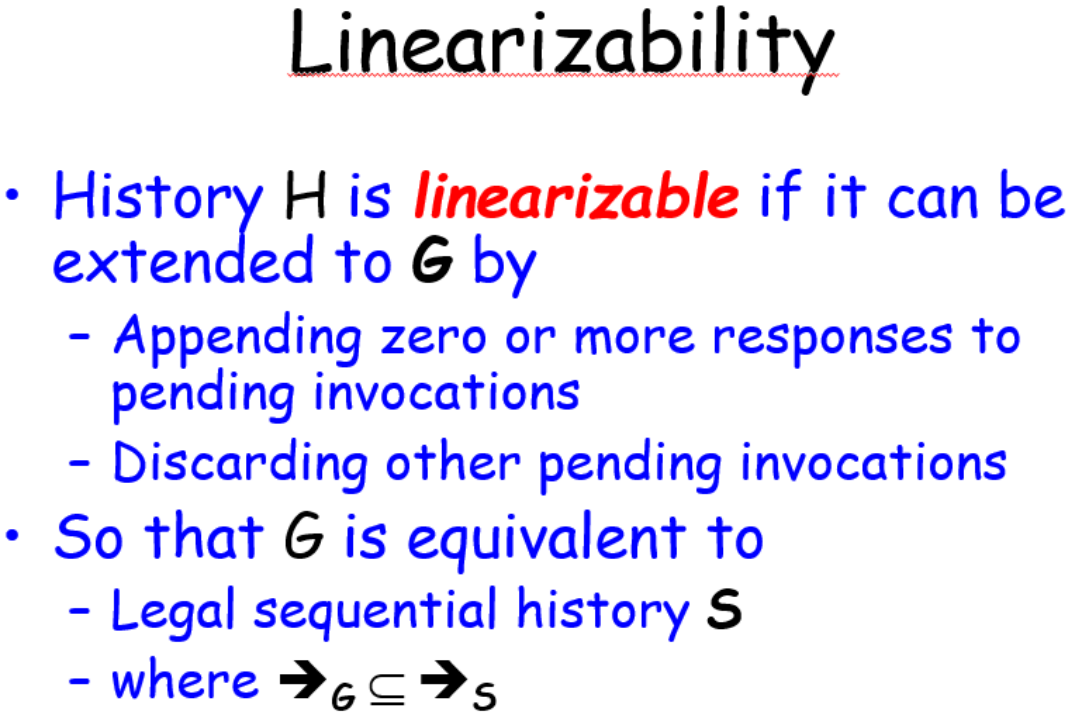
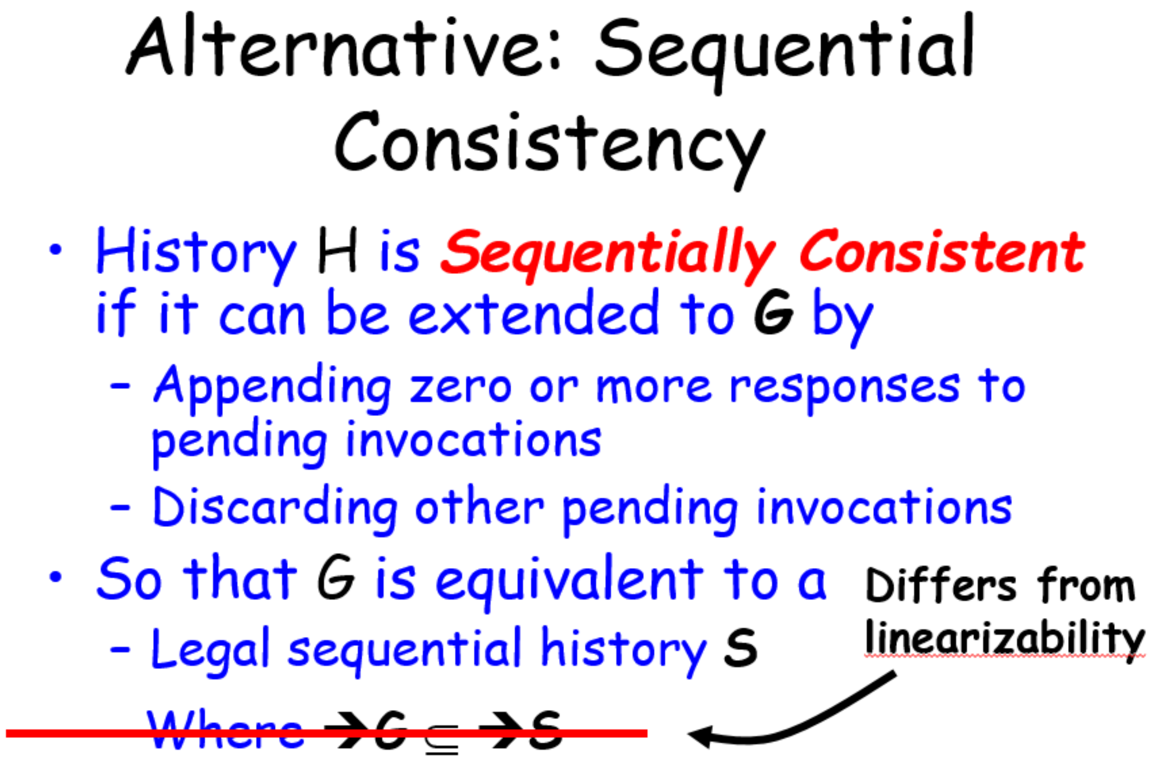
Concepts

•Linearizability, Sequential Consistency (Chapter 3)

Linearizability:



Sequential Consistency:



参考：https://blog.csdn.net/chao2016/article/details/81149674

个人理解：

线性一致性：所有线程共用一条时间线，线程之间的相对顺序是确定的

顺序一致性：每个线程内部的顺序是确定的，但线程之间的相对顺序可以随意改变

•Progress: Lock-freedom, Wait-Freedom, Deadlock-freedom, Starvation-Freedom (Chapter 3)

```
Deadlock-free: some thread trying to acquire the lock eventually succeeds.
Starvation-free: every thread trying to acquire the lock eventually succeeds.
Lock-free: some thread calling a method eventually returns.
Wait-free: every thread calling a method eventually.
```

（1）wait-free：不管OS如何调度线程，每个线程始终在做有用的事情。

（2）lock-free：不管OS如何调度线程，至少有一个线程在做有用的事情。

因此，如果服务中有了锁，有可能拿到锁的线程去做IO，等待；其他线程又依赖这个锁，整个线程没有做有用的事情，因此有锁一定不是lock-free，更不可能是wait-free。

(1)Lock-free is the same as wait-free if the execution is finite.

(2)“Lock-free is to wait-free as deadlock-free is to lockout-free.” In other words, Lock-free and Deadlock-free are “stalinistic”, preferring group progress, while Wait-free and Lockout-free guarantee individual progress.

(3) Any wait-free implementation is lock-free.

参考：https://www.jianshu.com/p/baaf53d69b51

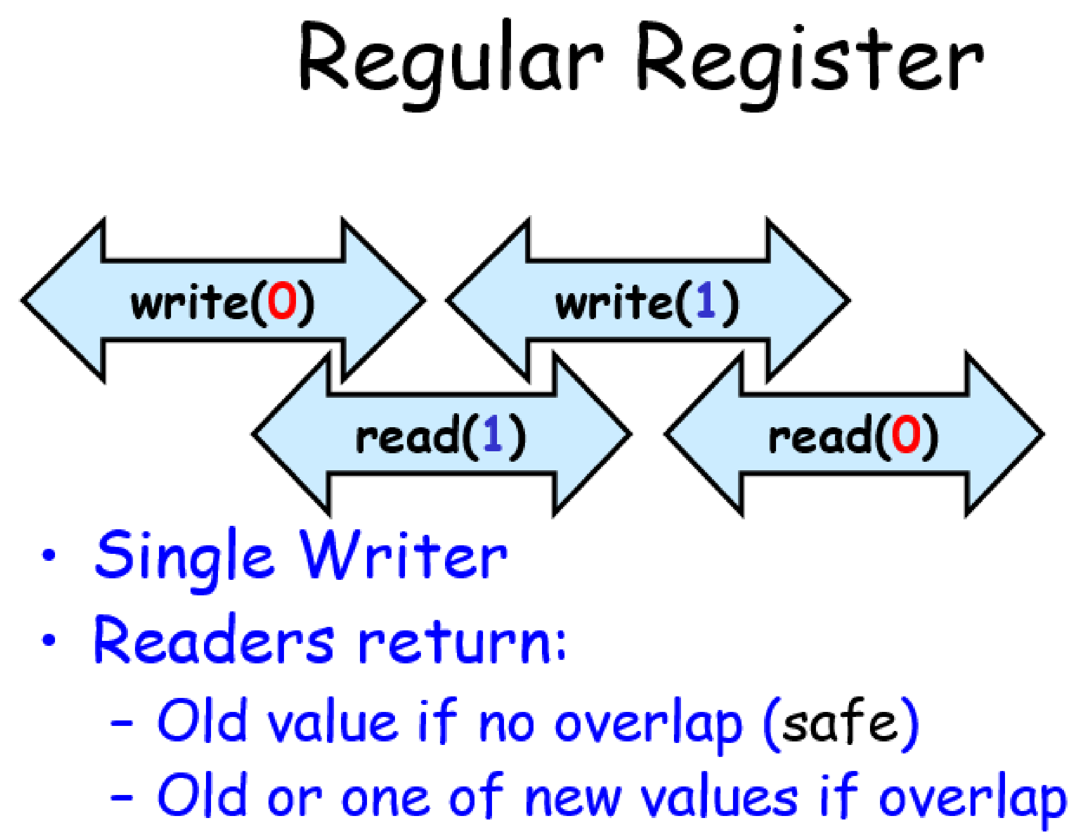
https://blog.csdn.net/misayaaaaa/article/details/100063319

•For locks: Mutual Exclusion, Deadlock-freedom, Starvation-Freedom

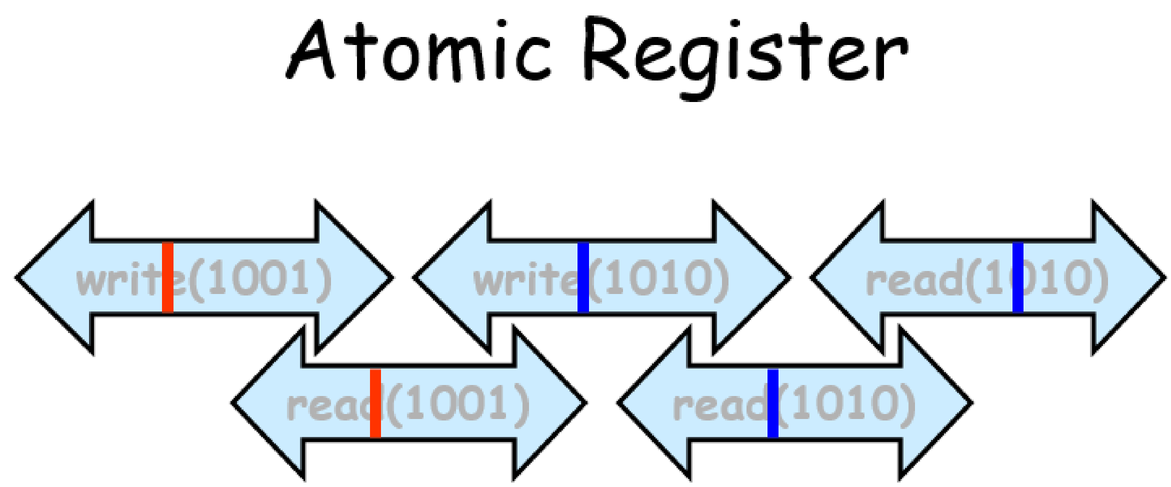
互斥、无死锁、无饥饿

•For registers: Safe, Regular, Atomic (Chapter 4)

安全：读和写不重叠



Regular：More precisely, (1) A regular register is safe; (2) If a read call overlaps the i-th write call, then the read call may return either the i-th or (i-1)-th value.



原子寄存器是可线性化到顺序安全寄存器的寄存器（这里原子=可线性化）

参考：https://www.thinbug.com/q/8871633

•Consensus number (Chapter 5)

Consistent: all threads decide the same value

Valid: the common decision value is some thread's input

Consensus Numbers

- An object X has consensus number n
  - If it can be used to solve n-thread consensus
    - Take any number of instances of X
    - together with atomic read/write registers
    - and implement n-thread consensus
  - But not (n+1)-thread consensus

•Read-Modify-Write Operations (Chapter 5)

•Objects

•Method call

–Returns object's prior value **x**

–Replaces **x** with **mumble(x)**

–Atomically

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
  getAndMumble() {
    int prior  = this.value;
    this.value = mumble(this.value);
    return prior;
  }
}

Example:
public abstract class RMWRegister {
 private int value;
 public int synchronized
   getAndIncrement() {
  int prior  = this.value;
  this.value = this.value + 1;
  return prior;
 }
 …
}
```

Theorem

•Any non-trivial RMW object has consensus number at least 2

•Corollary: No wait-free implementation of non-trivial RMW objects from atomic registers

•Hardware RMW instructions not just a convenience

Proof:

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

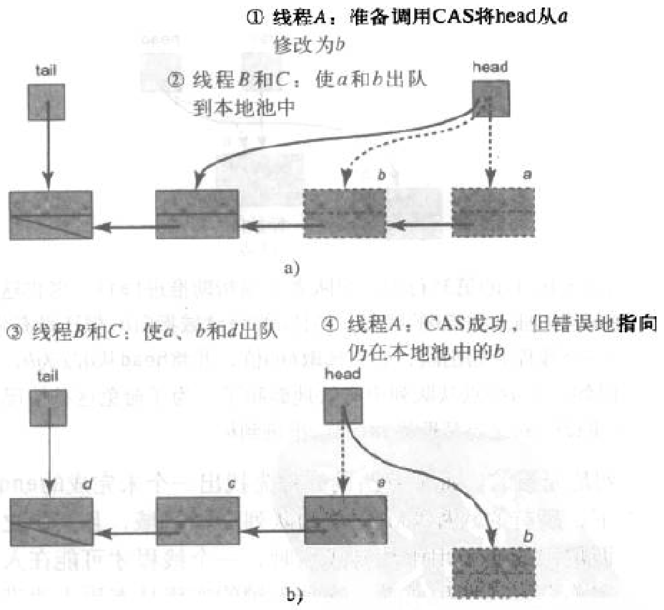
•ABA Problem, Lost-Wakeup Problem

Lost-Wakeup Problem：一个或多个线程将永远等待，而不会意识到它们等待的条件已经为真。因为线程唤醒是没有顺序的，所以可能导致一些线程永远在等待。解决方法：Always use signalAll and notifyAll

ABA Problem：就是说一个线程把数据A变为了B，然后又重新变成了A。此时另外一个线程读取的时候，发现A没有变化，就误以为是原来的那个A。解决方法：加时间戳

当一个入队线程需要一个新结点时，它尝试从线程本地空闲链表中删除一个结点。如果空闲链表为空，则使用new操作分配一个结点。当一个出队线程准备释放一个结点时，它将该结点链入到线程本地空闲链表。因为链表是线程本地的，因此不需要很大的同步开销。只要每个线程的入队和出队次数大致相等，这种设计的效果就非常好。如果两种操作次数不平衡，则需要更加复杂的技术，例如定期从其他线程窃取结点。

令人惊讶的是，如果采用最直接的方式回收结点，那么这种无锁队列将会出错。考虑图10-14所描述的场景。在图a中，出队线程1发现哨兵结点为a，下一个结点是b。然后准备用旧值a和新值b调用compareAndSet()来修改head。在进入第二步之前，其他线程让b和它的后继结点相继出队，并将a和b放入空闲池。如图b所示，结点a被循环使用，并最终重新作为队列的哨兵结点。线程现在唤醒，调用compareAndSet()，由于head的旧值的确是a，所以成功返回。不幸的是，已经重设head指向了一个被回收的结点。



Algorithms

•Registers & Snapshot

Simple Snapshot： Linearizable、Update is wait-free、But Scan can starve

```
public class SimpleSnapshot implements Snapshot {
  private AtomicMRSWRegister[] register;

  public void update(int value) {
    int i = Thread.myIndex();
    LabeledValue oldValue = register[i].read();
    LabeledValue newValue =
     new LabeledValue(oldValue.label+1, value);
    register[i].write(newValue);
  }

  private LabeledValue[] collect() {
   LabeledValue[] copy =
    new LabeledValue[n];
   for (int j = 0; j < n; j++)
    copy[j] = this.register[j].read();
   return copy;
  }

  public int[] scan() {
```

```
    LabeledValue[] oldCopy, newCopy;
    oldCopy = collect();
    collect: while (true) {
     newCopy = collect();
     if (!equals(oldCopy, newCopy)) {
          oldCopy = newCopy;
          continue collect;
       }}
     return getValues(newCopy);
}}}
```

Wait-Free Snapshot

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}

public void update(int value) {
  int i = Thread.myIndex();
  int[] snap = this.scan();
  SnapValue oldValue = r[i].read();
  SnapValue newValue =
   new SnapValue(oldValue.label+1,
                 value, snap);
  r[i].write(newValue);
}

public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
      if (oldCopy[j].label != newCopy[j].label) {
        if (moved[j]) { // second move
            return newCopy[j].snap;
        } else {
        moved[j] = true;
        oldCopy = newCopy;
        continue collect;
    }}}
  return getValues(newCopy);
}}
```

•Consensus

Generic Consensus Protocol

```
abstract class ConsensusProtocol implements Consensus {
 protected Object[] proposed =
  new Object[N];

 private void propose(Object value) {
  proposed[ThreadID.get()] = value;
 }

 abstract public Object
          decide(Object value);
 }}
```

a two-dequeuer wait-free FIFO queue

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

assign atomically to 2 out of 3 array locations

```
class MultiConsensus extends …{
 Assign2 a = new Assign2(3, EMPTY);
 public Object decide(object value) {
  a.assign(i, i, i+1, i);
  int other = a.read((i+2) % 3);
  if (other==EMPTY||other==a.read(1))
   return proposed[i];
  else
   return proposed[j];
  }}
```

compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus
     extends ConsensusProtocol {
 private AtomicInteger r =
   new AtomicInteger(-1);
 public Object decide(object value) {
  propose(value);
  r.compareAndSet(-1,i);
  return proposed[r.get()];
 }
}
```

•Locks

•Sets

•Queues

A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}

public void enq(T x) throws FullException {
  lock.lock();
  try {
    if (tail - head == items.length)
       throw new FullException();
    items[tail % items.length] = x;
    tail++;
  } finally {
    lock.unlock();
  }
}

public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
```

```
  }
}
```

Lock-free 2-Thread Queue

For simplicity, only two threads. One thread enq only. The other deq only

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity];
     head++;
     return item;
}}
```

•Stacks

Locks（Chapter02、07）

•Peterson Lock (for two threads)

互斥、无死锁、无饥饿

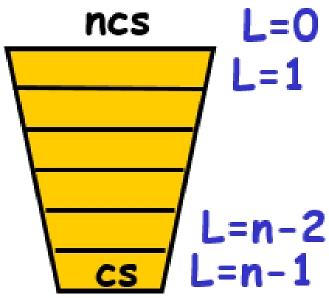
```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

•Filter Lock

无饥饿 但不公平

There are ***n-1*** "waiting rooms" called levels

- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through




```
class Filter implements Lock {
   int[] level;  // level[i] for thread i
   int[] victim; // victim[L] for level L

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

•Lamport's Bakery Lock

•Provides First-Come-First-Served

•How?

–Take a "number"

–Wait until lower numbers have been served

•Lexicographic order

–(a,i) > (b,j)

•If a > b, or a = b and i > j

```
class Bakery implements Lock {
   boolean[] flag;
   Label[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0], …,label[n-1])+1;
  while ($k flag[k]
           && (label[i],i) > (label[k],k));
 }
  public void unlock() {
   flag[i] = false;
 }


 …
 }
```

•Test-And-Set Lock

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
 }
}

class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

•Test-Test-And-Set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}//首先读取锁 让它看上去可用
   if (!state.getAndSet(true))//一旦看上去可用就立刻尝试获得
    return;
 }
}
```

•Exponential Backoff Lock

缺点：需要谨慎选择参数，难以移植

```
public class Backoff implements lock {
 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

•Anderson’s Array-Based Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 ThreadLocal<Integer> mySlot;

 public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false;
 }

 public unlock() {
```

```
    flags[(mySlot+1) % n] = true;
 }
```

空间 O(LN)

•CLH Queue Lock

```
class Qnode {
 AtomicBoolean locked =
   new AtomicBoolean(false);
}

class CLHLock implements Lock {
 AtomicReference<Qnode> tail
   = new AtomicReference<Qnode>(new Qnode());
 ThreadLocal<Qnode> myNode = new Qnode();
 ThreadLocal<Qnode> pred = null;

 public void lock() {
  myNode.locked.set(true);
  pred = tail.getAndSet(myNode);
  while (pred.locked) {}
 }

 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }

}
```

•MCS Queue Lock

```
class Qnode {
 boolean locked = false;
 qnode   next   = null;
}

class MCSLock implements Lock {
 AtomicReference tail;

 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }
 }
```

```
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
  qnode.next.locked = false;
 }
}
```

List-Based Sets

•Lock-Coupling List

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    while (curr.key <= key) {
        if (item == curr.item) {
            pred.next = curr.next;//LP
            return true;
        }
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();//LP
    }
    return false;

 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

•Optimistic List

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
```

```
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}

public boolean remove(Item item) {//add contain类似
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   }
   try {
    pred.lock(); curr.lock();
    if (validate(pred,curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        }
        else {
            return false;
        }
    }
   } finally {
    pred.unlock();
    curr.unlock();
   }
 }
}
```

•Lazy List

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
 }

public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
```

```
  return curr.key == key && !curr.marked;
}

public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   }
   try {
    pred.lock(); curr.lock();
    if (validate(pred,curr) {
        if (curr.item == item) {
            curr.mark = true;
            pred.next = curr.next;
            return true;
        }
        else {
            return false;
        }
    }
   } finally {
    pred.unlock();
    curr.unlock();
   }
 }
}
```

•Lock-Free List

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.attemptMark(succ, true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
     return true;
}}}

public boolean add(T item) {
 boolean splice;
 while (true) {
```

```
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
       return false;
    } else {
    Node node = new Node(item);
    node.next = new AtomicMarkableRef(curr, false);
    if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
 }}}

 public boolean contains(Tt item) {
     boolean marked;
     int key = item.hashCode();
     Node curr = this.head;
     while (curr.key < key)
       curr = curr.next;
     Node succ = curr.next.get(marked);
     return (curr.key == key && !marked[0])
   }

 public Window find(Node head, int key) {
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false}; boolean snip;
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
     succ = curr.next.get(marked);
     while (marked[0]) {
      snip = pred.next.compareAndSet(curr, succ, false, false);
      if (!snip) continue retry;
      curr = succ;
      succ = curr.next.get(marked);
     }
     if (curr.key >= key)
           return new Window(pred, curr);
        pred = curr;
        curr = succ;
      }
 }}
```

Queues and Stacks

•Queues

•**Two-Lock Queue Bounded**

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}

public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 if (mustWakeDequeuers) {
    deqLock.lock();
    try {
      notEmptyCondition.signalAll();
    } finally {
      deqLock.unlock();
    }
  }
}

public T deq() {
 T result;
 boolean mustWakeEnqueuers = false;
 deqLock.lock();
 try {
  while (permits.get() == capacity)
    notEmptyCondition.await();
  result = head.next.value;
  head = head.next;
  if (permits.getAndIncrement() == 0)
    mustWakeEnqueuers = true;
 } finally {
   deqLock.unlock();
 }
 if (mustWakeEnqueuers) {
    enqLock.lock();
    try {
      notFullCondition.signalAll();
```

```
    } finally {
      enqLock.unlock();
    }
  }
  return result;
}
```

•**Two-Lock Queue Unbounded**

```
public void enq(T x) {
 enqLock.lock();
 try {
  Node e = new Node(x);
  tail.next = e;
  tail = e;
 } finally {
   enqLock.unlock();
 }
}

public T deq() {
 T result;
 deqLock.lock();
 try {
  if (head.next==null){
      throw new EmptyException();
  }
  result = head.next.value;
  head = head.next;
 } finally {
   deqLock.unlock();
 }
  return result;
}
```

•**Lock-Free Queue Unbounded**

```
public void enq(T x) {
 Node e = new Node(x);
 while (true) {
   Node t = tail.get();
   Node n = t.next.get();
   if (t == tail.get()) {
     if (n == null) {
       if (t.next.compareAndSet(n, e)) {
          tail.compareAndSet(t, e);
          return;
       }
     } else {
       tail.compareAndSet(t, n);
     }
   }
 }
}
```

```
public T deq() throws EmptyException{
 while (true) {
   Node h = head.get();
   Node n = h.next.get();
   Node t = tail.get();
   if (h == head.get()) {
     if (h == t) {
       if (n == null)  throws new EmptyException();
       tail.compareAndSet(t, n);
     } else {
       T result = n.value;
       if (head.compareAndSet(h, n)) return result;
     }
   }
 }
}
```

•Lock-Free Queue-Solve ABA

```
public T deq() throws EmptyException{
 int[1] hStamp, nStamp, tStamp;
 while (true) {
   Node h = head.get(hStamp);
   Node n = h.next.get(nStamp);
   Node t = tail.get(tStamp);
   if (h == t) {
     if (n == null)  throws new EmptyException();
     tail.compareAndSet(t, n,
              tStamp[0], tStamp[0] + 1);
   } else {
     T result = n.value;
     if (head.compareAndSet(h, n,
                  hStamp[0], hStamp[0] + 1)) {
       free(h);
       return result;
     }
   }
 }
}
```

•Stacks

•Lock-Free Stack (Exponential Backoff)

```
public class Backoff {
 public void backoff() {
   sleep(random() % delay);
   if (delay < MAX_DELAY)
     delay = 2 * delay;
 }
}
```

```
public class LockFreeStack {
  private AtomicReference top =
    new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node))
  }
  public void push(T value) {
  Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff();//加Delay
  }}

  protected Node tryPop() throws EmptyException{
    Node oldTop = top.get();
    if (oldTop == null){
        throw new EmptyException();
    }
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop,newTop)){
        return oldTop;
    }else{
        return null;
    }
  }

  public T pop() throws EmptyException{
  Node node = new Node(value);
    while (true) {
      Node returnNode = tryPop()
      if (returnNode!= null) {
        return returnNode.value;
      } else backoff.backoff();//加Delay
    }
  }
```

•**Elimination Backoff Stack**

```
public T Exchange(T myItem, long nanos) throws      TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {EMPTY};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY:   // slot is free
            if (slot.compareAndSet(herItem, myItem, EMPTY, WAITING)) {
                 while (System.nanoTime() < timeBound){
                    herItem = slot.get(stampHolder);
```

```
                    if (stampHolder[0] == BUSY) {
                        slot.set(null, EMPTY);
                        return herItem;
                     }
                 }
                if (slot.compareAndSet(myItem, null, WAITING, EMPTY))
                    throw new TimeoutException();
                herItem = slot.get(stampHolder);
                slot.set(null, EMPTY);
                return herItem;
            }
           break;
        case WAITING: // someone waiting for me
            if (slot.compareAndSet(herItem, myItem,WAITING, BUSY))
                return herItem;
            break;
        case BUSY:    // others exchanging
            break;
        }
    }}

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

public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue = eliminationArray.visit(value,policy.Range);
      if (otherValue == null) {
         return;
    }
}

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

Bounded vs unbounded?

•Bounded

–Fixed capacity

–Good when resources an issue

•Unbounded

–Holds any number of objects

Blocking vs unblocking when deq from empty queue?

•Problem cases:

–Removing from empty pool

–Adding to full (bounded) pool

•Blocking

–Caller waits until state changes

•Non-Blocking

–Method throws exception