          }
16
17
18          if(lFound == -1 && key == curr.key && curr.item == x)
19          {
20              lFound = level;
21          }
22
23          preds[level] = pred;
24          succs[level] = curr;
25      }
26
27      return lFound;
28  }
```

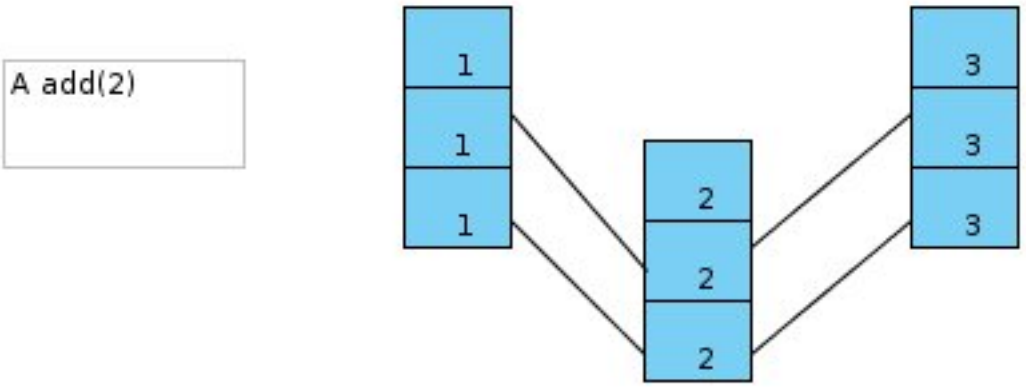
167.

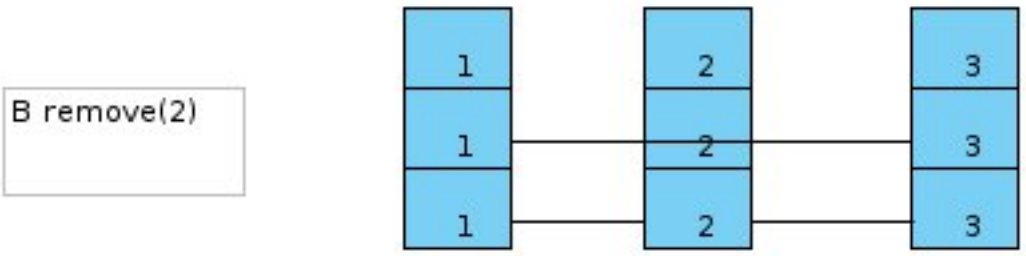
这个方法不成功的可线性化点在 find返回 false。

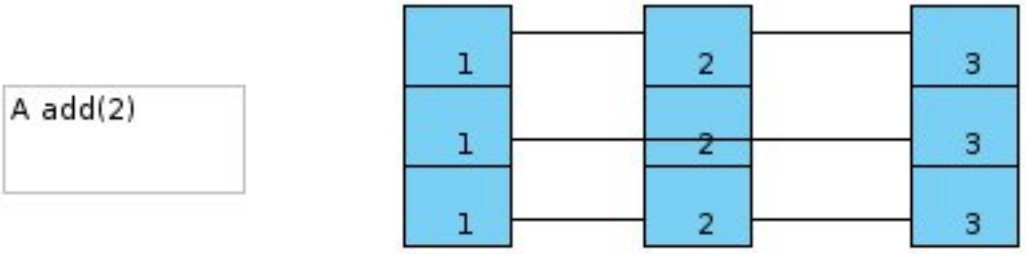
这仍然是一个无锁算法。根据无锁的定义：it guarantees that infinitely often some method call finishes in a finite number of steps. 即无数次的方法调用中总有在有限步内完成的。如果无数次的方法调用都没有能够返回的调用，对应的case是 find总能找到并行add的可删除节点，但是总是标记失败且发现被删除节点已经被标记。即失败的调用者总是对应一个成功的将被删除节点标记并返回的调用。符合lockless的调用。

168.

这只是一种临时状态。







169.

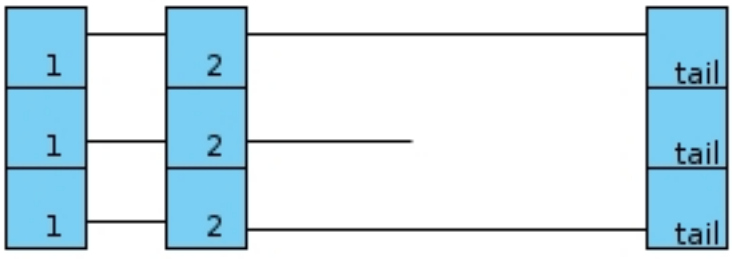
[java] view plain copy

```
1     public boolean find(T x, Node[] preds, Node[] succs)
2     {
3       int bottomLevel = 0;
4       int key = x.hashCode();
5       boolean[] marked = {false};
6       boolean snip;
7       Node pred = null, curr = null, succ = null;
8       Node origCurr = null;
9
10      retry:
11        while(true)
```

```
12          {
13              pred = head;
14              for(int level = MAX_LEVEL; level >= bottomLevel; level --)
15              {
16                  curr = pred.next[level].getReference();
17
18                  while(true)
19                  {
20                      origCurr = curr;
21                      succ = curr.next[level].get(marked);
22                      while(marked[0])
23                      {
24                          curr = succ;
25                          succ = curr.next[level].get(marked);
26                      }
27                      snip = pred.next[level].compareAndSet(origCurr, curr, false, false);
28                      if(!snip)
29                      {
30                          continue retry;
31                      }
32
33                      if(curr.key < key)
34                      {
35                          pred = curr;
36                          curr = succ;
37                      }else
38                      {
39                          break;
40                      }
41                  }
42
43                  preds[level] = pred;
44                  succs[level] = curr;
45              }
46
47              return (curr.key == key);
48          }
49      }
```

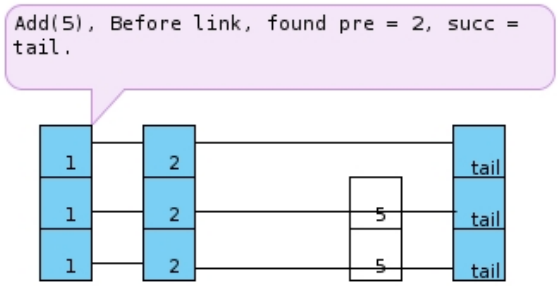
求例子。

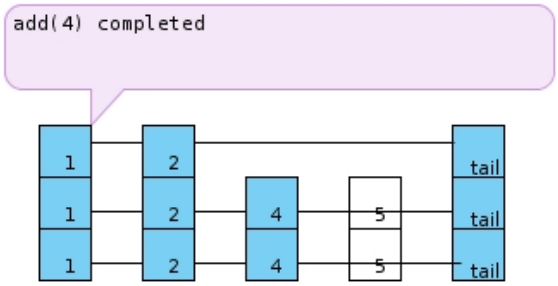
170.
Add: 如图所示，find(x)将会得到succ = null；

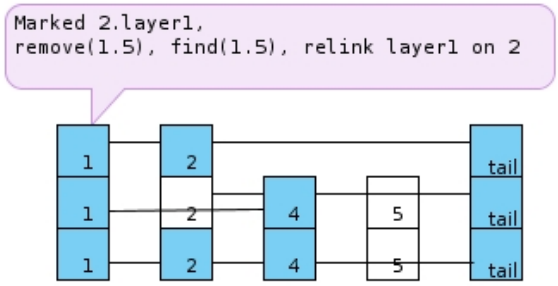


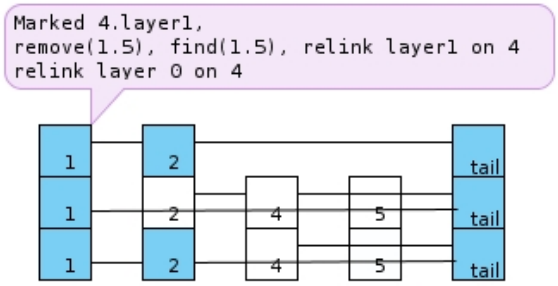
Remove： 如图所示，如果这时候调用contain，将得到错误的结果；或者任何值 > 2的update操作，都会在find的时候在

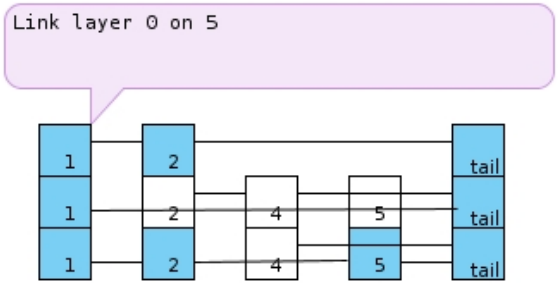
retry --> sip = pred.next[level].compareAndSet(curr, succ, false, false) 死循环，因为2.layer1.marked = true。

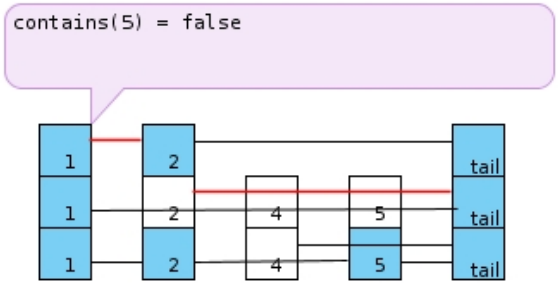






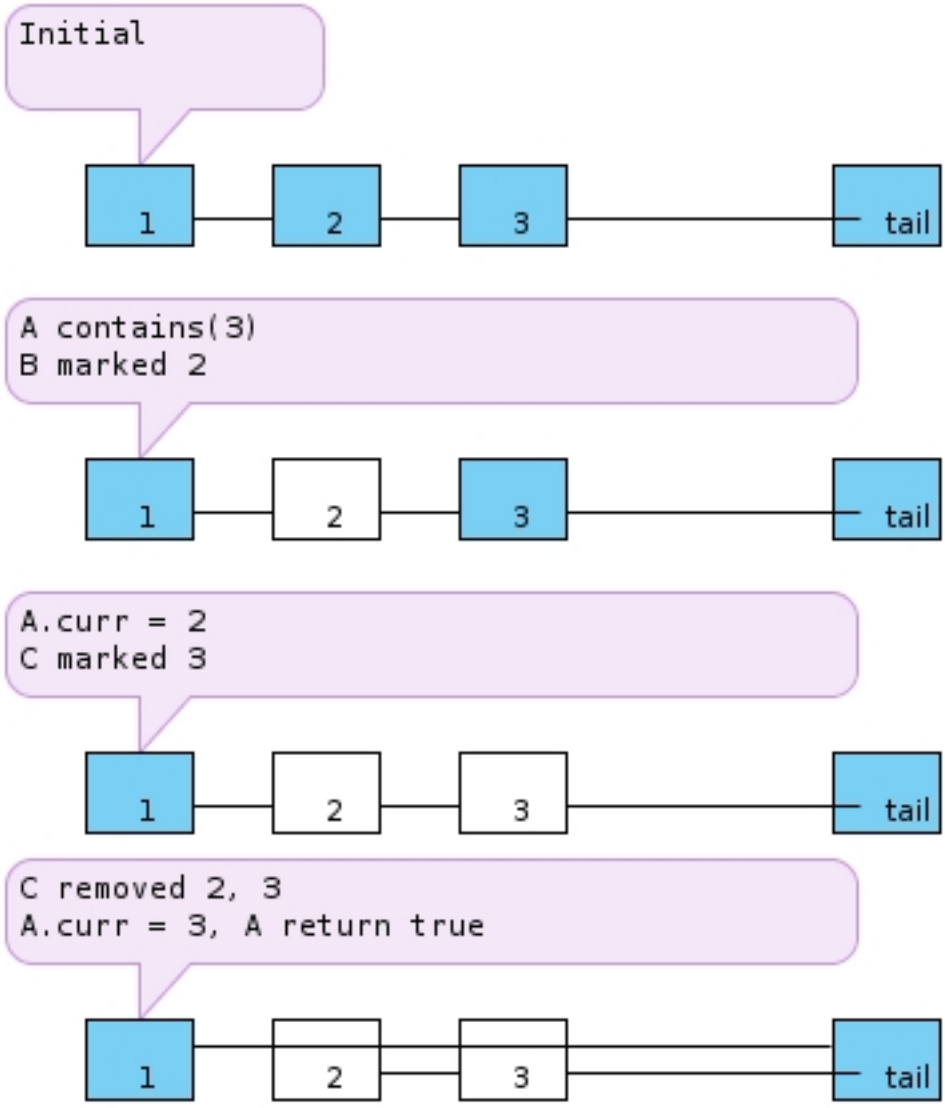






171.

172.



Chapter15

173. SimpleTree 是不可线性化但是静态一致的。

174. 思路是
如果不要求保存状态可重用，则可以在输出端连接H个开关，比如说，如果output=1已经有token经过了，则当前counter>H，返回H。
如果考虑可重用，则发现>H则再加入一个反令牌，<0则再加入一个令牌。一个问题是如何在decrement用反令牌实现的前提下用开关标记是否是超过H的。
因为衍射树是静态一致的，AtomicBoolean也是静态一致的，他们的组合也是静态一致的。

[java] view plain copy

```
1   package p174;
2
3   public class Balancer {
4       Boolean toggle;
5
6       public Balancer()
7       {
8        toggle = true;
```

```
9         }
10
11        public synchronized int traverse(boolean token)
12        {
13           if(token)
14           {
15              try
16              {
17                 if (toggle)
18                 {
19                    return 0;
20                 } else
21                 {
22                    return 1;
23                 }
24              } finally
25              {
26                 toggle = !toggle;
27              }
28           }else
29           {
30              toggle = !toggle;
31              if(toggle)
32              {
33                 return 0;
34              }else
35              {
36                 return 1;
37              }
38           }
39        }
40     }
41
42
43
44
45     package p174;
46
47     public class DiffractingTree
48     {
49        Balancer root;
50        DiffractingTree[] child;
51        int size;
52
53        public DiffractingTree(int mySize)
54        {
55           size = mySize;
56           root = new Balancer();
57           if(size > 2)
58           {
59              child = new DiffractingTree[]
60                 {
61                    new DiffractingTree(size / 2),
62                    new DiffractingTree(size / 2),
63                 };
64           }
```

```
65        }
66
67        public int traverse(boolean token)
68        {
69            int half = root.traverse(token);
70
71            if(size > 2)
72            {
73                return (2 * (child[half].traverse(token)) + half);
74            }else
75            {
76                return half;
77            }
78        }
79    }
80
81
82    package p174;
83
84    import java.util.concurrent.atomic.AtomicBoolean;
85
86    public class Counter
87    {
88        private final int bound;
89        private DiffractingTree tree;
90        private AtomicBoolean flags[];
91
92        public Counter(int bound)
93        {
94            this.bound = bound;
95            tree = new DiffractingTree(bound);
96            flags = new AtomicBoolean[bound];
97
98            for(int i = 0; i < flags.length; i ++)
99            {
100               flags[i] = new AtomicBoolean(false);
101           }
102       }
103
104       public int boundedGetAndIncrement()
105       {
106           int tmpRc = tree.traverse(true);
107           if(flags[tmpRc].compareAndSet(false, true))
108           {
109               return tmpRc;
110           }else
111           {
112               tmpRc = tree.traverse(false);
113               flags[tmpRc].compareAndSet(true, false);
114               return bound;
115           }
116       }
117
118       public int boundedGetAndDecrement()
119       {
120           int tmpRc = tree.traverse(false);
```

```
121
122             if(flags[tmpRc].compareAndSet(true, false))
123             {
124                 return tmpRc;
125             }else
126             {
127                 tmpRc = tree.traverse(true);
128                 flags[tmpRc].compareAndSet(false, true);
129                 return 0;
130             }
131         }
132
133         public static void main(String[] args)
134         {
135             Counter counter = new Counter(4);
136
137             for(int i = 0; i < 2; i ++)
138             {
139                 int rc = counter.boundedGetAndIncrement();
140                 System.out.println("" + i + "---------> " + rc);
141             }
142
143             for(int i = 0; i < 9; i ++)
144             {
145                 int rc = counter.boundedGetAndDecrement();
146                 System.out.println("-" + i + "---------> " + rc);
147             }
148         }
149     }
```

175.
负数，非零，走到错误的叶子。

176. 看不懂题目。这个本来就是有界容量的。

177.
removeMin超过add，在内部节点中发现无路可走。

178.

179.

[java] view plain copy

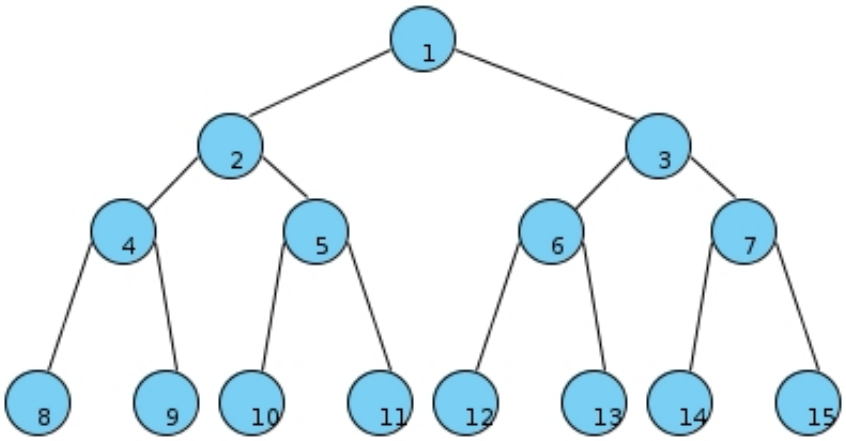
```
1   import java.util.ArrayList;
2   import java.util.concurrent.locks.Lock;
3   import java.util.concurrent.locks.ReentrantLock;
4
5   /**
6    * Heap with fine-grained locking and arbitrary priorities.
7    * @param T type manged by heap
8    * @author mph
9    */
10  public class FineGrainedHeap implements PQueue {
```

```
11
12      private static int ROOT = 1;
13      private static int NO_ONE = -1;
14      private Lock heapLock;
15      int next;
16     //  HeapNode[] heap;
17      ArrayList> heap;
18
19      /**
20       * Constructor
21       * @param capacity maximum number of items heap can hold
22       */
23      public FineGrainedHeap(int capacity) {
24        heapLock = new ReentrantLock();
25        next = ROOT;
26
27        heap = new ArrayList>(capacity + 1);
28
29      }
30
31      /**
32       * Add item to heap.
33       * @param item Uninterpreted item.
34       * @param priority item priority
35       */
36      public void add(T item, int priority) {
37        heapLock.lock();
38        int child = next++;
39
40        if(heap.size() <= child)
41        {
42           HeapNode childElem = new HeapNode ();
43           heap.add(childElem);
44        }
45
46        //It should be right as next ++, add new element are sequential as they were locked by helpLock.
47        //And there is no removal operation from list.
48        //Anyway, the get(child) can throw Exception when index and element really mismatched.
49        heap.get(child).lock();
50        heapLock.unlock();
51        //...
52      }
53
54      //...
55    }
```

临时创建对象的开销，尤其是在全局锁中创建对象的开销。

180.
好处在于如果没有这个特性，相邻的2个插入在时间上是相差最短的，而且最容易有相同的父节点，所以产生同步开销的机率最大。

以 reverseIncrement为例：




```
1--------> 1
2--------> 10
3--------> 11
4--------> 100
5--------> 110
6--------> 101
7--------> 111
8--------> 1000
9--------> 1100
10--------> 1010
11--------> 1110
12--------> 1001
13--------> 1101
14--------> 1011
15--------> 1111
```

[java] view plain copy

```java
public int reversedIncrement()
{
    //如果counter越界，则归零
    if(count ++ == 0)
    {
        reverse = highBit = 1;
        return reverse;
    }

    //取bit为除了最高位的第一位。即根结点下的第一层子树位。
    int bit = highBit >> 1;

    while(bit != 0)
    {
        //从高位开始异或，所以相邻的2个节点必然不在同一个子树上。
        reverse ^= bit;

        /*
        * 因为reverse ^= bit，又（reverse & bit == 0）
        * ==> 假设bit = 0100,则异或之前reverse = x0xx
        * ==> 原来的reverse在左树上，则找到右子树上的对应节点，退出。
        * ==> 如果原来的reverse在右树上，则异或后转到左子树
        * ==> 且bit >>= 1，即到下一层子树继续这个过程，直到找到一个右子树上的点。
        *
        * ==> 以上图为例，count = 8对应的reverse为1000，为子树1的最左节点
        * ==> count = 9时，因为8在左子树，找到子树1的右子树的对应节点为12
        * ==> count = 10，第一轮找到对称节点为8，在左子树，
        *   ==> 下降一层，到子树2的右子树上找对应的节点，找到10
        * ==> count = 11，找到1的右子树的对应节点14
        * ==> count = 12，第一轮得到对应节点为10，在左子树，
        *   ==> 下降一层，在子树2上，得到对应节点为8，得知原节点在2的右子树上，
        *       ==> 下降一层，在子树4上，得到对应右节点为9
        * ==> ...
        * ==> count = 15，得到节点15
        * ==> count = 16，
        */
```

```
37            if((reverse & bit) != 0)
38            {
39                break;
40            }
41            bit >>= 1;
42        }
43
44        /*
45        * count = 16,得到bit = 0，即叶子节点层已经没有空余节点了
46        * 将最高位左移一位，即树高加一层，并在新层上找next。
47        */
48        if(bit == 0)
49        {
50            reverse = highBit << = 1;
51        }
52
53        return reverse;
54    }
```

181.
作者有给出

182.
不从头开始也是可以的，因为要求只是静态一致性。

[java] view plain copy

```
1   package p182;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
5
6
7
8   public class LazySkipListQueue
9   {
10      static final int MAX_LEVEL = 4;
11      static int randomSeed = (int)(System.currentTimeMillis()) | 0x0100;
12
13      private static int randomLevel() {
14          int x = randomSeed;
15          x ^= x << 13;
16          x ^= x >>> 17;
17          randomSeed = x ^= x << 5;
18          if ((x & 0x80000001) != 0) // test highest and lowest bits
19              return 0;
20          int level = 1;
21          while (((x >>>= 1) & 1) != 0) ++level;
22          return Math.min(level, MAX_LEVEL-1);
23      }
24      private static final class Node
25      {
26          final Lock lock  = new ReentrantLock();
27          final T item;
28          final int priority;
```

```
29          final Node[] next;
30          volatile boolean marked = false;
31          volatile boolean fullLinked = false;
32          private int topLevel;
33          @SuppressWarnings("unchecked")
34          public Node(int priority)
35          {
36              this.item = null;
37              this.priority = priority;
38              next = (Node[])new Node[MAX_LEVEL + 1];
39              topLevel = MAX_LEVEL;
40          }
41
42          @SuppressWarnings("unchecked")
43          public Node(T x, int priority)
44          {
45              this.item = x;
46              this.priority = priority;
47              int height = randomLevel();
48              next =  (Node[])new Node[height + 1];
49              topLevel = height;
50          }
51
52          public void lock()
53          {
54              lock.lock();
55          }
56
57          public void unlock()
58          {
59              lock.unlock();
60          }
61      }
62
63      final Node head = new Node(Integer.MIN_VALUE);
64      final Node tail = new Node(Integer.MAX_VALUE);
65
66      public LazySkipListQueue()
67      {
68          for(int i = 0; i < head.next.length; i ++)
69          {
70              head.next[i] = tail;
71          }
72      }
73
74      int find(Node node, Node[] preds, Node[] succs)
75      {
76          int lFound = -1;
77          Node pred = head;
78          for(int level = MAX_LEVEL; level >= 0; level --)
79          {
80              Node curr = pred.next[level];
81              while(node.priority > curr.priority)
82              {
83                  pred = curr;
84                  curr = pred.next[level];
```

```
85              }
86              if(lFound == -1 && node.priority == curr.priority)
87              {
88                  lFound = level;
89              }
90              preds[level] = pred;
91              succs[level] = curr;
92          }
93
94          return lFound;
95      }
96
97      @SuppressWarnings("unchecked")
98      boolean add(Node newNode)
99      {
100         int topLevel = randomLevel();
101         Node[] preds = (Node[]) new Node[MAX_LEVEL + 1];
102         Node[] succs = (Node[]) new Node[MAX_LEVEL + 1];
103         while(true)
104         {
105             int lFound = find(newNode, preds, succs);
106             if(lFound != -1)
107             {
108                 Node nodeFound = succs[lFound];
109                 if(!nodeFound.marked)
110                 {
111                     while(!nodeFound.fullLinked){}
112                     return false;
113                 }
114                 continue;
115             }
116
117             int highestLocked = -1;
118
119             try
120             {
121                 Node pred, succ;
122                 boolean valid = true;
123                 for(int level = 0; valid && (level < topLevel); level ++)
124                 {
125                     pred = preds[level];
126                     succ = succs[level];
127                     pred.lock();
128                     highestLocked = level;
129                     valid = !pred.marked && !succ.marked && pred.next[level] == succ;
130                 }
131                 if(!valid)
132                 {
133                     continue;
134                 }
135
136                 for(int level = 0; level <= topLevel; level ++)
137                 {
138                     newNode.next[level] = succs[level];
139                 }
140                 for(int level = 0; level <= topLevel; level ++)
```

```
141                 {
142                     preds[level].next[level] = newNode;
143                 }
144                 newNode.fullLinked = true;
145                 return true;
146             }finally
147             {
148                 for(int level = 0; level <= highestLocked; level ++)
149                 {
150                     preds[level].unlock();
151                 }
152             }
153         }
154     }
155 
156     @SuppressWarnings("unchecked")
157     private boolean remove(Node victim)
158     {
159         boolean isMarked = false;
160         Node[] preds = (Node[]) new Node[MAX_LEVEL + 1];
161         Node[] succs = (Node[]) new Node[MAX_LEVEL + 1];
162         int topLevel = victim.topLevel;
163 
164         while(true)
165         {
166             int lFound = find(victim, preds, succs);
167 
168             if(!(lFound != -1
169                     && lFound == victim.topLevel
170                     && victim.marked))
171             {
172                 return true;
173             }
174 
175             victim.lock();
176 
177             int highestLocked = -1;
178 
179             try
180             {
181                 Node pred;
182                 boolean valid = true;
183 
184                 for(int level = 0; valid && (level <= topLevel); level ++)
185                 {
186                     pred = preds[level];
187                     pred.lock();
188                     highestLocked = level;
189                     valid = !pred.marked && pred.next[level] == victim;
190                 }
191                 if(!valid)
192                 {
193                     continue;
194                 }
195                 for(int level = topLevel; level >= 0; level --)
196                 {
```

```
197                 preds[level].next[level] = victim.next[level];
198             }
199             victim.unlock();
200             return true;
201         }finally
202         {
203             for(int i = 0; i <= highestLocked; i ++)
204             {
205                 preds[i].unlock();
206             }
207         }
208
209     }
210 }
211
212
213 public Node findAndMarkMin()
214 {
215     Node ret = null;
216     Node pred = null;
217     Node curr = null;
218
219     while(true)
220     {
221         pred = head;
222         curr = pred.next[0];
223
224         while(curr.marked && curr != tail)
225         {
226             pred = curr;
227             curr = pred.next[0];
228         }
229
230         if(curr == tail)
231         {
232             return null;
233         }else
234         {
235             try
236             {
237                 pred.lock();
238                 curr.lock();
239                 if(!curr.marked && !pred.marked && pred.next[0] == curr)
240                 {
241                     curr.marked = true;
242                     ret = curr;
243
244                     remove(ret);
245                     return ret;
246                 }
247             }finally
248             {
249                 pred.unlock();
250                 curr.unlock();
251             }
252         }
```

```
253             }
254
255         }
256
257     }
```

183.
2个线程同时findAndRemoveMin，因为都是从head开始沿list[bottom]寻找，所以很有可能在head.next[0]上冲突。

184.
因为有了全局同步的时间，所以任何相关事件都可以全排序，所以可以线性化。

Chapter16

185.
M1(n) = 2M1(n/2) + O(n)
==> M1(n) = O(nlogn)
M∞(n) = M∞(n/2) + O(n)
==> M∞(n) = O(n)
P = M∞(n) / M1(n) = logn

186.

187.

[plain] view plain copy

```
1   package p187;
2
3   import java.util.concurrent.ExecutionException;
4   import java.util.concurrent.ExecutorService;
5   import java.util.concurrent.Executors;
6   import java.util.concurrent.Future;
7
8   public class ArraySum
9   {
10      static ExecutorService Execs = Executors.newCachedThreadPool();
11
12      static int Add(int[] data) throws ExecutionException, InterruptedException
13      {
14          Result rc = new Result(0);
15          Future<?> future = Execs.submit(new AddClass(data, 0, data.length - 1, rc));
16          future.get();
17
18          return rc.get();
19      }
20
21      static class Result
22      {
23          private int val;
24
25          public Result(int val)
26          {
27              this.val = val;
28          }
```

```
29
30          public int get()
31          {
32              return this.val;
33          }
34
35          public void set(int val)
36          {
37              this.val = val;
38          }
39      }
40
41      static class AddClass implements Runnable
42      {
43          int[] data;
44          Result rc;
45          int lIndex;
46          int rIndex;
47
48
49          public AddClass(int[] data, int lIndex, int rIndex, Result rc)
50          {
51              this.data = data;
52              this.rc = rc;
53              this.lIndex = lIndex;
54              this.rIndex = rIndex;
55          }
56
57          public void run()
58          {
59              try
60              {
61                  if(rIndex < lIndex)
62                  {
63                      rc.set(0);
64                  }else if(lIndex == rIndex)
65                  {
66                      rc.set(data[lIndex]);
67                  }else
68                  {
69                      Result leftRc = new Result(0);
70                      Result rightRc = new Result(0);
71                      int mid = (lIndex + rIndex) / 2;
72
73                      Future<?> lf = Execs.submit(new AddClass(data, lIndex, mid, leftRc));
74                      Future<?> rf = Execs.submit(new AddClass(data, mid + 1, rIndex, rightRc));
75
76                      lf.get();
77                      rf.get();
78
79                      rc.set(leftRc.get() + rightRc.get());
80                  }
81
82              }catch(Exception e)
83              {
84                  e.printStackTrace();
```

```
85                }
86
87            }
88        }
89
90    }
```

188.
T1 <= T4 * 4 = 320; T1 <= T64 * 64 = 320; ==> T1 <= 320;
T∞ <= T4 = 80; T∞ <= T64 = 10; ==> T∞ <= 10;
而且都满足第三个不等式。
T10 <= T1 / 10 + (T∞ * 9) / 10 = 41。

189.

[java] view plain copy

```
1   package p189;
2
3   public class Matrix
4   {
5       private T[][] vcs;
6       private final int sRow;
7       private final int eRow;
8       private final int sCol;
9       private final int eCol;
10      private final int rLen;
11      private final int cLen;
12
13      @SuppressWarnings("unchecked")
14      public Matrix(int rowLen, int colLen)
15      {
16          this.vcs = (T[][]) new Object[rowLen][];
17          for(int i = 0; i < rowLen; i ++)
18          {
19              vcs[i] = (T[]) new Object[colLen];
20          }
21
22          sRow = 0;
23          eRow = rowLen - 1;
24          sCol = 0;
25          eCol = colLen - 1;
26          rLen = rowLen;
27          cLen = colLen;
28      }
29
30      public Matrix(T[][] vcs, int sRow, int eRow, int sCol, int eCol)
31      {
32          this.vcs = vcs;
33          this.sRow = sRow;
34          this.eRow = eRow;
35          this.sCol = sCol;
36          this.eCol = eCol;
37          this.rLen = eRow - sRow + 1;
38          this.cLen = eCol - sCol + 1;
```

```
39      }
40
41      public void set(int row, int col, T val) throws Exception
42      {
43          if((sRow + row) > eRow) throw new Exception("Invalid row ");
44          if((sCol + col) > eCol) throw new Exception("Invalid col");
45          vcs[sRow + row][sCol + col] = val;
46      }
47
48      public T get(int row, int col) throws Exception
49      {
50          if((sRow + row) > eRow) throw new Exception("Invalid row ");
51          if((sCol + col) > eCol) throw new Exception("Invalid col");
52          return vcs[sRow + row][sCol + col];
53      }
54
55      public int getRowLength()
56      {
57          return rLen;
58      }
59
60      public int getColLength()
61      {
62          return cLen;
63      }
64
65      @SuppressWarnings("unchecked")
66      public Matrix[][] split()
67      {
68          int mRow = (sRow + eRow) /2;
69          int mCol = (sCol + eCol) / 2;
70
71          Matrix[][] rcs = (Matrix[][]) new Matrix<?>[2][];
72
73          for(int i = 0; i < rcs.length; i ++)
74          {
75              rcs[i] = (Matrix[]) new Matrix<?>[2];
76          }
77
78          rcs[0][0] = new Matrix(vcs, sRow, mRow, sCol, mCol);
79
80          if(sCol == eCol)
81          {
82              rcs[0][1] = new Matrix(vcs, 0, -1, 0, -1);
83              rcs[1][1] = new Matrix(vcs, 0, -1, 0, -1);
84          }else
85          {
86              rcs[0][1] = new Matrix(vcs, sRow, mRow, mCol + 1, eCol);
87          }
88
89          if(sRow == eRow)
90          {
91              rcs[1][0] = new Matrix(vcs, 0, -1, 0, -1);
92              rcs[1][1] = new Matrix(vcs, 0, -1, 0, -1);
93          }else
94          {
```

```
95              rcs[1][0] = new Matrix(vcs, mRow + 1, eRow, sCol, mCol);
96              rcs[1][1] = new Matrix(vcs, mRow + 1, eRow, mCol + 1, eCol);
97          }
98          return rcs;
99      }
100
101
102     public String toString()
103     {
104         if(rLen <= 0 || cLen <= 0)
105         {
106             return "Empty";
107         }
108
109         StringBuilder str = new StringBuilder();
110
111         for(int i = sRow; i <= eRow; i ++)
112         {
113             for(int j = sCol; j <= eCol; j ++)
114             {
115                 str.append(vcs[i][j]);
116                 str.append(", ");
117             }
118             str.append("\n");
119         }
120
121         return str.toString();
122     }
123
124     public static void main(String[] args)
125     {
126         Integer[][] data = new Integer[8][];
127         for(int i = 0; i < data.length; i ++)
128         {
129             data[i] = new Integer[8];
130         }
131
132         for(int i = 0; i < data.length; i ++)
133         {
134             for(int j = 0; j < data[i].length; j ++)
135             {
136                 data[i][j] = i * 10 + j;
137             }
138         }
139
140         Matrix m = new Matrix(data, 0, data.length - 1, 0, data[0].length - 1);
141
142         while(m.getRowLength() > 0)
143         {
144             System.out.println(m);
145
146             Matrix[][] subMs = m.split();
147
148             for(int i = 0; i < subMs.length; i ++)
149             {
150                 for(int j = 0; j < subMs[i].length; j ++)
```

```
151                 {
152                     System.out.println(subMs[i][j]);
153                 }
154             }
155 
156             m = subMs[1][1];
157         }
158     }
159 }
```

190.

[java] view plain copy

```
1   package p190;
2   
3   import java.util.concurrent.ExecutionException;
4   import java.util.concurrent.ExecutorService;
5   import java.util.concurrent.Executors;
6   import java.util.concurrent.Future;
7   
8   public class PolyOp
9   {
10      static ExecutorService Execs = Executors.newCachedThreadPool();
11  
12      static Long Op(int x, Polynomial leftPrefix, Polynomial rightPrefix, OP op) throws ExecutionException, InterruptedExc
    eption
13      {
14          long xs[] = new long[leftPrefix.getDegree() * 2];
15          long sum = 1;
16          for(int i = 0; i < xs.length; i ++)
17          {
18              xs[i] = sum;
19              sum *= x;
20          }
21  
22          Result rc = new Result((long)0);
23          Future<?> f = Execs.submit(new OpClass(leftPrefix, rightPrefix, xs, rc, op));
24          f.get();
25  
26          return rc.get();
27      }
28  
29      static class Result
30      {
31          private T val;
32  
33          public Result(T val)
34          {
35              this.val = val;
36          }
37          public T get()
38          {
```

```
39              return this.val;
40          }
41          public void set(T val)
42          {
43              this.val = val;
44          }
45      }
46
47      static enum OP
48      {
49          ADD,
50          MUL,
51      };
52
53      static class OpClass implements Runnable
54      {
55          private Polynomial leftPrefix;
56          private Polynomial rightPrefix;
57          private final long x[];
58          private Result rc;
59          private final OP op;
60
61          public OpClass(Polynomial leftPrefix, Polynomial rightPrefix, long x[], Result rc, OP op)
62          {
63              this.leftPrefix = leftPrefix;
64              this.rightPrefix = rightPrefix;
65              this.x = x;
66              this.rc = rc;
67              this.op = op;
68          }
69
70          private void add()
71          {
72              try
73              {
74                  if(leftPrefix.getDegree() == 1)
75                  {
76                      rc.set(new Long(leftPrefix.get(0) + rightPrefix.get(0)));
77                  }else
78                  {
79                      Result fRc = new Result((long)0);
80                      Result lRc = new Result((long)0);
81
82                      Polynomial[] leftPs = leftPrefix.split();
83                      Polynomial[] rightPs = rightPrefix.split();
84
85                      Future<?> ff = Execs.submit(new OpClass(leftPs[0], rightPs[0], x, fRc, op));
86                      Future<?> lf = Execs.submit(new OpClass(leftPs[1], rightPs[1], x, lRc, op));
87
88                      ff.get();
89                      lf.get();
90
91                      long result = fRc.get() + lRc.get() * x[leftPs[0].getDegree()];
92                      rc.set(result);
93                  }
94              }catch(Exception e)
```

```
95              {
96                  e.printStackTrace();
97              }
98          }
99
100         private void mul()
101         {
102             try
103             {
104                 if(leftPrefix.getDegree() == 1)
105                 {
106                     rc.set(new Long(leftPrefix.get(0) * rightPrefix.get(0)));
107                 }else
108                 {
109                     Result fRc = new Result((long)0);
110                     Result lRc = new Result((long)0);
111                     Result mRc1 = new Result((long)0);
112                     Result mRc2 = new Result((long)0);
113
114                     Polynomial[] leftPs = leftPrefix.split();
115                     Polynomial[] rightPs = rightPrefix.split();
116
117                     Future<?> ff = Execs.submit(new OpClass(leftPs[0], rightPs[0], x, fRc, op));
118                     Future<?> fm1 = Execs.submit(new OpClass(leftPs[0], rightPs[1], x, mRc1, op));
119                     Future<?> fm2 = Execs.submit(new OpClass(leftPs[1], rightPs[0], x, mRc2, op));
120                     Future<?> fr = Execs.submit(new OpClass(leftPs[1], rightPs[1], x, lRc, op));
121
122                     ff.get();
123                     fr.get();
124                     fm1.get();
125                     fm2.get();
126
127                     long result = fRc.get() + (lRc.get() * x[leftPrefix.getDegree()]) + (mRc1.get() + mRc2.get()) * x[leftPs[0].getDegr
    ee()];
128                     rc.set(result);
129                 }
130
131             }catch(Exception e)
132             {
133                 e.printStackTrace();
134             }
135         }
136
137         public void run()
138         {
139             switch(op)
140             {
141             case ADD:
142                 add();
143                 break;
144             case MUL:
145                 mul();
146                 break;
147             default:
148                 System.out.println("Invalid op");
```

```
149                 break;
150             }
151         }
152     }
153
154     public static void main(String[] args)
155     {
156         int degree = 4;
157         Polynomial left = new Polynomial(degree);
158         Polynomial right = new Polynomial(degree);
159
160         for(int i = 0; i < left.getDegree(); i ++)
161         {
162             left.set(i, 1);
163             right.set(i, 1);
164         }
165
166         try
167         {
168             long rc = PolyOp.Op(2, left, right, PolyOp.OP.MUL);
169             System.out.println("Get  " + rc);
170         }catch(Exception e)
171         {
172             e.printStackTrace();
173         }
174     }
175 }
```

191.

矩阵2分，8个线程做乘法。然后n^2个线程做加法。
==> M∞(n) = M∞(n/2) + O(1) ==> M∞(n) = O(logn)
==> M1(n) = 8 * M(n/2) + O(n^2) ==> M1(n) = O(n^3)
[java] view plain copy

```
1   package p191;
2
3   import java.util.concurrent.ExecutionException;
4   import java.util.concurrent.ExecutorService;
5   import java.util.concurrent.Executors;
6   import java.util.concurrent.Future;
7
8   import p189.Matrix;
9
10  public class MatrixMul
11  {
12      static ExecutorService Execs = Executors.newCachedThreadPool();
13
14      public static void MUL(Matrix a, Matrix b, Matrix c)  throws ExecutionException, InterruptedException
15      {
16          Future<?> f = Execs.submit(new MultiClass(a, b, c));
17          f.get();
18          return;
19      }
20
```

```
21      static class MultiClass implements Runnable
22      {
23          private Matrix a;
24          private Matrix b;
25          private Matrix c;
26
27          public MultiClass(Matrix a, Matrix b, Matrix c)
28          {
29              this.a = a;
30              this.b = b;
31              this.c = c;
32          }
33
34          public void run()
35          {
36              if(a.getRowLength() == 0 || a.getColLength() == 0)
37              {
38                  return;
39              }
40
41              if(a.getRowLength() == 1 && a.getColLength() == 1)
42              {
43                  try
44                  {
45                      c.set(0, 0, a.get(0, 0) * b.get(0, 0));
46                  } catch (Exception e)
47                  {
48                      // TODO Auto-generated catch block
49                      e.printStackTrace();
50                  }
51
52                  return;
53              }
54
55              if(a.getColLength() > 1 || a.getRowLength() > 1)
56              {
57                  try
58                  {
59                      Matrix as[][] = a.split();
60                      Matrix bs[][] = b.split();
61
62                      Future<?>[] fs = new Future<?>[8];
63                      Matrix tcs0 = new Matrix(as[0][0].getRowLength(), bs[0][0].getColLength());
64                      Matrix tcs1 = new Matrix(as[0][0].getRowLength(), bs[0][1].getColLength());
65                      Matrix tcs2 = new Matrix(as[1][0].getRowLength(), bs[0][0].getColLength());
66                      Matrix tcs3 = new Matrix(as[1][0].getRowLength(), bs[0][1].getColLength());
67                      Matrix tcs4 = new Matrix(as[0][1].getRowLength(), bs[1][0].getColLength());
68                      Matrix tcs5 = new Matrix(as[0][1].getRowLength(), bs[1][1].getColLength());
69                      Matrix tcs6 = new Matrix(as[1][1].getRowLength(), bs[1][0].getColLength());
70                      Matrix tcs7 = new Matrix(as[1][1].getRowLength(), bs[1][1].getColLength());
71
72
73                      fs[0] = Execs.submit(new MultiClass(as[0][0], bs[0][0], tcs0));
74                      fs[1] = Execs.submit(new MultiClass(as[0][0], bs[0][1], tcs1));
75                      fs[2] = Execs.submit(new MultiClass(as[1][0], bs[0][0], tcs2));
76                      fs[3] = Execs.submit(new MultiClass(as[1][0], bs[0][1], tcs3));
```

```
77                  fs[4] = Execs.submit(new MultiClass(as[0][1], bs[1][0], tcs4));
78                  fs[5] = Execs.submit(new MultiClass(as[0][1], bs[1][1], tcs5));
79                  fs[6] = Execs.submit(new MultiClass(as[1][1], bs[1][0], tcs6));
80                  fs[7] = Execs.submit(new MultiClass(as[1][1], bs[1][1], tcs7));
81
82                  for(int i = 0; i < fs.length; i ++)
83                  {
84                      fs[i].get();
85                  }
86
87                  Future<?>[][] rcFs = new Future<?>[c.getRowLength()][];
88                  for(int i = 0; i < rcFs.length; i ++)
89                  {
90                      rcFs[i] = new Future<?>[c.getColLength()];
91                  }
92
93                  for(int i = 0; i < tcs0.getRowLength(); i ++)
94                  {
95                      for(int j = 0; j < tcs0.getColLength(); j ++)
96                      {
97                          rcFs[i][j] = Execs.submit(new MatrixAddClass(tcs0, tcs4, c, i, j, i, j));
98                      }
99                  }
100                 for(int i = 0; i < tcs1.getRowLength(); i ++)
101                 {
102                     for(int j = 0; j < tcs1.getColLength(); j ++)
103                     {
104                         rcFs[i]
        [j + tcs0.getColLength()] = Execs.submit(new MatrixAddClass(tcs1, tcs5, c, i, j, i, j + tcs0.getColLength()));
105                     }
106                 }
107                 for(int i = 0; i < tcs2.getRowLength(); i ++)
108                 {
109                     for(int j = 0; j < tcs2.getColLength(); j ++)
110                     {
111                         rcFs[i + tcs0.getRowLength()]
        [j] = Execs.submit(new MatrixAddClass(tcs2, tcs6, c, i, j, i + tcs0.getRowLength(), j));
112                     }
113                 }
114                 for(int i = 0; i < tcs3.getRowLength(); i ++)
115                 {
116                     for(int j = 0; j < tcs3.getColLength(); j ++)
117                     {
118                         rcFs[i + tcs0.getRowLength()][j + tcs0.getColLength()] =
119
                                Execs.submit(new MatrixAddClass(tcs3, tcs7, c, i, j, i + tcs0.getRowLength(), j + tcs0.getColLength())
        );
120                     }
121                 }
122
123                 for(int i = 0; i < rcFs.length; i ++)
124                 {
125                     for(int j = 0; j < rcFs[i].length; j ++)
126                     {
127                         rcFs[i][j].get();
128                     }
```

```
129                 }
130 
131 
132             }catch(Exception e)
133             {
134                 e.printStackTrace();
135             }
136 
137 
138         }
139     }
140 }
141 
142 static class MatrixAddClass implements Runnable
143 {
144     private Matrix a;
145     private Matrix b;
146     private Matrix c;
147     private int srcRow;
148     private int dstRow;
149     private int srcCol;
150     private int dstCol;
151 
152     public MatrixAddClass(Matrix a, Matrix b, Matrix c, int srcRow, int srcCol, int dstRow, int dstCol)
153     {
154         this.a = a;
155         this.b = b;
156         this.c = c;
157         this.srcRow = srcRow;
158         this.dstRow = dstRow;
159         this.srcCol = srcCol;
160         this.dstCol = dstCol;
161     }
162 
163     public void run()
164     {
165         Integer val = 0;
166 
167         try
168         {
169             val = a.get(srcRow, srcCol) + b.get(srcRow, srcCol);
170         }catch(Exception e)
171         {
172             e.printStackTrace();
173             val = 0;
174         }
175 
176         try
177         {
178             c.set(dstRow, dstCol, val);
179         }catch(Exception ie)
180         {
181             ie.printStackTrace();
182         }
183     }
184 }
```

```
185
186     }
```

192.
因为size是不断变化的。有可能T1上锁时qa.size() < qb.size(); T2上锁时qa.size() > qb.size()。以交叉的顺序取锁可能造成死锁。

193.
1. 如果不是volatile, 因为popTop是用top和bottom的index判断队列空，所以将会有错误的判断。比如说
T1.popBottom() --> T1.(bottom = 0) --> T1.CAS(top) ==> top = 0 && pop task[0];
T2.popTop() --> T2读到没有同步的bottom = 1 && oldTop = 0 ==> bottom > oldTop ==> T2.CAS(top) --> T2.pop task[0]
将破坏mutual
2. 可以尽早使popTop()得到bottom <= oldTop。 图中23行应该是最早的安全位置。因为bottom由popBottom控制，最早使bottom = 0也要在取得相关值并且满足条件之后，即最早在23行。在此之后，popTop和popBottom都统一的由CAS(top)取到可线性化性。之后不论是谁在此尝试都得知bottom = 0，即队列为空。如果没有现成操作pop，push将会使队列溢出。

194.
1. 因为如果先CAS然后取值，在这2个操作中间，task[oldTop]可能会被popBottom()和pushBottom()覆盖。
2.可以。因为isEmpty将获取最新的值，如果isEmpty() == TRUE，那么确实队列为空。如果isEmpty() == FALSE，则在这段时间中，top和bottom的值可能有变化。但是没有关系，可以看成在不用isEmpty()的算法中取值和CAS中间的时间的变化。

195.
pushBottom():
bottom = oldBottom + 1。 如果bottom没有被改变，pop都不能看到这个task。
popTop():
if(size <= 0) return null   或者 top.CAS
popBottom():
line20 or line 22 or line 27

196.

[java] view plain copy

```
1     public Runnable popTop()
2     {
3        while(true)
4        {
5           int[] stamp = new int[1];
6           int oldTop = top.get(stamp);
7           int newTop = oldTop + 1;
8           int oldStamp = stamp[0];
9           int newStamp = oldStamp + 1;
10
11          if(bottom <= oldTop)
12          {
13             return null;
14          }
15
16          Runnable r = task[oldTop];
17          if(top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
18          {
19             return r;
20          }
21       }
22
23       //impossible
```

```
24          return null;
25      }
```

197.

不能。因为pop方法内部都有同样功能的判断，而且读到的是比isEmpty读到的更新的值，并且2这实现上的开销基本相同。如果isEmpty与pop当中的非空判断得到同样的结果，则用户代码中没有调用这个方法也没有关系；如果结果不相同，则调用isEmpty是浪费时间

ch17

198. 如果线程 k + 2^r 比 线程 k 快很多，当线程 k + 2^r 读 a[k]的时候将读到没有 sum k段的原始值，没有求和的作用。

[java] view plain copy

```
1       public void run() {
2        int d = 1, sum = 0;
3        while (d < N) {
4         if (i >= d)
5           sum = a[i-d];
6         b.await();
7         if (i >= d)
8          a[i] += sum;
9         b.await();
10        d = d * 2;
11      }}}
```

代码来自书作者ppt。

199.

[java] view plain copy

```
1       package p199;
2
3       import java.util.concurrent.atomic.AtomicInteger;
4
5       public class SenseBarrier
6       {
7          private AtomicInteger count;
8          private final int size;
9          private boolean sense;
10         private ThreadLocal threadSense;
11
12         public SenseBarrier(int n)
13         {
14            count = new AtomicInteger(n);
15            size = n;
16            sense = false;
17            threadSense = new ThreadLocal()
18                  {
19                     protected Boolean initialValue() { return !sense; };
20                  };
21         }
22
23         public void await()
24         {
```

```
25            boolean mySense = threadSense.get();
26            int position = count.getAndDecrement();
27
28            if(position == 1)
29            {
30               count.set(size);
31               sense = mySense;
32
33               this.notifyAll();
34            }else
35            {
36               while(sense != mySense)
37               {
38                  try
39                  {
40                     this.wait();
41                  }catch(InterruptedException e)
42                  {
43                     e.printStackTrace();
44                  }
45               }
46            }
47
48            threadSense.set(!mySense);
49         }
50
51      }
```

当参与的线程行为不确定性强，响应时间要求不高，cpu不希望独占是采用wait的方式合适；否则采用spin比较合适。

200.

[java] view plain copy

```
1     package ch17.basic;
2
3     import java.util.concurrent.atomic.AtomicInteger;
4
5     public abstract class TreeNode
6     {
7        protected AtomicInteger count;
8        protected TreeNode parent;
9        protected volatile boolean sense;
10       protected final int radix;
11
12       public TreeNode(int radix)
13       {
14          sense = false;
15          parent = null;
16          this.radix = radix;
17          count = new AtomicInteger(radix);
18       }
19
20       public TreeNode(int radix, TreeNode parent)
21       {
22          this(radix);
```

```
23          this.parent = parent;
24      }
25
26      public void nodeNotify()
27      {
28          return;
29      }
30
31      abstract public void await();
32
33      abstract public void build(int depth, Integer leaves, TreeNode leaf[]);
34
35  }
```

[java] view plain copy

```
1   package ch17.p200;
2
3   import ch17.basic.Barrier;
4   import ch17.basic.TreeNode;
5
6   public class RunnableTree implements Barrier
7   {
8       final int radix;
9       Node[] leaf;
10      int leaves;
11      ThreadLocal threadSense;
12      ThreadLocal ThreadID;
13      Runnable task;
14
15      final int n;
16
17      public RunnableTree(int n, int radix, Runnable task)
18      {
19          this.radix = radix;
20          this.n = n;
21          leaves = 0;
22
23          threadSense = new ThreadLocal()
24                  {
25                      protected Boolean initialValue() { return true; };
26                  };
27          ThreadID = new ThreadLocal()
28                  {
29                      protected Integer initialValue() { return 0; };
30                  };
31
32          this.task = task;
33
34          build();
35      }
36
37      public void await()
38      {
```

```
39          int me = ThreadID.get();
40          Node myNode = leaf[me / radix];
41          myNode.await();
42      }
43
44      private void build()
45      {
46          leaf = new Node[n / radix];
47
48          int depth = 0;
49          int layNum = n;
50          while(layNum > 1)
51          {
52              depth ++;
53              layNum = layNum / radix;
54          }
55
56          Node root = new Node(radix);
57          Integer leaveNum = new Integer(0);
58          root.build(depth, leaveNum, leaf);
59          leaves = leaveNum;
60      }
61
62
63
64      private class Node extends TreeNode
65      {
66          public Node(int radix)
67          {
68              super(radix);
69          }
70
71          public Node(int radix, Node myParent)
72          {
73              super(radix, myParent);
74          }
75
76          public void build(int depth, Integer leaves, TreeNode leaf[])
77          {
78              if(depth < 0)
79              {
80                  return;
81              }
82              if(depth == 0)
83              {
84                  leaf[leaves] = this;
85                  leaves ++;
86                  return;
87              }
88
89              for(int i = 0; i < radix; i ++)
90              {
91                  Node node = new Node(radix, this);
92                  node.build(depth - 1, leaves, leaf);
93              }
94          }
```

```
95
96          public void await()
97          {
98              boolean mySense = threadSense.get();
99              int position = count.getAndDecrement();
100
101             if(position == 1)
102             {
103                 if(parent != null)
104                 {
105                     parent.await();
106                 }else
107                 {
108                     task.run();
109                 }
110                 count.set(radix);
111                 sense = mySense;
112             }else
113             {
114                 while(sense != mySense) {}
115             }
116             threadSense.set(!mySense);
117         }
118     }
119 }
```

201.

[java] view plain copy

```
1   package ch17.p201;
2
3   import ch17.basic.TreeNode;
4
5   public class GeneralTree
6   {
7       protected final int radix;
8       protected TreeNode[] leaf;
9       protected int leaves;
10      protected ThreadLocal ThreadID;
11
12      protected final int n;
13      protected TreeNode root;
14
15      //The inherited class should have root initiated
16      public GeneralTree(int n, int radix, TreeNode root)
17      {
18          this.radix = radix;
19          this.n = n;
20          leaves = 0;
21
22          ThreadID = new ThreadLocal()
23                  {
24                      protected Integer initialValue() { return 0; };
25                  };
```

```
26
27          this.root = root;
28          build();
29      }
30
31      public void await()
32      {
33          int me = ThreadID.get();
34          TreeNode myNode = leaf[me / radix];
35          myNode.await();
36      }
37
38      protected void build()
39      {
40          leaf = new TreeNode[n / radix];
41
42          int depth = 0;
43          int layNum = n;
44          while(layNum > 1)
45          {
46              depth ++;
47              layNum = layNum / radix;
48          }
49
50          Integer leaveNum = new Integer(0);
51          root.build(depth, leaveNum, leaf);
52          leaves = leaveNum;
53      }
54
55  }
```

202.

[java] view plain copy

```
1   package ch17.p202;
2
3   public class TourBarrier
4   {
5       TourNode[] nodes;
6       TourNode[] leaf;
7       ThreadLocal ThreadID;
8       ThreadLocal threadSense;
9
10      public TourBarrier(int n)
11      {
12          ThreadID = new ThreadLocal()
13          {
14              protected Integer initialValue() { return 0; };
15          };
16          this.threadSense = new ThreadLocal() {
17              protected Boolean initialValue() { return true; };
18          };
19
20          int nodeNum = (1 << n) - 1;
```

```
21          nodes = new TourNode[nodeNum];
22          int rootIndex = 0;
23          nodes[rootIndex] = new TourNode(rootIndex);
24          nodes[rootIndex].build(n - 1, nodes);
25
26          int leafNum = (1 << (n - 1));
27          leaf = new TourNode[leafNum];
28          for(int i =0; i < leafNum; i ++)
29          {
30              leaf[leafNum - i - 1] = nodes[nodeNum - i - 1];
31          }
32      }
33
34      public void await()
35      {
36          int me = ThreadID.get();
37          TourNode myLeaf = leaf[me / 2];
38          boolean sense = threadSense.get();
39          myLeaf.await(sense);
40          threadSense.set(!sense);
41      }
42
43
44      private class TourNode
45      {
46          final int myIndex;
47          volatile boolean flag;
48          boolean active;
49          int parentIndex;
50          int partnerIndex;
51
52          public TourNode(int index)
53          {
54              this.myIndex = index;
55              this.flag = false;
56              this.active = false;
57              this.parentIndex = -1;
58              setPartnerIndex();
59          }
60
61          public TourNode(int parentIndex, int index)
62          {
63              this(index);
64              this.active = true;
65              this.parentIndex = parentIndex;
66          }
67
68          private void setPartnerIndex()
69          {
70              if(myIndex % 2 == 0)
71              {
72                  this.partnerIndex = myIndex + 1;
73              }else
74              {
75                  this.partnerIndex = myIndex - 1;
76              }
```

```
77          }
78
79          public int getIndex()
80          {
81              return myIndex;
82          }
83
84          public void build(int depth, TourNode nodes[])
85          {
86              if(depth < 0)
87              {
88                  System.out.println("Invalid depth");
89                  return;
90              }else if(depth == 0)
91              {
92                  return;
93              }else
94              {
95                  int leftIndex = this.myIndex * 2 + 1;
96                  int rightIndex = (this.myIndex * 2) + 2;
97
98                  nodes[leftIndex] = new TourNode(this.getIndex(), leftIndex);
99                  nodes[rightIndex] = new TourNode(rightIndex);
100
101                 nodes[leftIndex].build(depth - 1, nodes);
102                 nodes[rightIndex].build(depth - 1, nodes);
103             }
104         }
105
106         public void await(boolean sense)
107         {
108             if(active)
109             {
110                 if(parentIndex >= 0)
111                 {
112                     while(flag != sense) {}
113
114                     nodes[parentIndex].await(sense);
115                     nodes[partnerIndex].flag = sense;
116                 }
117             }else
118             {
119                 nodes[partnerIndex].flag = sense;
120                 while(flag != sense){}
121             }
122         }
123     }
124
125 }
```

203.

这是不对的.因为每次在特定节点上竞争的线程是不固定的.比如,4个线程的树, 第一次由线程1和线程3到达root，假设所有ThreadLocal初始值为false，则线程1和3从root返回后mySense = true；同时线程2和线程4在root上的mySense=false；root的sense = false。第二次barrier由线程2和4到达，线程2发现sense == mySense，于是没有与线程同步而直接从root返回。

204.
这种树的算法是不正确的。
如果障碍的用法是先执行操作，然后等待在树上，则有可能出现的情形是，左子树的所有节点都已经完成操作，右子树的所有Active节点都到达parent直至root，但是有的passive节点还没有完成phaseI的操作 ==》 左子树节点都从root向下返回，并开始phaseII的操作。这是可能会涉及与右子树节点的交互，并在不同步的状态下得到结果作为phaseII的结果。

如果障碍的用法是先在树上等待返回再执行操作：障碍的正确性在于如果不是所有的线程都完成了phase1则不能开始phase2。但是这个算法只能保证如果不是所有线程都开始phase1则不能开始phase2。比如这样的情形：左右子树的所有节点都已经开始执行phase1==》所有的active节点都完成了phase1的操作==》所有active节点向root回溯 ==》根结点的左子树都已经完成phase1 ==》根结点的左子树的所有节点都开始执行phase2但是右子树还有节点正在执行phase1 ==》不同步的状态。

205.

[java] view plain copy

```
1   package ch17.p205;
2
3   import counting.Bitonic;
4
5   public class CounterBarrier implements ch17.basic.Barrier
6   {
7       private Bitonic counter;
8       private volatile boolean sense;
9       private ThreadLocal ThreadId;
10      private ThreadLocal threadSense;
11      private final int size;
12
13      public CounterBarrier(int size)
14      {
15          this.size = size;
16          this.sense = false;
17
18          this.counter = new Bitonic(size);
19
20          ThreadId = new ThreadLocal()
21          {
22              protected Integer initialValue() { return 0; };
23          };
24          this.threadSense = new ThreadLocal() {
25              protected Boolean initialValue() { return (!sense); };
26          };
27      }
28
29      public void await()
30      {
31          int myId = ThreadId.get();
32          boolean mySense = threadSense.get();
33
34          int output = counter.traverse(myId);
35          if((output % this.size) == 0)
36          {
37              sense = !sense;
38          }else
39          {
40              while(mySense != this.sense) {}
```

```
41            }
42            threadSense.set(!mySense);
43        }
44
45    }
```

206.

[java] view plain copy

```
1   package ch17.p206;
2
3   import register.WFSnapshot;
4
5   public class SnapshotBarrier implements TDBarrier
6   {
7       private WFSnapshot snapshot;
8
9   //  ThreadLocal ThreadId;
10
11      public SnapshotBarrier(int capacity)
12      {
13          snapshot = new WFSnapshot(capacity, false);
14      }
15
16      public void setActive(boolean state)
17      {
18          snapshot.update(state);
19      }
20
21      public boolean isTerminated()
22      {
23          Boolean[] results = snapshot.scan();
24          for(int i = 0; i < results.length; i ++)
25          {
26              if(results[i])
27              {
28                  return false;
29              }
30          }
31          return true;
32      }
33  }
```

207.

可以用归纳法证明，当一个线程完成了第i步，即已经收到了i-(2^r)(mod n）的通知 ，则线程k已经与线程 k - 1, k - 2, ..., k - (2 ^ i) + 1 同步了。

假设在i 时为真，则 i + 1时，线程k在i是已经与前 (2 ^ i)个线程同步了，同时线程k与线程(k - (2 ^ i))同步，线程(k - (2 ^ i)) 已经与线程 (k -(2 ^ i))， (k - (2 ^ i) - 1), ... , (k - (2 ^ (i + 1)) + 1) 同步了。得证。

所以要所有的线程都与其他的线程同步，需要 logn轮同步。

如果线程数不是2 ^ i，也没有关系，只是线程k在同步时，同步的前第1到第n个线程，与第n+1到2n个线程会有重叠的部分。但是并不会有重复的消息，只要能够消息能够区分是针对哪一步。因为先发后收，也不会有死锁。

208.

[java] view plain copy

```
1  package ch17.p208;
2
3
4  public class DisBarrier implements ch17.basic.Barrier
5  {
6      private final int size;
7      private final int powSize;
8      volatile private boolean[][] flags;
9      private ThreadLocal ThreadId;
10     private ThreadLocal flag;
11
12     public DisBarrier(int powSize)
13     {
14         this.powSize = powSize;
15         this.size = (int)Math.pow(2, powSize);
16
17         flags = new boolean[size][];
18         for(int i = 0; i < size; i ++)
19         {
20             //initialized as all false
21             flags[i] = new boolean[powSize];
22         }
23
24         ThreadId = new ThreadLocal()
25         {
26             protected Integer initialValue() { return 0; };
27         };
28
29         flag = new ThreadLocal()
30         {
31             protected Boolean initialValue() { return true; };
32         };
33     }
34
35     public void await()
36     {
37         int myId = ThreadId.get();
38         boolean myFlag = flag.get();
39
40         int step = 1;
41         for(int i = 0; i < powSize; i ++)
42         {
43             int toIndex = (myId + step) % size;
44             flags[toIndex][i] = myFlag;
45
46             while(flags[myId][i] != myFlag) {}
47             step <<= 1;
48         }
49         flag.set(!myFlag);
50     }
51 }
```

209.
设线程数为 n = (r ^ d)，树为r叉树，一共有(1 - (r ^d)) / (1 - r)个节点。

组合树：每个节点经过了r个getAndDecrement操作，2个反转操作（setCounter, !sense）和r个 threadSense反转操作。
静态树：每个节点经历了r个递减操作和r个threadSense反转操作。
分发树：每个线程经历了 logn轮的操作，每轮包括一个发消息操作和一个收消息操作。
更具体的分析见 barrier

210.

[java] view plain copy

```
1   package ch17.p210;
2
3   import java.util.concurrent.atomic.AtomicInteger;
4
5   import programm.ch17.Barrier.src.barrier.TDBarrier;
6
7   public class ActiveTDBarrier implements TDBarrier
8   {
9       AtomicInteger count;
10      ThreadLocal ThreadId;
11      final int capacity;
12      AtomicInteger totalActivated;
13      Runnable[] tasks;
14      DEQueue[] queues;
15
16      public ActiveTDBarrier(int n, DEQueue[] queue, Runnable[] tasks)
17      {
18          this.count = new AtomicInteger(0);
19          this.capacity = n;
20          this.queues = queue;
21          this.tasks = tasks;
22          totalActivated = new AtomicInteger(0);
23          ThreadId = new ThreadLocal()
24          {
25              protected Integer initialValue() { return 0; };
26          };     // TODO Auto-generated constructor stub
27      }
28
29      public void setActive(boolean active)
30      {
31          if(active)
32          {
33              count.getAndIncrement();
34              totalActivated.getAndIncrement();
35          }else
36          {
37              count.getAndDecrement();
38          }
39      }
40
41      public boolean isTerminated()
42      {
```

```
43            int localActivated = totalActivated.get();
44            int checkNum = 0;
45
46            while(checkNum < 2)
47            {
48               if(count.get() != 0)
49               {
50                  return false;
51               }
52
53               if(localActivated != totalActivated.get())
54               {
55                  return false;
56               }
57               //No one moves in between count reading
58               //task may be pushed in queue or popped out the queue
59
60               for(int i = 0; i < queues.length; i ++)
61               {
62                  if(!queues[i].isEmpty())
63                  {
64                     return false;
65                  }
66               }
67               //task my be pushed in queue or popped out the queue or running
68
69               for(int i = 0; i < tasks.length; i ++)
70               {
71                  if(tasks[i] != null)
72                  {
73                     return false;
74                  }
75               }
76               //task may be popped out the queue or new task pushed into queue
77
78               /*After first check, the new task may be
79                  in queue
80                  or in task
81                  or had finished
82               They could be detected by
83                  queue check
84                  or task check
85                  or totalActivated
86                */
87               checkNum ++;
88            }
89
90            return localActivated == totalActivated.get();
91         }
92      }
```

[java] view plain copy

```
1     package ch17.p210;
2
3     import java.util.Random;
4
```

```
5   public class ActiveTDThread
6   {
7       DEQueue[] queue;
8       ActiveTDBarrier tdBarrier;
9       Runnable[] tasks;
10      Random random;
11      final static int QueueSize = 64;
12
13      public ActiveTDThread(int n) {
14       queue = new DEQueue[n];
15       tasks = new Runnable[n];
16       tdBarrier = new ActiveTDBarrier(n, queue, tasks);
17       random = new Random();
18       for (int i = 0; i < n; i++) {
19         queue[i] = new DEQueue(QueueSize);
20       }
21       for(int i = 0; i < n; i ++)
22       {
23          tasks[i] = null;
24       }
25      }
26
27      public void run()
28      {
29       int me = ThreadID.get();
30       tdBarrier.setActive(true);
31       tasks[me] = queue[me].popBottom(); // attempt to pop 1st item
32       while (true)
33       {
34          while (tasks[me] != null)
35          { // if there is an item
36             tasks[me].run();      // execute it and then
37             tasks[me] = queue[me].popBottom(); // pop the next item
38          }
39          tdBarrier.setActive(false); // no work
40          while (tasks[me] == null)
41          { // steal an item
42             int victim = random.nextInt(queue.length);
43             if (!queue[victim].isEmpty())
44             {
45                tdBarrier.setActive(true);  // tentatively active
46                tasks[me] = queue[victim].popTop();
47                if (tasks[me] != null)
48                {
49                   tdBarrier.setActive(true);
50                }
51             }
52             if (tdBarrier.isTerminated())
53             {
54                return;
55             }
56          }
57       }
58    }
59
60  }
```