

2008 DaCapo



# Towards A Scalable Non-Blocking Coding Style

**Dr. Cliff Click**

Chief JVM Architect & Distinguished Engineer

[blogs.azulsystems.com/cliff](http://blogs.azulsystems.com/cliff)

Azul Systems

Jan 3, 2008



# What is Non-Blocking?



[www.azulsystems.com](http://www.azulsystems.com)

- Formally:
  - Stopping one thread will not prevent global progress
- Less formally:
  - No thread 'locks' any resource
    - (and then gets pre-empted)
  - No 'critical sections', locks, mutexes, spin-locks, etc
- Individual threads might starve

# XXX-Free Hierarchy



- Wait-Free
  - All threads complete in finite count of steps
  - No priority inversion possible
- Lock-Free (*this work*)
  - Every successful step makes Global Progress
  - But individual threads may starve
    - Hence priority inversion is possible
  - No live-lock
- Obstruction-Free
  - A single thread *in isolation* completes in finite count of steps
  - Threads may block each other
    - Hence live-lock is possible

- Multi-core is now almost unavoidable
- Larger core counts more common:
  - 8+ (X86), 64 (Sun/ Rock), 768 (Azul, more coming)
- Locking suffers serious contention issues
  - Amdahl's Law, etc
- Obstruction-free can live-lock
  - More prone with higher cpu count
  - Or higher thread count
- Wait-free algorithms behave the best
  - But tend to have very large big-O constants
  - And are *very* hard to code

- Most large-CPU count hardware is:
  - Parallel-read, Independent-write
  - Unlimited readers, free 'cache-hit-loads'
- Multiple CPUs writing to the same location serialize
  - Speed limited to '1-cache-miss-per-write' or '1-memory-bus-update-per-write'
- Must avoid all CPUs writing same location for independent operations
  - i.e., no shared counters, single lock-words, etc
- Classic reader/writer lock chokes w/ >100 CPUs
  - Contention on single reader-count word limits scaling

# Agenda



[www.azulsystems.com](http://www.azulsystems.com)

- Motivation
- **A Scalable Non-Blocking Coding Style**
- Example 1: BitVector
- Example 2: HashTable
- Example 3: Nearly FIFO Queue
- Summary

# Atomic Update



- Need some form of Atomic-Update
- Update 1 word IFF old-value is expected-value
- Generally Compare-And-Swap (CAS) or Load-Linked / Store-Conditional (LL/SC)
- LL/SC suffers from live-lock
- Both can suffer spurious failure on some hardware
  - Infinite spurious failures is live-lock(?)
  - Finite failures fixed with spin loop
- Useful if CAS does not spuriously fail (e.g. Azul)
  - Especially at high CPU count
  - If 1000 CPUs attempt update, 1 should succeed

# Atomic Update: Failure



[www.azulsystems.com](http://www.azulsystems.com)

- CAS failure returns old value (all? hardware)
  - Old value is evidence CAS did not fail spuriously
  - The “witness” - the “proof of failure”
- The witness not available **after** the CAS
  - Overwritten by another thread
- Java API mistake: witness turned into a boolean
  - Hence failure-for-cause can not be distinguished from spurious-failure
- Some algorithms need to spin strictly for this reason
- A CAS-returning-old-value could avoid the spin

# Towards A Scalable Lock-Free Coding Style



[www.azulsystems.com](http://www.azulsystems.com)

- Scalable: large count of words to update in parallel
  - An Array
- Lock-Free: Each CAS makes progress
  - CAS success is local progress
  - No spurious CAS failure
  - CAS failure means another CAS succeeded (global progress, local starvation)
- CAS single Array Words
- Construct algorithm from many 'steps'
  - Each CAS is a 'step' in a State Machine
- State-machine per-array-word

# Towards A Scalable Lock-Free Coding Style



[www.azulsystems.com](http://www.azulsystems.com)

- Fast: no indirections in common case
  - Directly reach main array in 1 step
- Array size?
  - *Make array growable*
- Support array resize via State Machine
  - Really: array-copy while in use
- All words are independent
  - Copy is parallel, incremental, concurrent

# Towards A Scalable Lock-Free Coding Style



[www.azulsystems.com](http://www.azulsystems.com)

- Hard bit during resize:
  - Copy without losing late-write
- Fix: “mark” payload with no-more-updates flag
- Payload still visible
- Updater's of marked payload must **copy** then update in new array
- Readers seeing mark: finish **copy** then read in new array

# Agenda



[www.azulsystems.com](http://www.azulsystems.com)

- Motivation
- A Scalable Non-Blocking Coding Style
- **Example 1: BitVector**
- Example 2: HashTable
- Example 3: Nearly FIFO Queue
- Summary

# Example 1: BitVector



- Size:  $O(\text{max element})$ 
  - Auto-resizing
- Supports concurrent insert, remove, test
- Obvious implementation:
  - Array of 64-bit payload words
  - Index & Mask accessors
- How to 'mark' payload?
  - Steal 1 bit out of 64
  - MOD 63 to index words
  - Actually: avoid slow MOD by moving every 64<sup>th</sup> bit to recursive bitvector
- Code up in SourceForge, high-scale-lib

# Example 1: BitVector



- Basic get & test/set (using MOD)

```
boolean get( int x ) {  
    long[] A = _A;  
    int idx = x/63;  
    if( idx >= A.length)  
        return false;  
    mask = 1L << (x%63);  
    int old = A[idx];  
    if( old < 0 )  
        return copy(x).get(x);  
    return A[idx] & mask;  
}
```

```
boolean test_set( int x ) {  
    long[] A = _A; // read once  
    int idx = x/63;  
    if( idx >= A.length )  
        return grow(x);  
    while( true ) { // spin loop  
        int old = A[idx];  
        if( old < 0 ) // marked?  
            return copy(x).test_set(x);  
        if( (old & mask) != 0 )  
            return true;  
        if( CAS(A,idx,old,old|mask) )  
            return false;  
    }  
}
```

# Example 1: BitVector

- Read array once – it is changing!

```
boolean get( int x ) {
    long[] A = _A;
    int idx = x/63;
    if( idx >= A.length)
        return false;
    mask = 1L << (x%63);
    int old = A[idx];
    if( old < 0 )
        return copy(x).get(x);
    return A[idx] & mask;
}

boolean test_set( int x ) {
    long[] A = _A; // read once
    int idx = x/63;
    if( idx >= A.length )
        return grow(x);
    while( true ) { // spin loop
        int old = A[idx];
        if( old < 0 )
            return copy(x).test_set(x);
        if( (old & mask) != 0)
            return true;
        if( CAS(A,idx,old,old|mask))
            return false;
    }
}
```

# Example 1: BitVector



- Out-of-bounds triggers resize

```
boolean get( int x ) {  
    long[] A = _A;  
    int idx = x/63;  
    if( idx >= A.length)  
        return false;  
    mask = 1L << (x%63);  
    int old = A[idx];  
    if( old < 0 )  
        return copy(x).get(x);  
    return A[idx] & mask;  
}
```

```
boolean test_set( int x ) {  
    long[] A = _A; // read once  
    int idx = x/63;  
    if( idx >= A.length )  
        return grow(x);  
    while( true ) { // spin loop  
        int old = A[idx];  
        if( old < 0 )  
            return copy(x).test_set(x);  
        if( (old & mask) != 0 )  
            return true;  
        if( CAS(A,idx,old,old|mask))  
            return false;  
    }  
}
```

# Example 1: BitVector



- 'Mark' triggers copy & retry

```
boolean get( int x ) {  
    long[] A = _A;  
    int idx = x/63;  
    if( idx >= A.length)  
        return false;  
    mask = 1L << (x%63);  
    int old = A[idx];  
    if( old < 0 )  
        return copy(x).get(x);  
    return A[idx] & mask;  
}
```

```
boolean test_set( int x ) {  
    long[] A = _A; // read once  
    int idx = x/63;  
    if( idx >= A.length )  
        return grow(x);  
    while( true ) { // spin loop  
        int old = A[idx];  
        if( old < 0 )  
            return copy(x).test_set(x);  
        if( (old & mask) != 0 )  
            return true;  
        if( CAS(A,idx,old,old|mask))  
            return false;  
    }  
}
```

# Example 1: BitVector



- Failed CAS spins – BUT!
  - Means another thread made progress

```
boolean get( int x ) {
    long[] A = _A;
    int idx = x/63;
    if( idx >= A.length)
        return false;
    mask = 1L << (x%63);
    int old = A[idx];
    if( old < 0 )
        return copy(x).get(x);
    return A[idx] & mask;
}
```

```
boolean test_set( int x ) {
    long[] A = _A; // read once
    int idx = x/63;
    if( idx >= A.length )
        return grow(x);
    while( true ) { // spin loop
        int old = A[idx];
        if( old < 0 )
            return copy(x).test_set(x);
        if( (old & mask) != 0)
            return true;
        if( CAS(A,idx,old,old|mask))
            return false;
    }
}
```

# Example 1: BitVector



- Almost as fast as plain BitVector
  - Normal load & mask for get/set
  - Range check
  - Extra '<0' test (triggers copy & retry)
  - Set uses CAS spin-loop
- Copy: CAS sign-bit to stop further updates
  - Then copy word to new array
  - Repeat operation on new array
  - State Machine!
    - per Array word

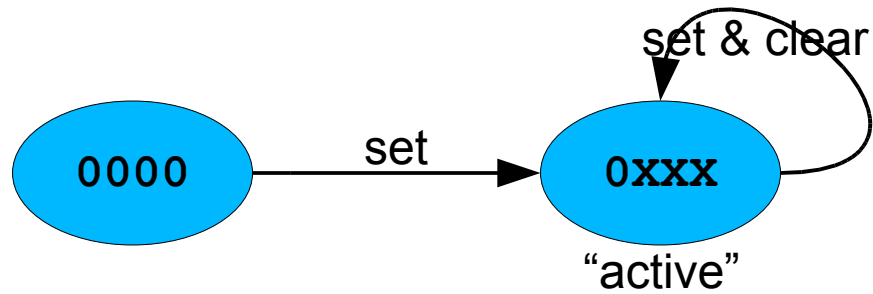
# BitVector State Machine



[www.azulsystems.com](http://www.azulsystems.com)

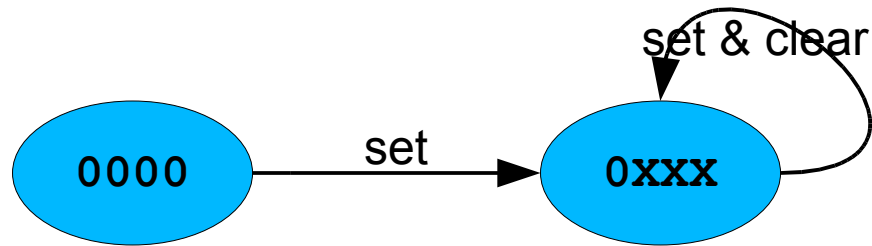
0000  
"initial"

# BitVector State Machine



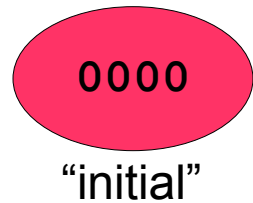
**A: Normal operations**

# BitVector State Machine



A: Normal operations

Out-of-Bounds set  
triggers resize!

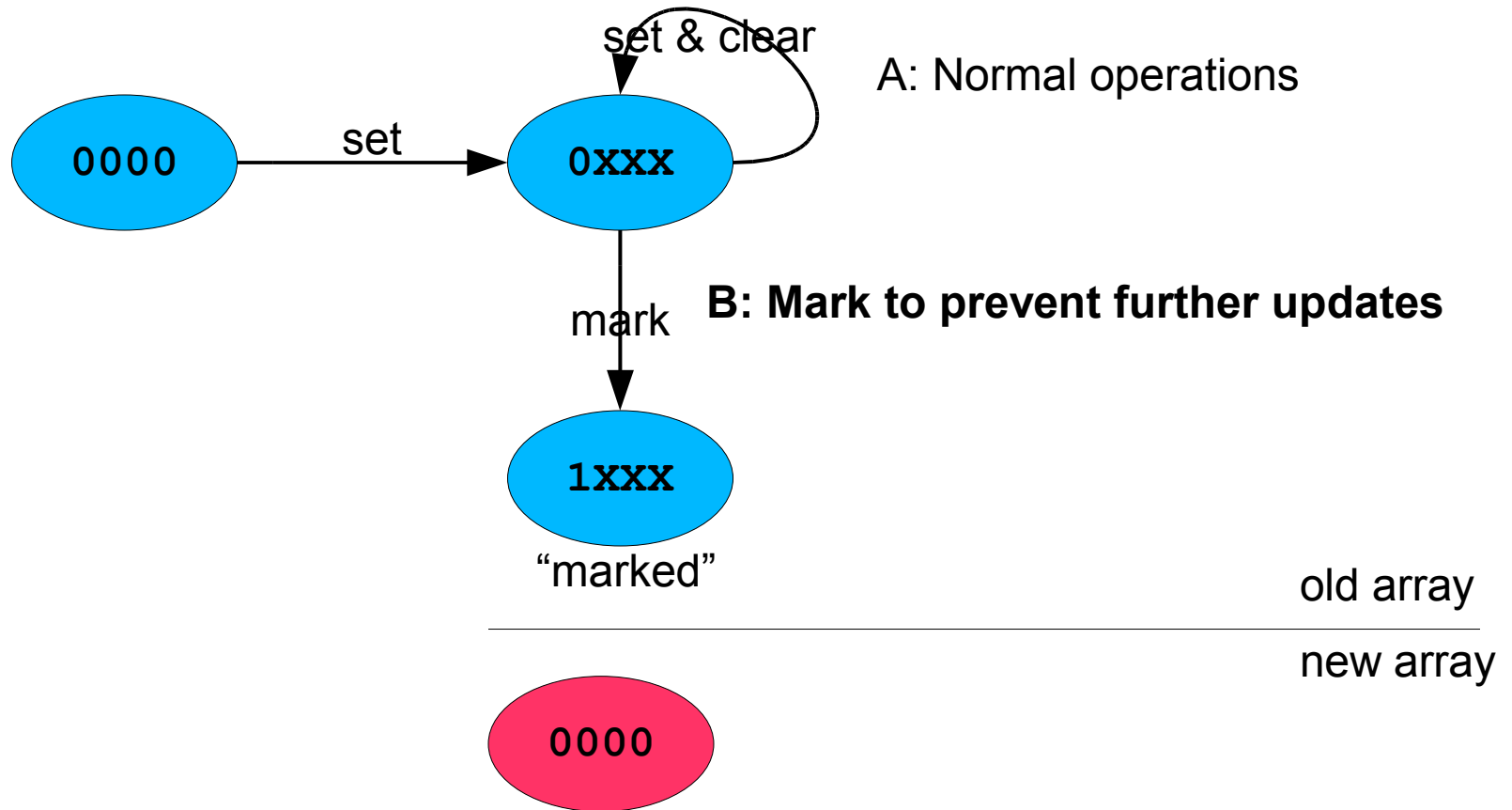


old array  
new array

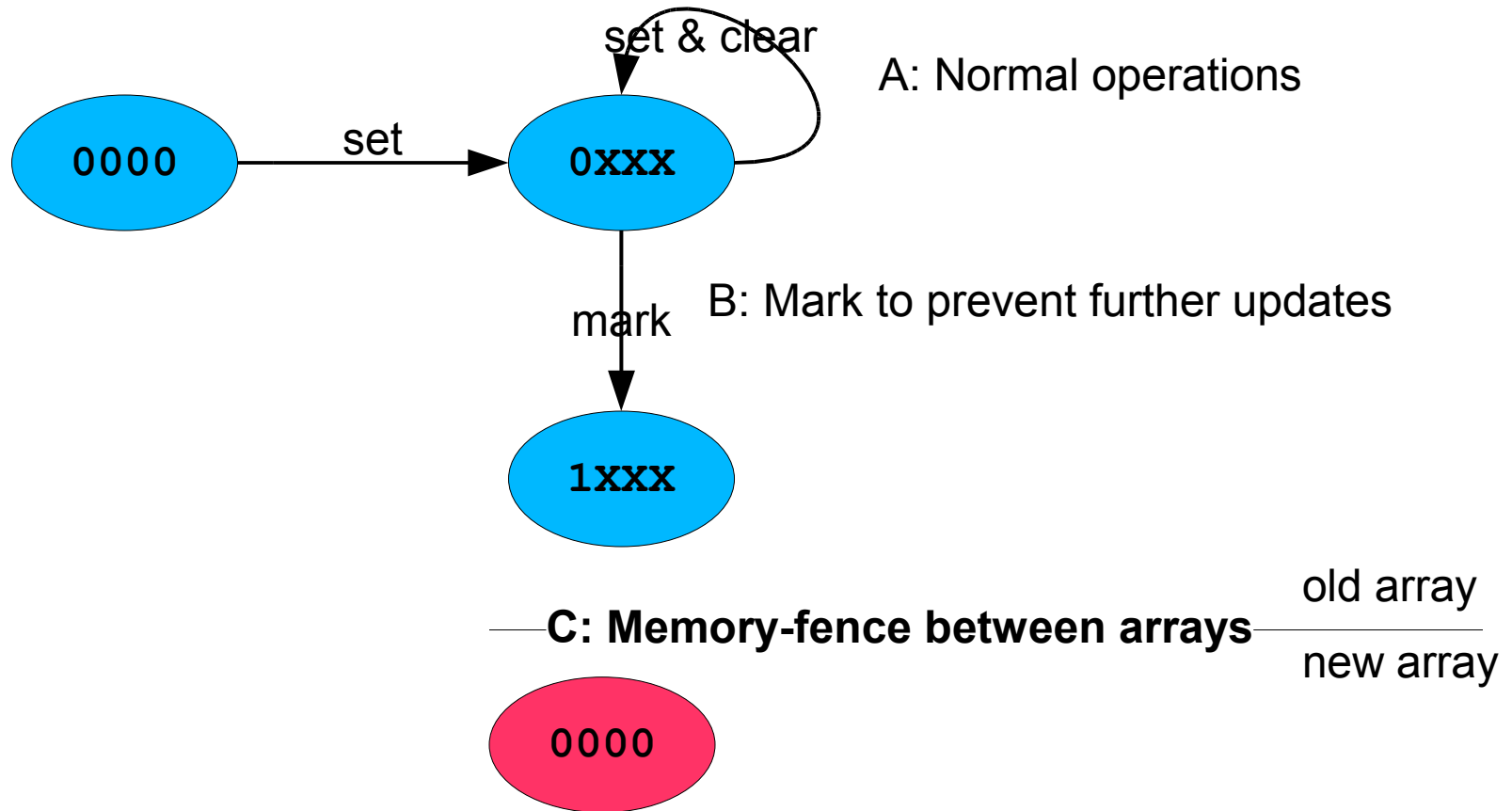
# BitVector State Machine



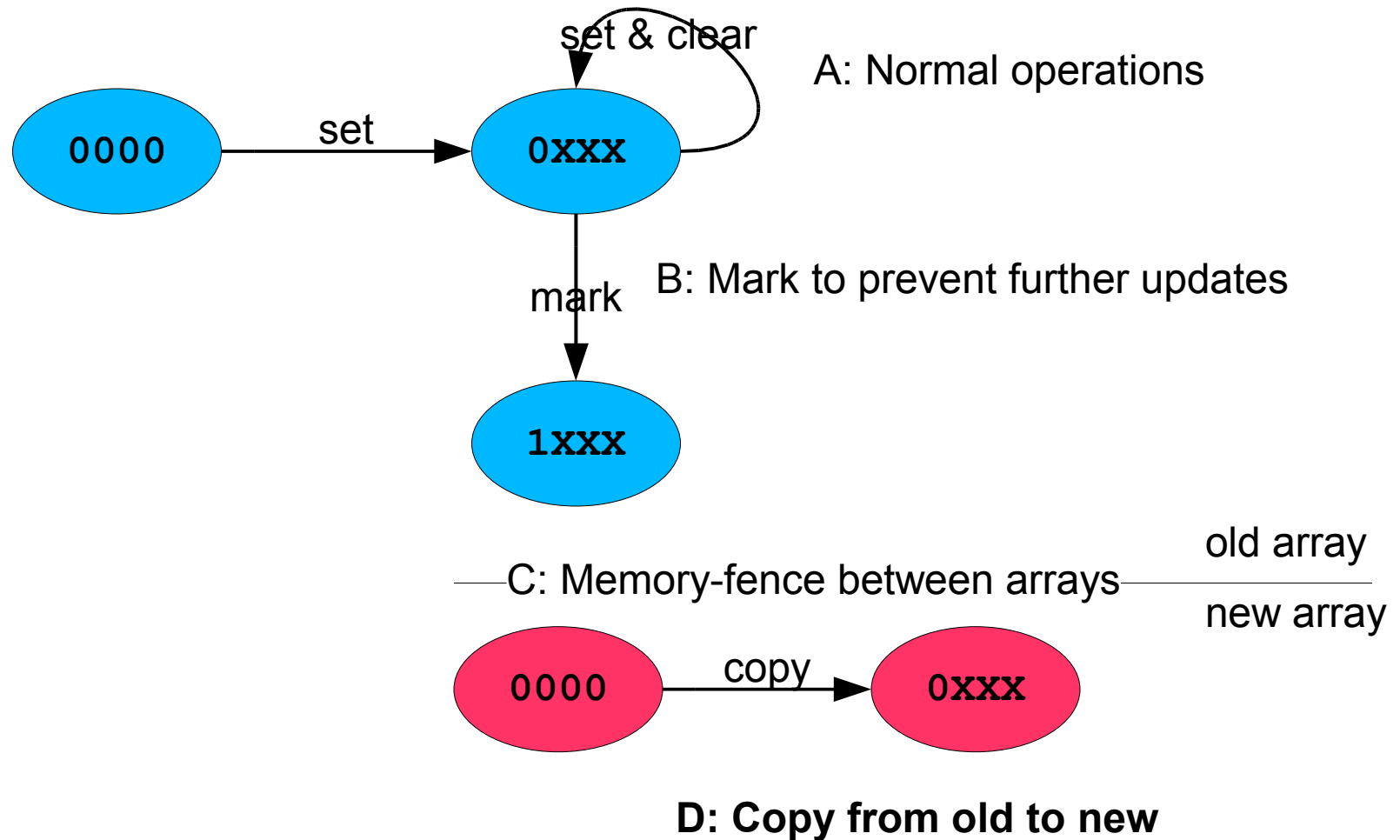
www.azulsystems.com



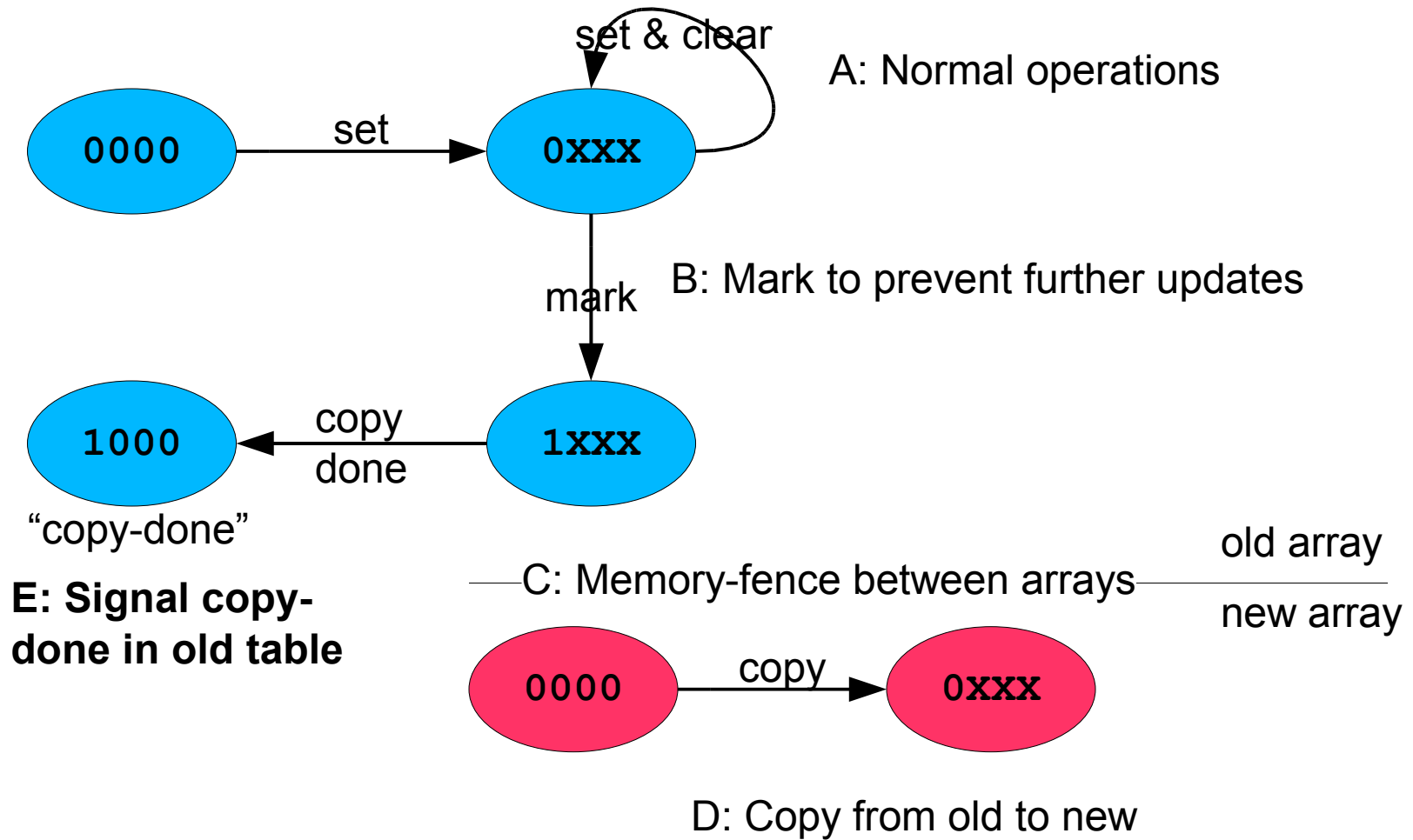
# BitVector State Machine



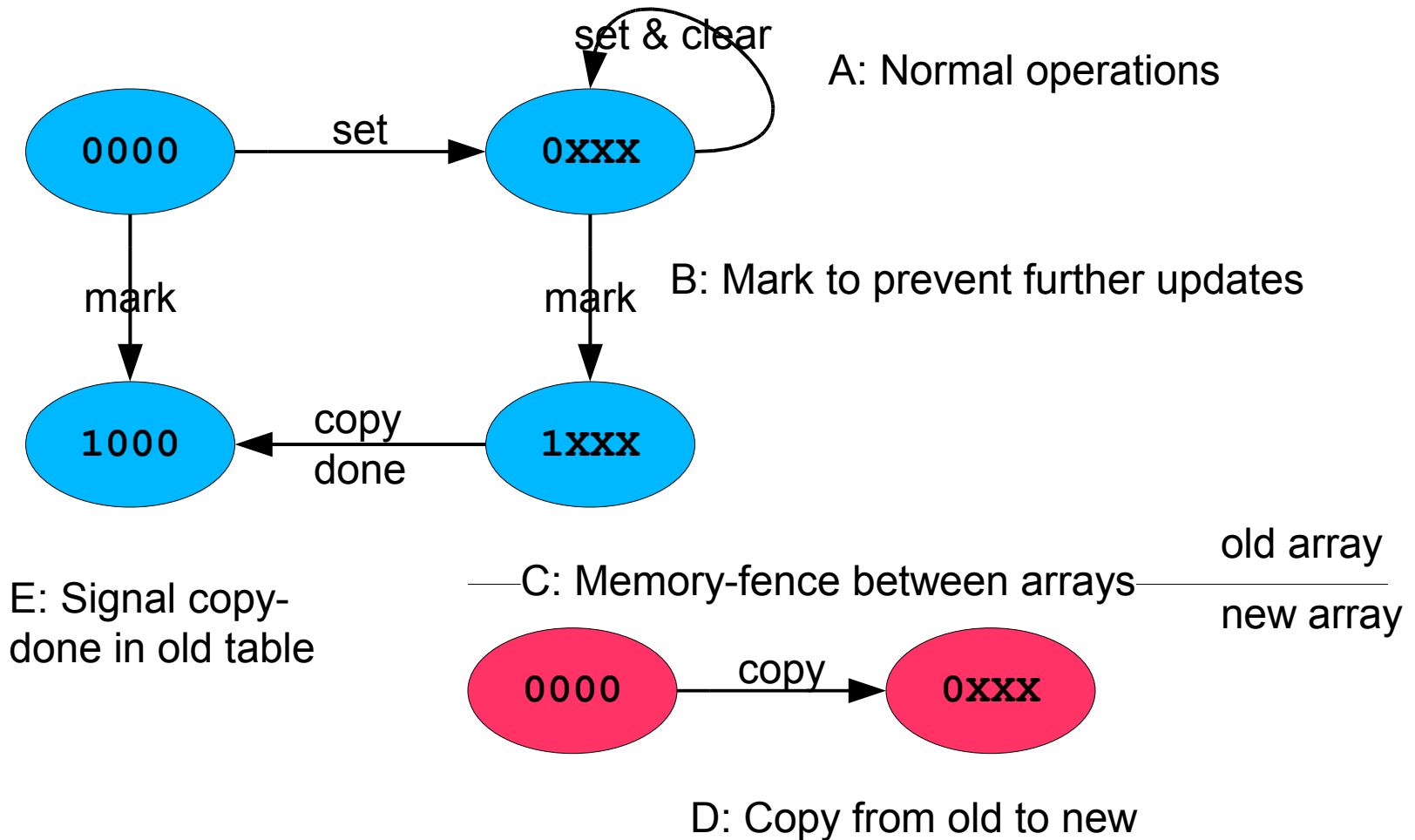
# BitVector State Machine



# BitVector State Machine



# BitVector State Machine



# Resize - motivation



- Triggered by adding larger element
- **Copy** each word before get/set
- Pay indirection even after **copy**
  - Visit old table, fence, operate on new table
- So need to **copy** all words eventually, and
- **Promote**: make new array **the** array
  - No more indirection
- **Policy?** How to copy all words?
- Visiting threads can “copy some words”
  - Or background threads, or only-writers, etc
- Good standard engineering, nothing special

# Resize - Mechanics



[www.azulsystems.com](http://www.azulsystems.com)

- Helpers CAS-up a “promise to copy” counter
  - CAS-up by fixed N (e.g. 16 words)
- Helpers **copy** words via State Machine
- Helpers CAS-up “done work” counter
  - On transition to “copy-done” state
- Promote when “done work” == A.length
- Helper stalled? (promises but never copies)
- Allow “double-promise”!
  - Worst case: each thread can complete entire copy
- Eventually, copy completes & array promotes

# Coding Style Elements



[www.azulsystems.com](http://www.azulsystems.com)

- Large array for parallel update
- State Machine per-word
  - Successful CAS is FSM transition
  - Failed CAS causes retry
    - (but another thread made progress)
- 'Mark' payload words to stop 'late updates'
- Array copy for Resize (or flush stale, etc)
  - Resize is parallel, incremental, concurrent
  - Copy part of State Machine
  - Unrelated threads can progress during resize
  - Fence between old and new tables

# Agenda



[www.azulsystems.com](http://www.azulsystems.com)

- Motivation
- A Scalable Non-Blocking Coding Style
- Example 1: BitVector
- **Example 2: HashTable**
- Example 3: Nearly FIFO Queue
- Summary

# Example 2: HashTable



[www.azulsystems.com](http://www.azulsystems.com)

- Array of K/V Pairs
  - Keys in even slots, Values odd slots
  - CAS each word separately
  - Value can be 'Tombstone'
  - Key & Value both start as **null**
- Mark payload by 'boxing' values
- Copy on resize, or to flush stale keys
- Supports concurrent insert, remove, test, resize
- Linear scaling on Azul to 768 CPUs
  - > billion reads/sec AND
  - > 10million updates/sec
- Code up in SourceForge, high-scale-lib

# “Uninteresting” Details



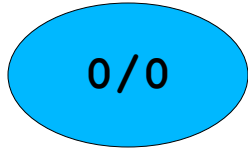
[www.azulsystems.com](http://www.azulsystems.com)

- Closed Power-of-2 Hash Table
  - Reprobe on collision
  - Stride-1 reprobe: better cache behavior
  - (complicated argument about  $2^n$  vs prime goes here)
- Key & Value on same cache line
- Hash memoized
  - Should be same cache line as  $K + V$
  - But hard to do in pure Java
- No allocation on `get()` or `put()`
- Auto-Resize

# HashTable State Machine



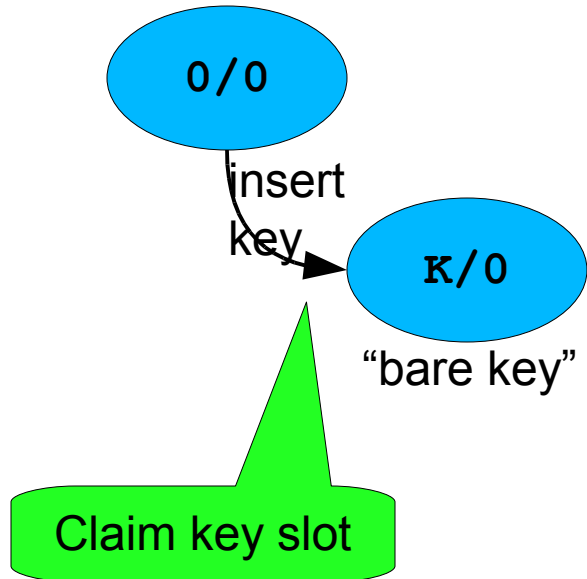
[www.azulsystems.com](http://www.azulsystems.com)



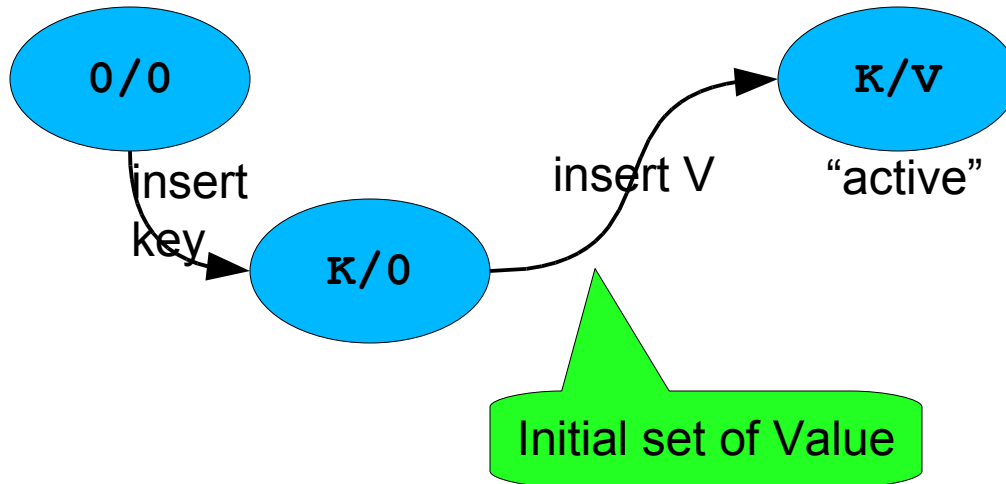
“initial”

- Already probed table, missed
- Found proper empty slot
- Ready to claim slot for this Key

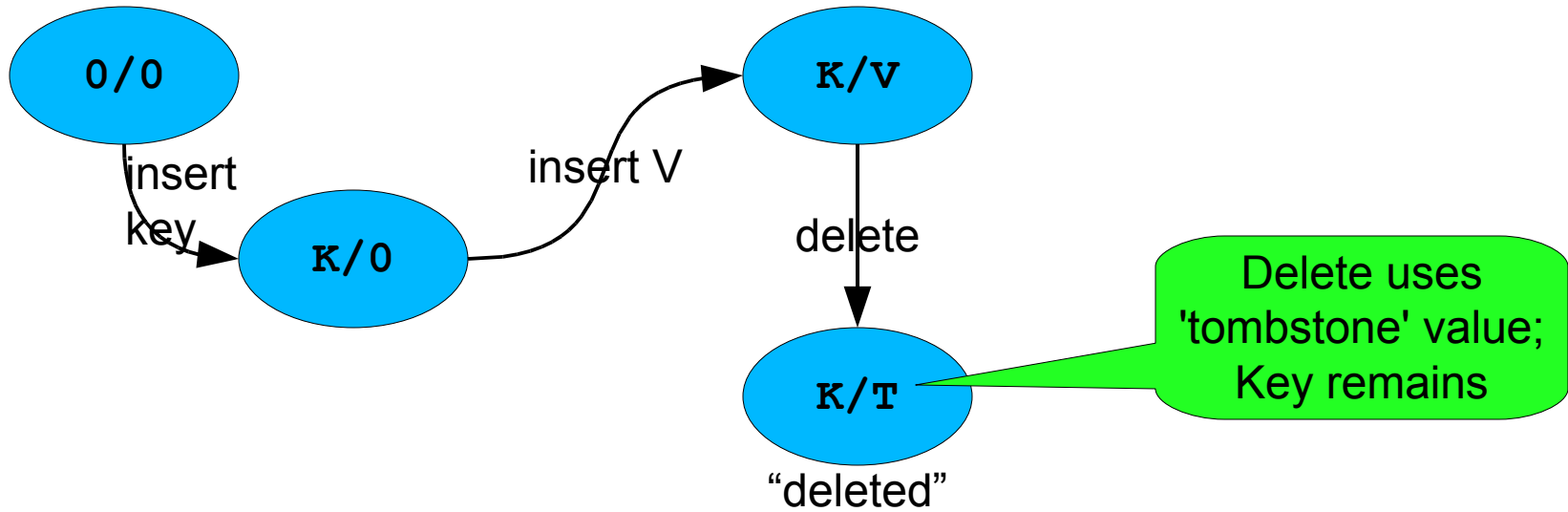
# HashTable State Machine



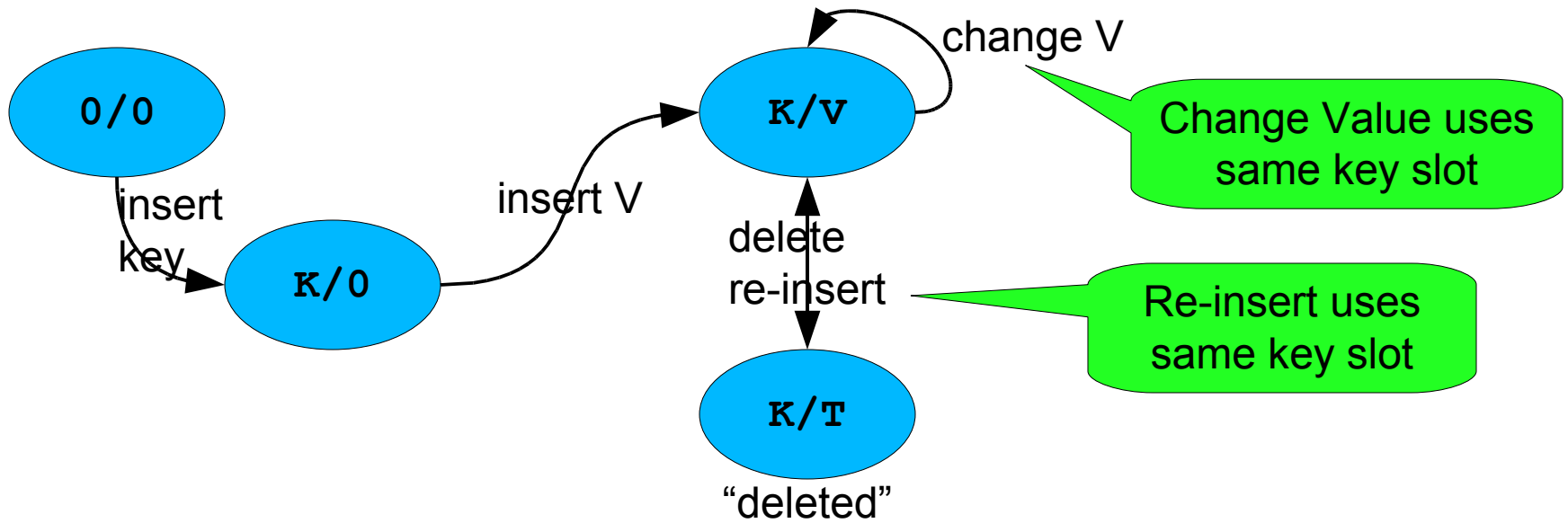
# HashTable State Machine



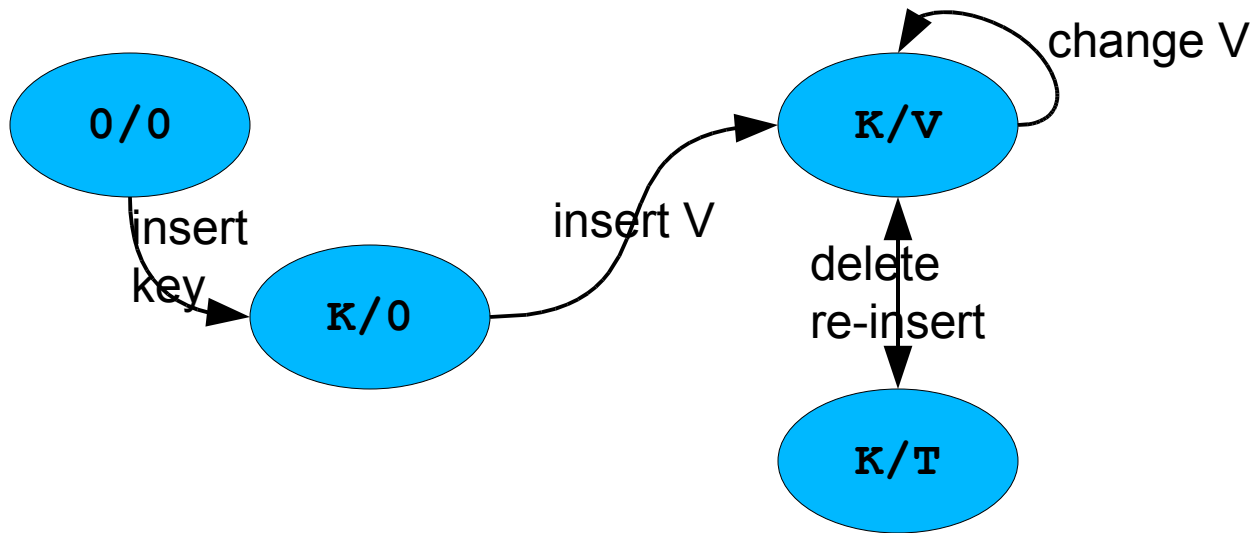
# HashTable State Machine



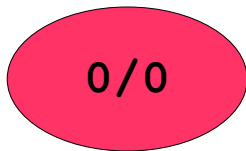
# HashTable State Machine



# HashTable State Machine



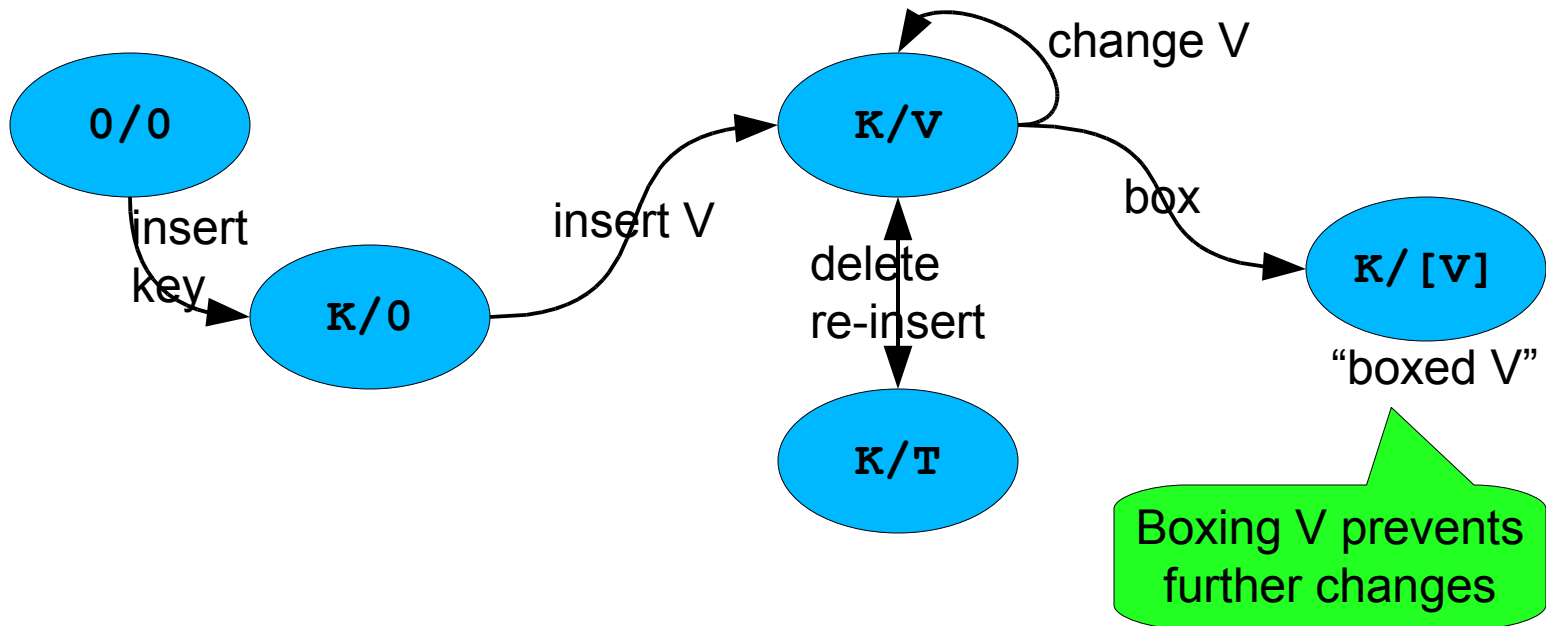
Resize triggered,  
new array created



"initial"

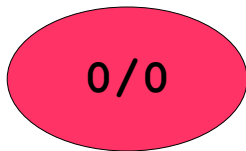
old array  
-----  
new array

# HashTable State Machine

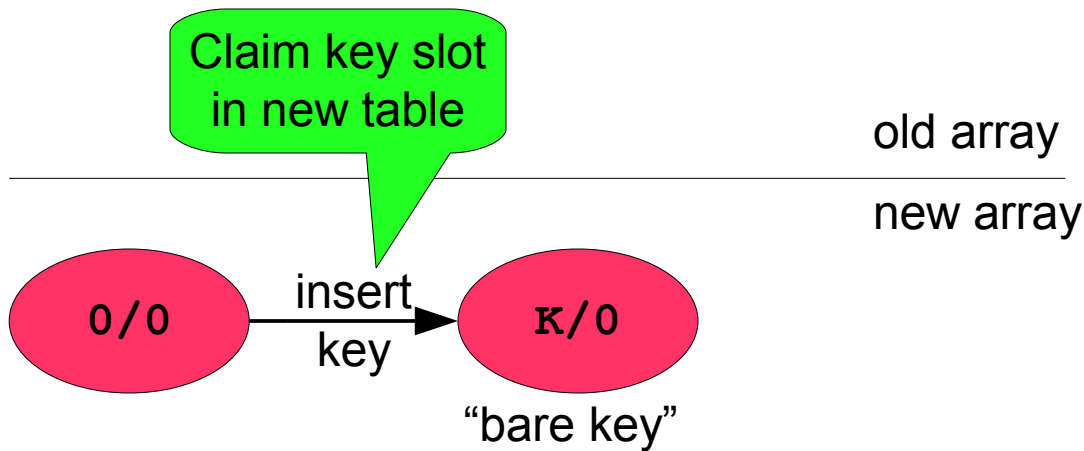
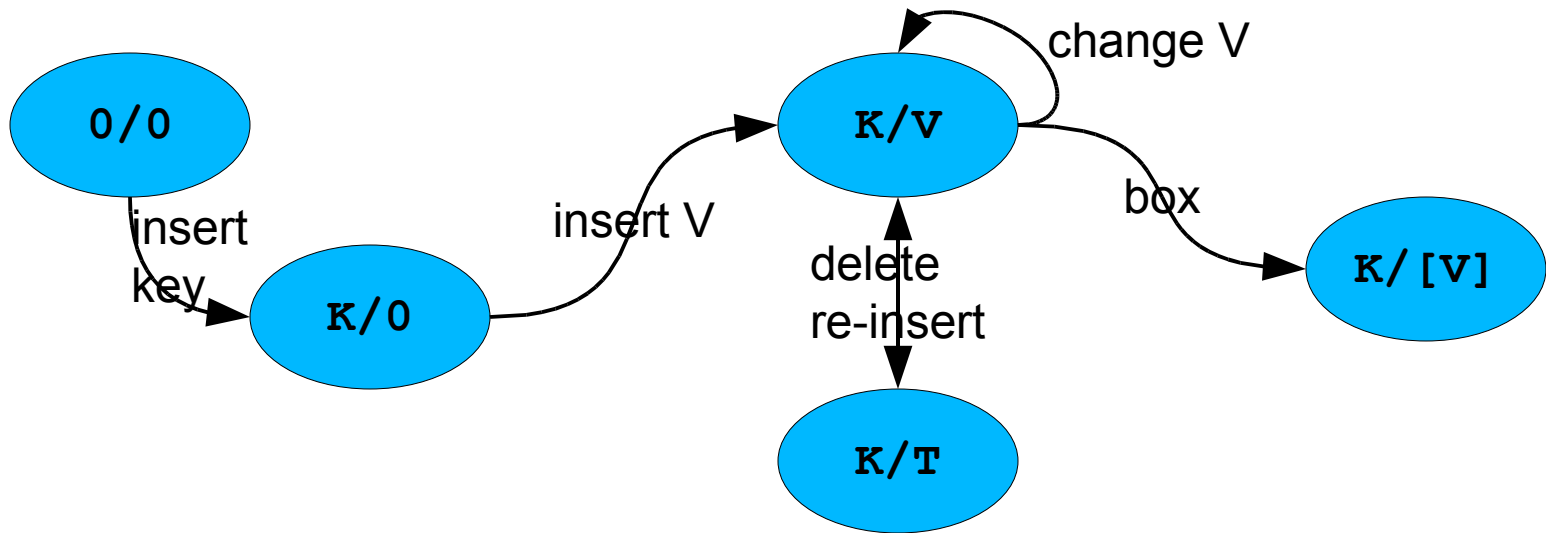


old array

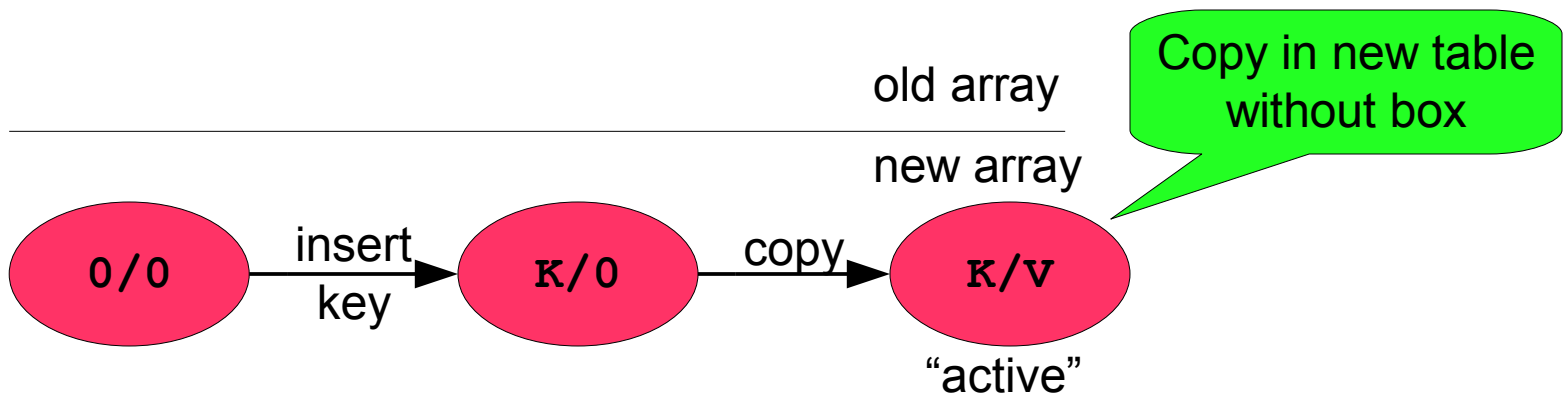
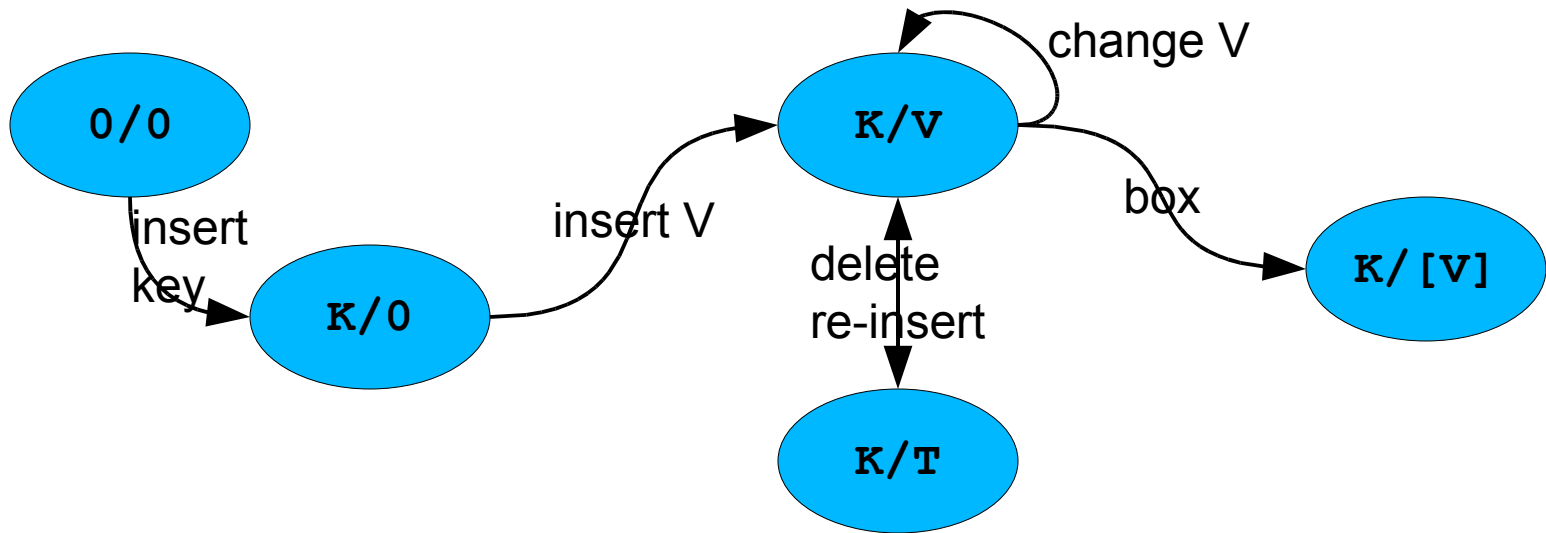
new array



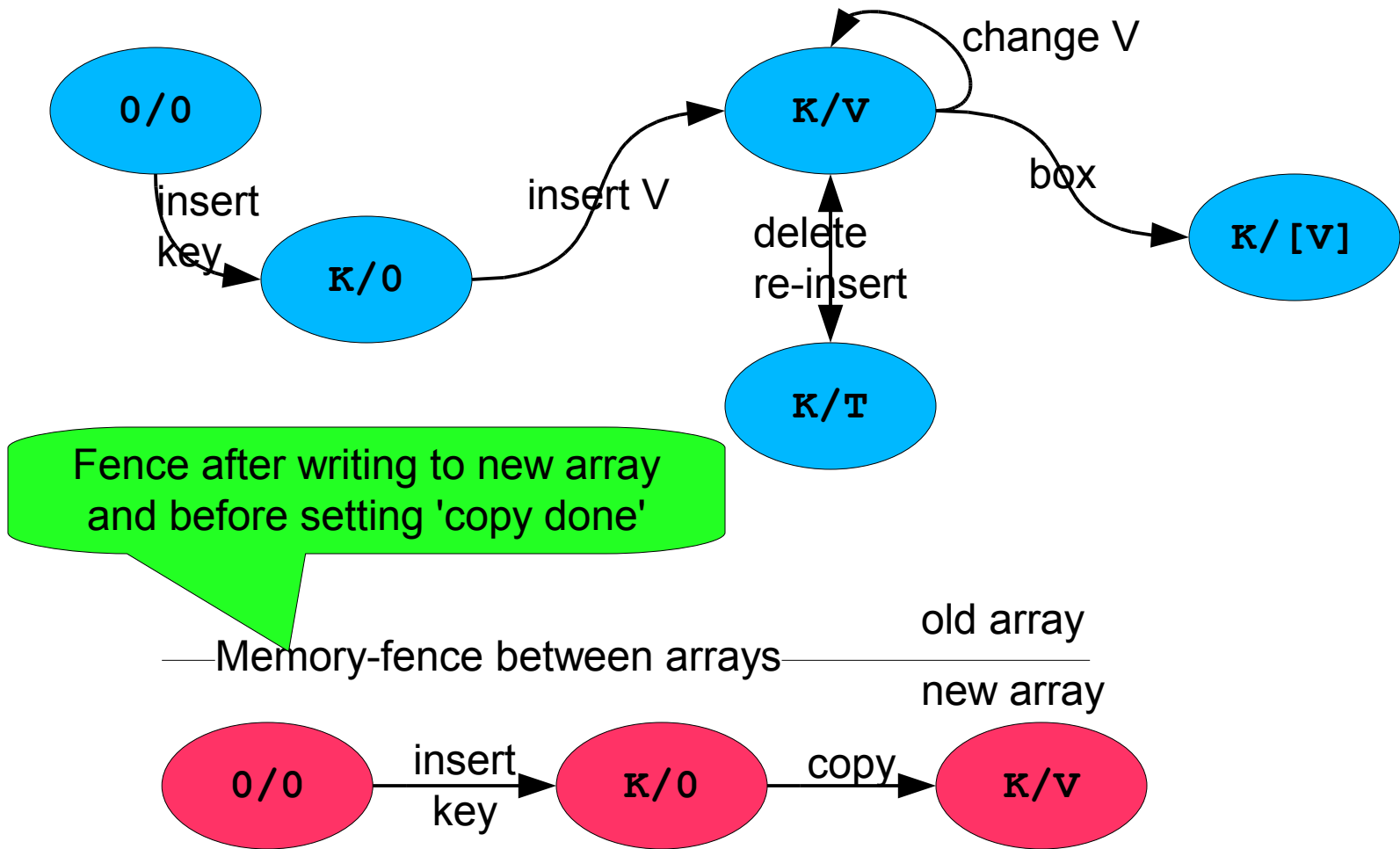
# HashTable State Machine



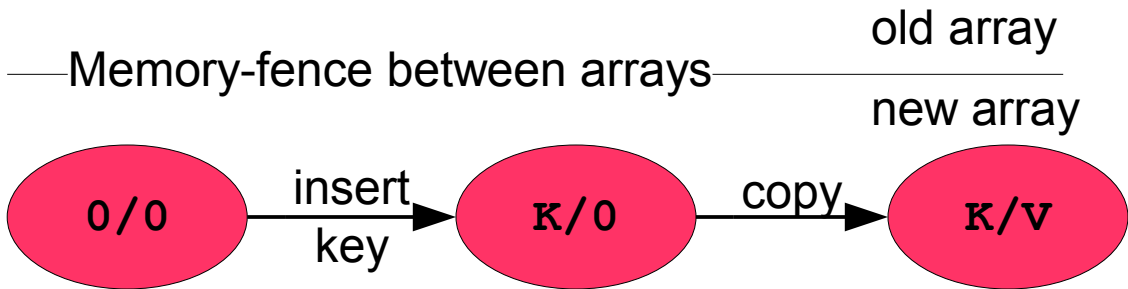
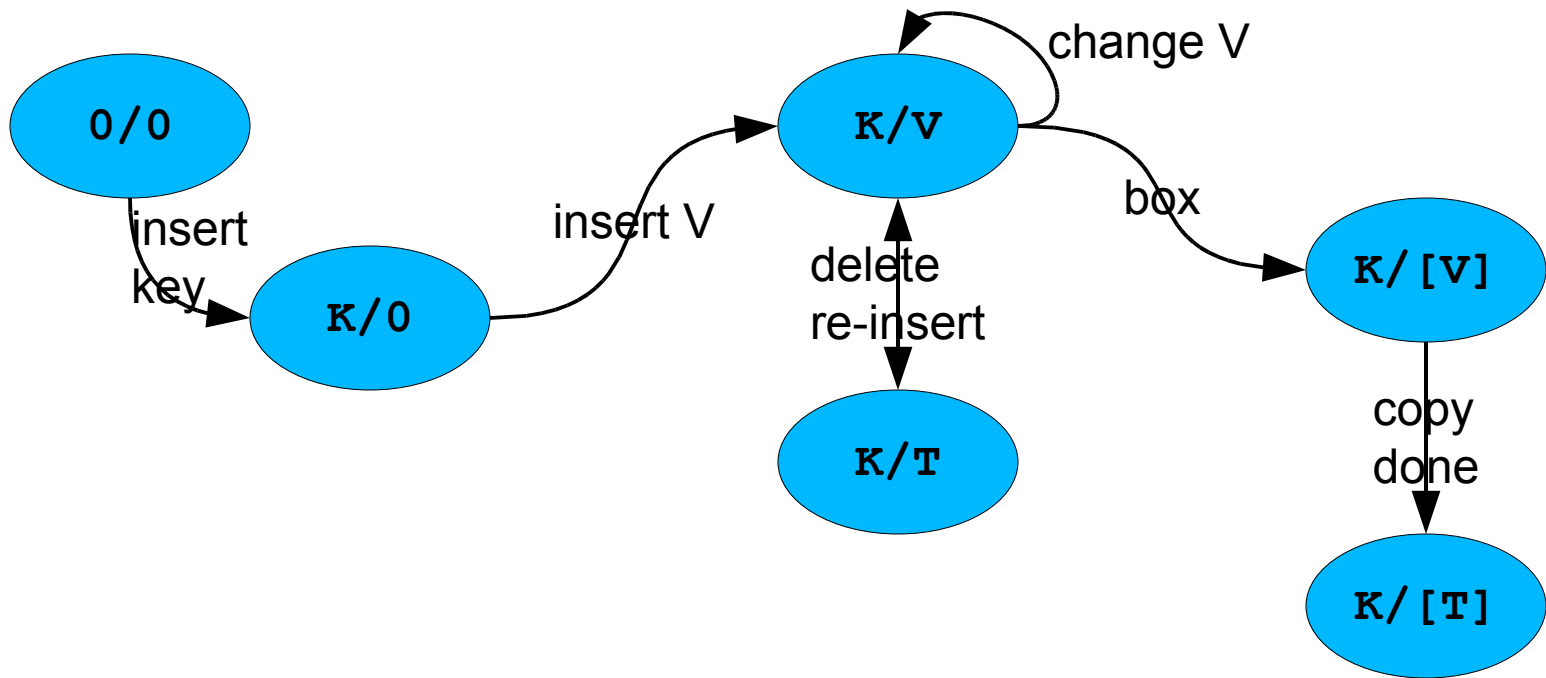
# HashTable State Machine



# HashTable State Machine



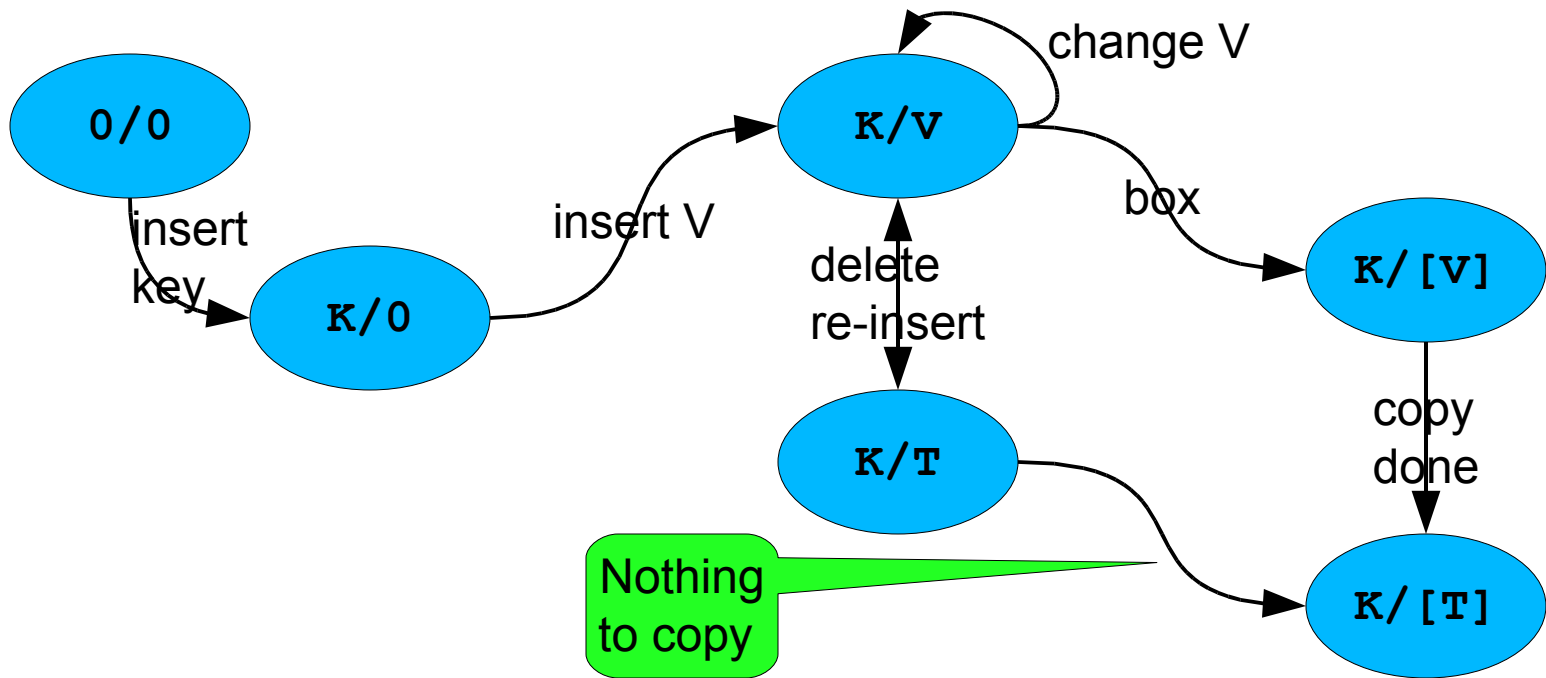
# HashTable State Machine



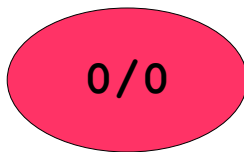
“copy done”

Final state: “new Array has Value”

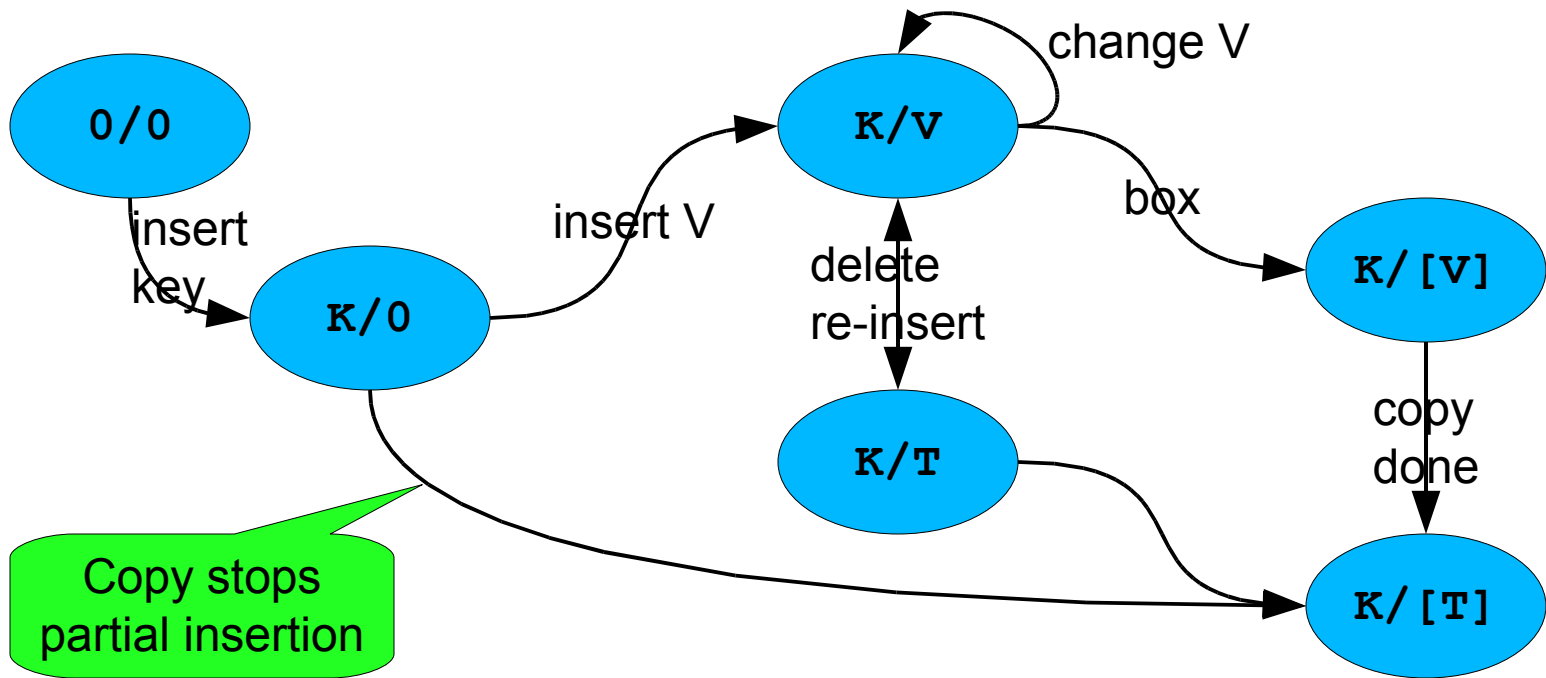
# HashTable State Machine



—Memory-fence between arrays—  
old array  
new array

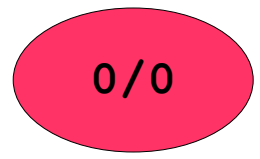


# HashTable State Machine

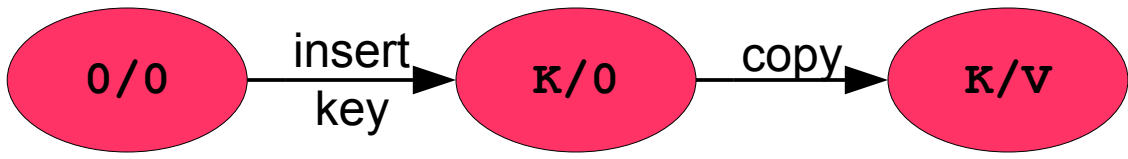
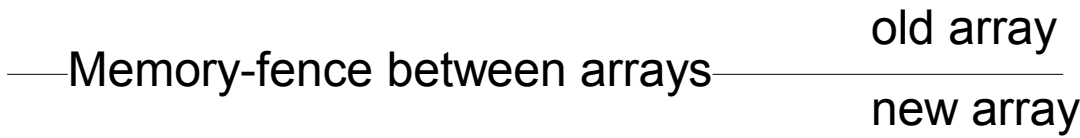
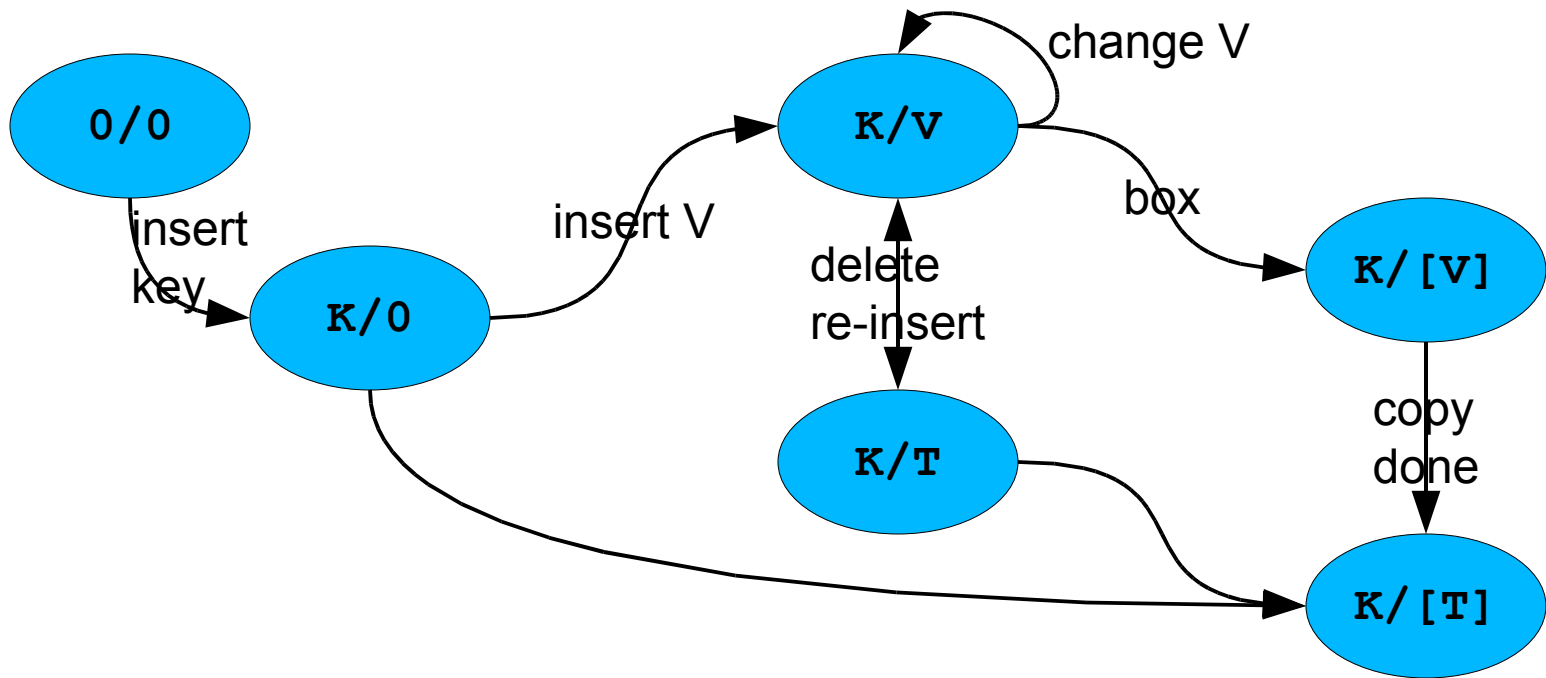


Copy stops partial insertion

— Memory-fence between arrays —  
old array  
new array



# HashTable State Machine



# Agenda



[www.azulsystems.com](http://www.azulsystems.com)

- Motivation
- A Scalable Non-Blocking Coding Style
- Example 1: BitVector
- Example 2: HashTable
- **Example 3: Nearly FIFO Queue**
- Summary

# Example 3: Nearly FIFO Queue



[www.azulsystems.com](http://www.azulsystems.com)

- Concurrent near-FIFO Queue
  - e.g. producer / consumer worklist
  - Producers & consumers are large thread pools
- Scaling bottleneck:
  - Locking or single word CAS on push & pop
- So could stripe Queue:
  - Hash to random Queue
  - Many short Queues
  - Many locks or many words to CAS
  - Pick at random to push or pop
  - Must search for not-full or not-empty

# Example 3: Nearly FIFO Queue



- Stripe Ad-Absurdum
  - 1000's of CPUs need 1000's of Queues
  - Queues get ever-smaller
- Single-entry Queue: either full or empty
  - Implement as a single word
  - Either **null** or value
- Array of single word Queues
- Producers start @ random index
  - Search for null, CAS down value
- Consumers start @ random index
  - Search for value, CAS down null

# Example 3: Nearly FIFO Queue



- Nearly FIFO:
  - Consumers *must* advance scan point
  - Means any value in array gets visited eventually
- Tricky bit: correct array size for efficiency
  - Too small, table gets full, producers spin uselessly
  - Too large, table is empty, consumers scan uselessly
- Array copy & promote is easier:
  - Risk: late insert in old array just prior to promote abandons value
  - Consumers fill old array with 'tombstone'
  - Promote when old array 'stoned'
- Still need feedback mechanisms on P/C threadpools

# Example 3: Nearly FIFO Queue



[www.azulsystems.com](http://www.azulsystems.com)

- Work in progress, no code yet...
- But out of time anyways ;-)
- Nice idea, hope it pans out

# Agenda



[www.azulsystems.com](http://www.azulsystems.com)

- Motivation
- A Scalable Non-Blocking Coding Style
- Example 1: BitVector
- Example 2: HashTable
- Example 3: Nearly FIFO Queue
- **Summary**

# Summary



www.azulsystems.com

- Lock-Free
- Highly scalable (proven scalable to ~1000 CPUs)
- Use large array for data
  - Allows fast parallel-read
  - Allows parallel, incremental, concurrent copy
- Use State Machine to control writes
  - FSM-per-word
  - Successful CAS advances FSM
  - Failed CAS retries
- During copy, FSM includes words from both arrays

<http://www.azulsystems.com/blogs/cliff>

[WWW.AZULSYSTEMS.COM](http://WWW.AZULSYSTEMS.COM)

THE ERA OF UNBOUND COMPUTE IS NOW

Thank You

