Towards a Scalable Non-Blocking Coding Style

Dr. Cliff Click, Distinguished Engineer

Azul Systems

http://blogs.azulsystems.com/cliff

The Computer Revolution is Here

We already did the 0->1 cpu transition

Concurrent Programming is Now 'The Norm'

and very hard to do

We're doing the 1->2 cpu transition

Scalable Concurrent Programming

is even harder

Time to think about the 2->N cpu transition

Here is a different way of thinking about the problem

What is A Non-Blocking Algorithm?

- Formally:
  - Stopping one thread will not prevent global progress
- Less formally:
  - No thread 'locks' any resource
    - and then gets pre-empted by OS
    - Or blocked in I/O, etc
  - No 'critical sections', locks, mutexs, spin-locks, etc
- Individual threads might starve

XXX-Free Hierarchy

- **Wait**-Free Algorithms (the best)
  - All threads complete in finite count of steps
  - Low priority threads cannot block high priority threads
  - 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

Motivation

- 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
- Would like to write correct code without locks!
- Obstruction-free can live-lock
  - More prone with higher cpu count
  - Or higher thread count
- Wait-free algorithms behave the best
  - But tend to be ***slow***
  - And are ***very*** hard to code
    - Handful of people on the planet can write these

Scalable

- Most large-CPU count shared-memory hardware is:
  - Parallel-read, Independent-write
- Multiple CPUs reading the same location is fast
  - Free 'cache-hitting-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/ >50 CPUs
  - Contention on single reader-count word limits scaling

Agenda

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

Parts we need...

- An Array to hold all Data
  - Fast parallel (scalable) access
- Atomic-update on single Array Words
  - java.util.concurrent.Atomic.*
  - “No spurious failure CAS”
- A Finite State Machine
  - Replicated per array word (or small set of words)
  - Use Atomic-Update to 'step' in the FSM
- Construct algorithm from many FSM 'steps'
  - Lock-Free: Each CAS makes progress
  - CAS success is local progress
  - CAS failure means another CAS succeeded (global progress, local starvation)

How Big is the Array?

- *Don't answer that: Make array growable*
  - Resize array as needed
  - Common operation for Collection classes
- Support array resize via State Machine
  - Really: array-copy while in use
  - All array words are independent
  - Copy is parallel, incremental, concurrent
- But mostly operate without a copy-in-progress
  - So the common situation is simple, fast

Concurrent Array Resize

- Copy old Array into a new larger Array
- The hard part during a resize operation:
  - Copy without losing any late-writes to old Array
- Fix: “mark” old Array words with no-more-updates flag
  - Payload still visible through the “mark”
- Updaters' of marked payload must
  copy then update in new array
- Readers' seeing mark must
  copy then read in new array

Atomic Update

- Need some form of Atomic-Update
  - `java.util.concurrent.atomic.*`
- Update 1 word IFF old-value is equal to expected-value
- Generally Compare-And-Swap (CAS, Azul/Sparc/X86) or Load-Linked / Store-Conditional (LL/SC, IBM)
- Common Hardware Limitations
  - LL/SC suffers from live-lock
  - Both CAS & LL/SC 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

- CAS failure returns old value on most (all?) hardware?
  - Old value is evidence CAS did not fail spuriously
  - The “witness” - the “proof of failure”
  - LL/SC never provides old value
- The witness not available **after** the CAS
  - Overwritten by another thread
- JDK API mistake: witness turned into a boolean
  - Hence failure-for-cause can not be distinguished from spurious-failure
- Hence must spin on CAS failure until see reason for failure
  - Report either CAS success OR
  - CAS failure-for-cause
- Spinning builds a “No spurious failure CAS”

Towards A Scalable Lock-Free Coding Style

- Big Array to hold Data
- Parallel, Scalable read access
- Concurrent writes via: CAS & Finite State Machine
  - No JMM issues during Finite State Machine updates
  - No **lock**s, no **volatile**
- Fast as a best-of-breed not-thread-safe implementation
  - But as correct as thread-safe implementations
  - *Much* faster than locking under heavy load
  - No indirections in common case
  - Directly reach main data array in 1 step
- Resize as needed
  - Copy Array to a larger Array on demand
  - Use State Machine to help copy
  - “Mark” old Array words to avoid missing late updates

Agenda

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

Example 1: BitVector

- Size: O(max element)
  - Auto-resizing
- Supports concurrent insert, remove, test&set
- Obvious implementation:
  - Array of 'long' - 64-bit payload words
  - Bit mask & shift accessors
- How to 'mark' payload?
  - Steal 1 bit out of 64
  - MOD 63 to select index words – this example only
    - (Actually: avoid slow MOD by moving every 64^{th} 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;

  int old = A[idx];
  if( old < 0 )
    return copy(x).get(x);
  long mask = 1L <<(x%63);
  return (old & mask)!=0;
}
```

```
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);
    long mask = 1L <<(x%63);
    if( (old & mask) != 0)
      return true;
    if( CAS(A[idx],old,old|mask))
      return false;
}
```

Example 1: BitVector

- Read Array once – it may change out from under us!

```
boolean get( int x ) {
  long[] A = _A;
  int idx = x/63;
  if( idx >= A.length)
    return false;

  int old = A[idx];
  if( old < 0 )
    return copy(x).get(x);
  long mask = 1L <<(x%63);
  return (old & mask)!=0;
}
```

```
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);
    long mask = 1L <<(x%63);
    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;

  int old = A[idx];
  if( old < 0 )
    return copy(x).get(x);
  long mask = 1L <<(x%63);
  return (old & mask)!=0;
}
```

```
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);
    long mask = 1L <<(x%63);
    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;

  int old = A[idx];
  if( old < 0 )
    return copy(x).get(x);
  long mask = 1L <<(x%63);
  return (old & mask)!=0;
}
```

```
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);
    long mask = 1L <<(x%63);
    if( (old & mask) != 0)
      return true;
    if( CAS(A[idx],old,old|mask))
      return false;
}
```

Example 1: BitVector

- Failed CAS must retry – BUT!
  - Means another thread made progress

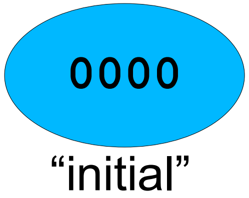
```
boolean get( int x ) {
  long[] A = _A;
  int idx = x/63;
  if( idx >= A.length)
    return false;

  int old = A[idx];
  if( old < 0 )
    return copy(x).get(x);
  long mask = 1L <<(x%63);
  return (old & mask)!=0;
}
```

```
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);
    long mask = 1L <<(x%63);
    if( (old & mask) != 0)
      return true;
    if( CAS(A[idx],old,old|mask))
      return false;
  }
```

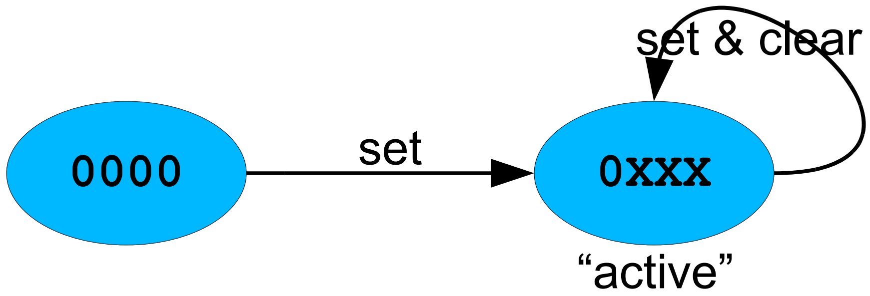
Example 1: BitVector

- 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: Sign-bit to stop further updates
  - Use CAS to set sign-bit
  - Then copy word to new array
  - Repeat operation on new array
- Finite State Machine!
  - per Array word
  - Hidden in the code
- Let's make the FSM obvious...

BitVector State Machine

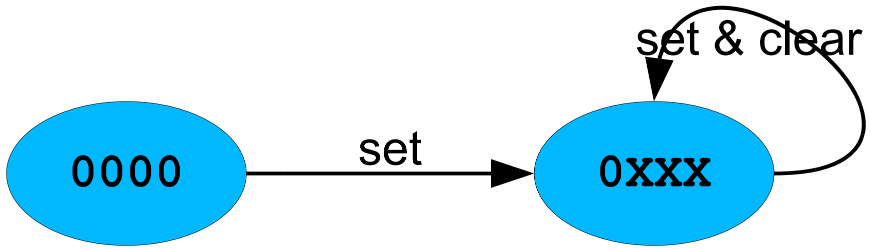


BitVector State Machine



A: Normal operations

BitVector State Machine

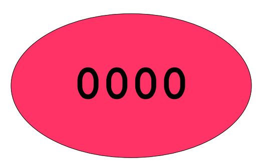


A: Normal operations

Out-of-Bounds set
triggers resize!

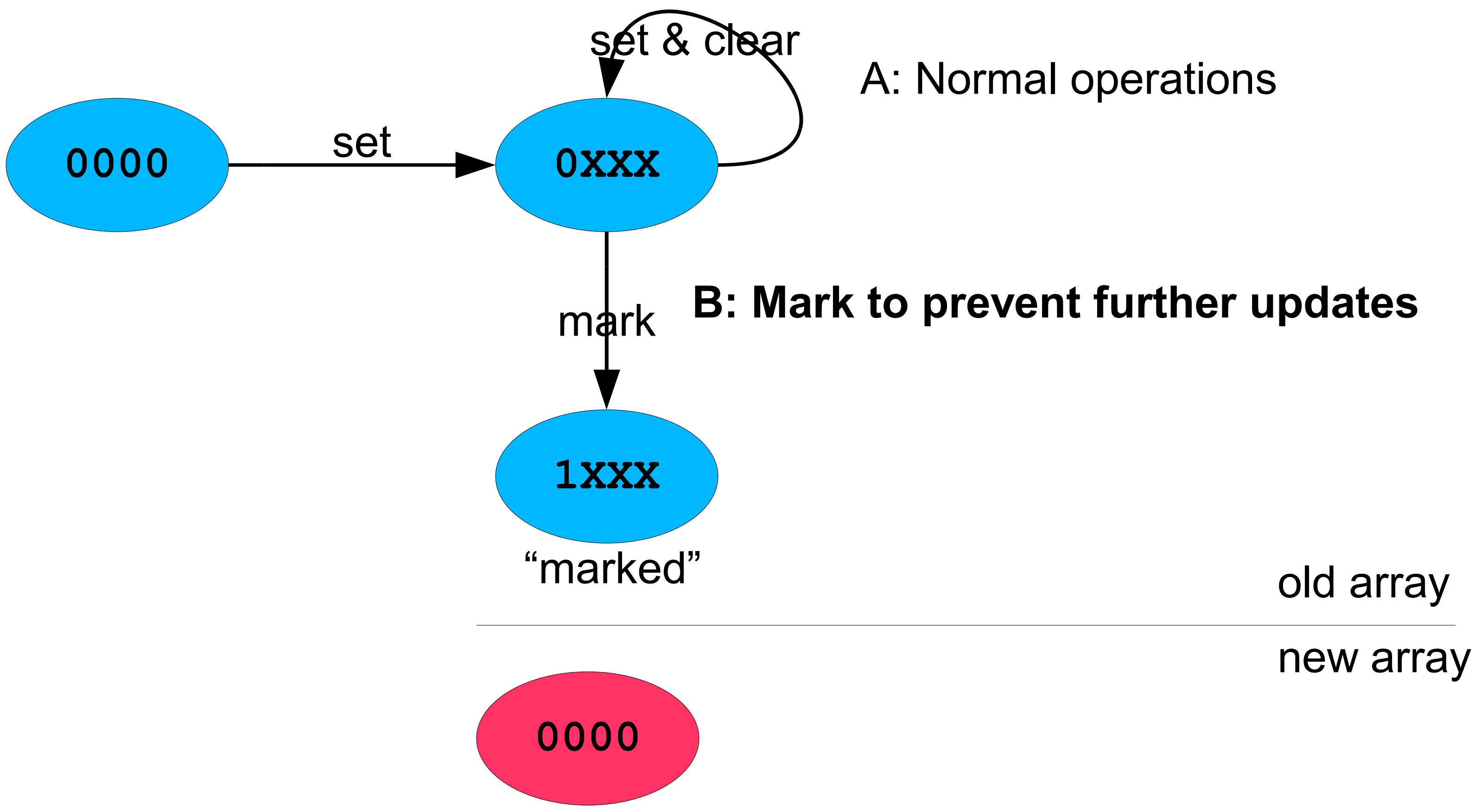
old array

new array

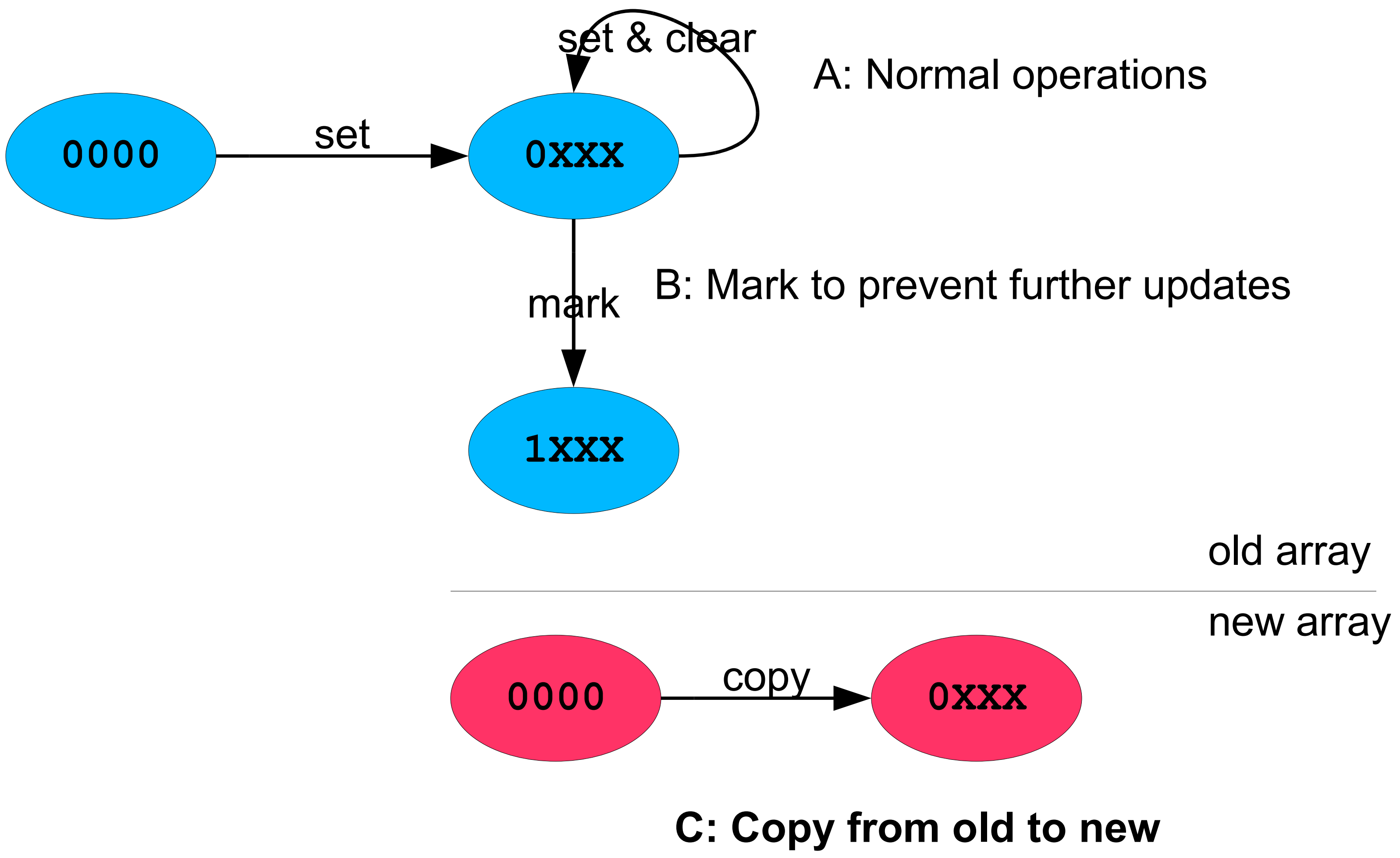


“initial”

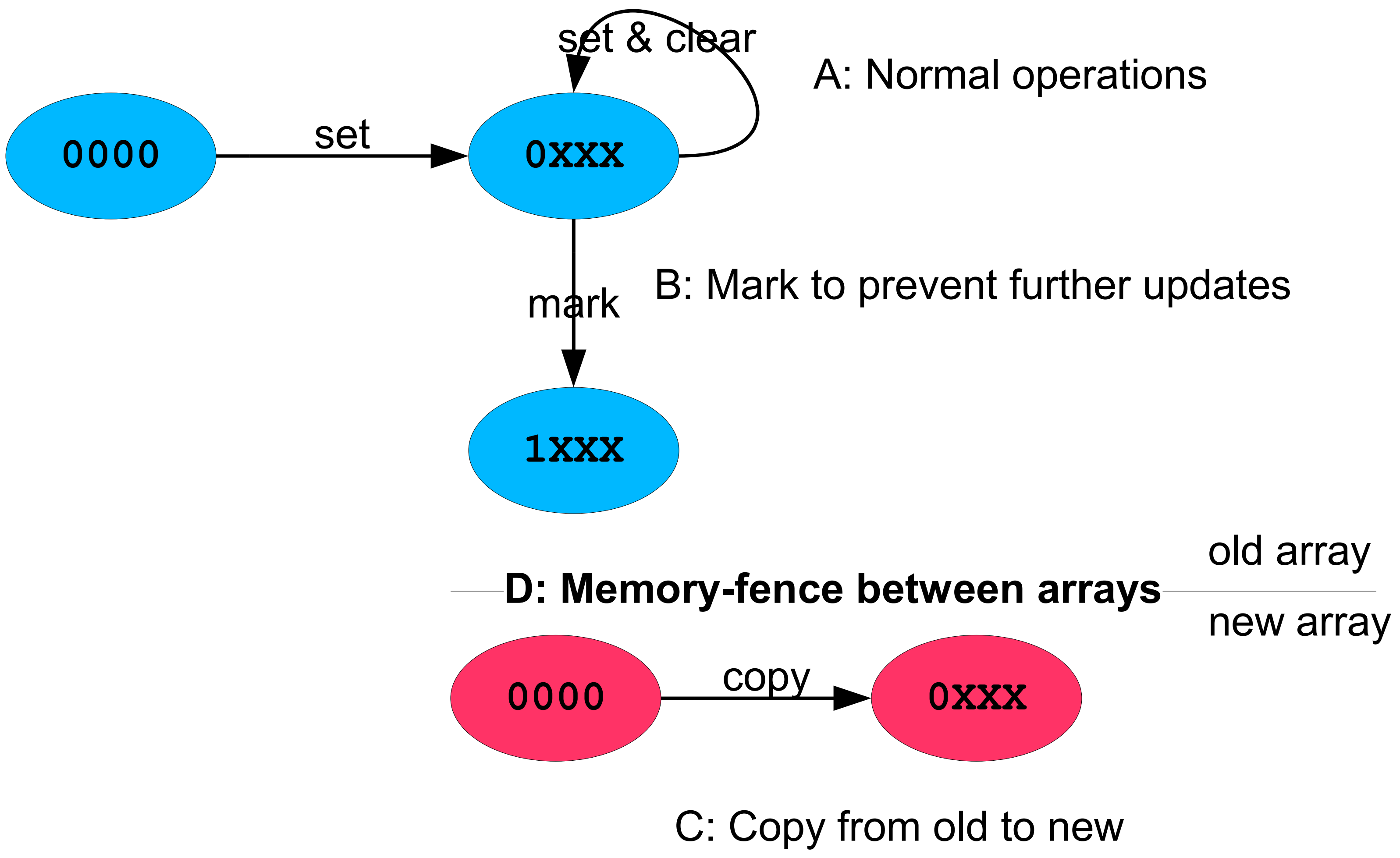
BitVector State Machine



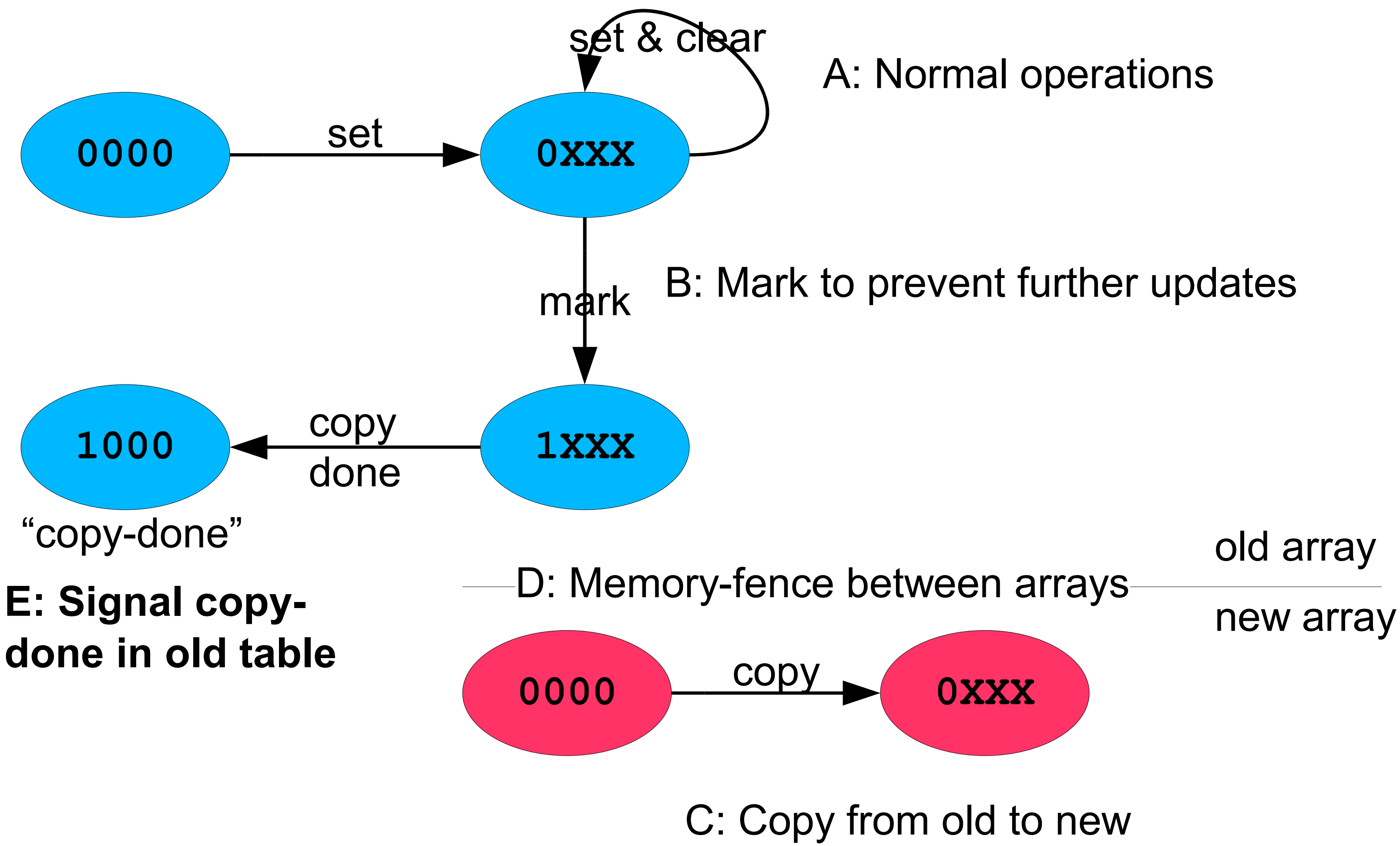
BitVector State Machine



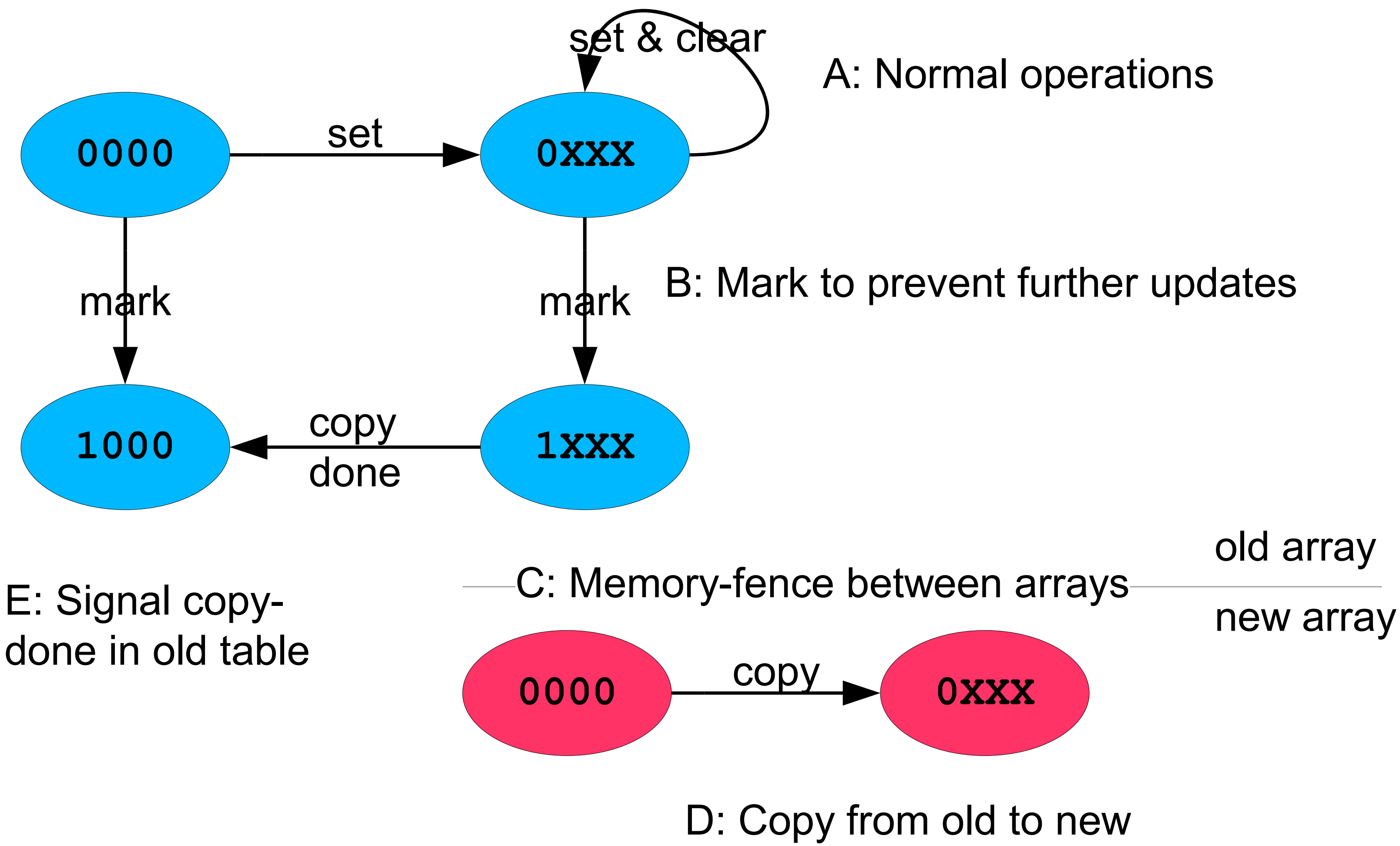
BitVector State Machine



BitVector State Machine



BitVector State Machine



Resize - motivation

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

Resize – Copy Mechanics

- Helper: any thread copying words it does not directly need
- Helpers CAS-up a "promise to copy" counter
  - Atomic-increment by fixed N (e.g. 16 words)
- Helpers **copy** words via State Machine
- Helpers atomic-increment "done work" counter
  - On transition to "copy-done" state
- Promote new Array when "done work" == A.length
- What If: Helper stalled? (promises but never copies)
  - Allow helpers to "double-promise"!
  - Worst case: each thread can complete entire copy
- Eventually, copy completes & array promotes

Coding Style Elements

- Large array for parallel read & update
  - No JMM issues for read or update (no lock, no volatile)
- State Machine per-array-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
  - Copy is parallel, incremental, concurrent
  - Copy part of State Machine
  - Unrelated threads can make progress during resize
  - Fence between old and new tables

Agenda

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

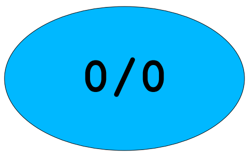
Example 2: HashTable

- Array of K/V Pairs
  - Keys in even slots, Values odd slots
  - CAS each word separately, but FSM spans both words
  - Value can also 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
  - More than billion reads/sec simultaneous with
  - More than 10million updates/sec
- Code up in SourceForge, high-scale-lib
  - Passes Java Compatibility Kit (JCK) for ConcurrentHashMap

"Uninteresting" Details

- Good, standard engineering – nothing special
- 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

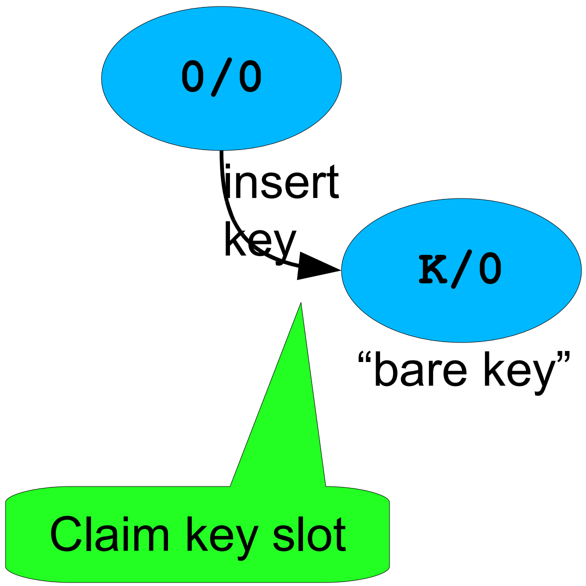
HashTable State Machine



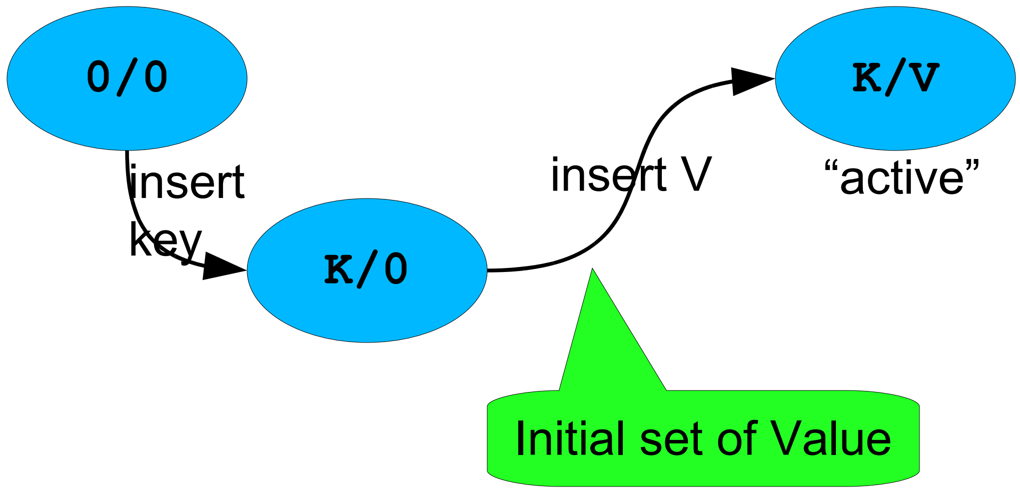
“initial”

- Inserting K/V pair
- Already probed table, missed
- Found proper empty K/V slot
- Ready to claim slot for this Key

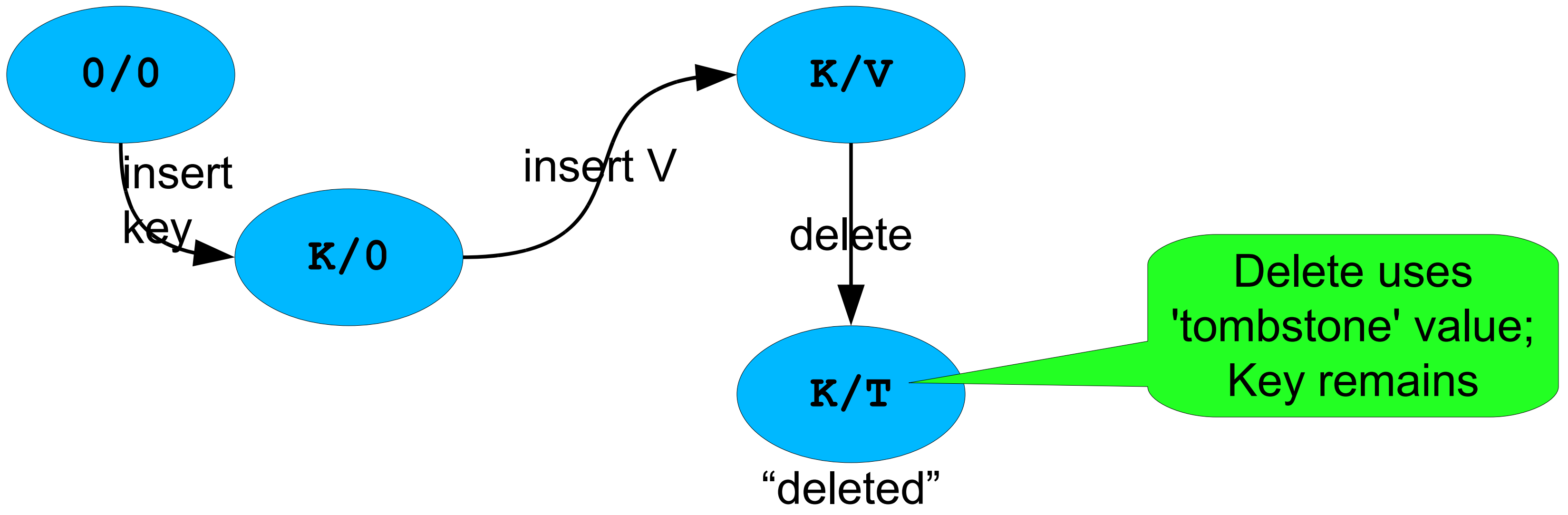
HashTable State Machine



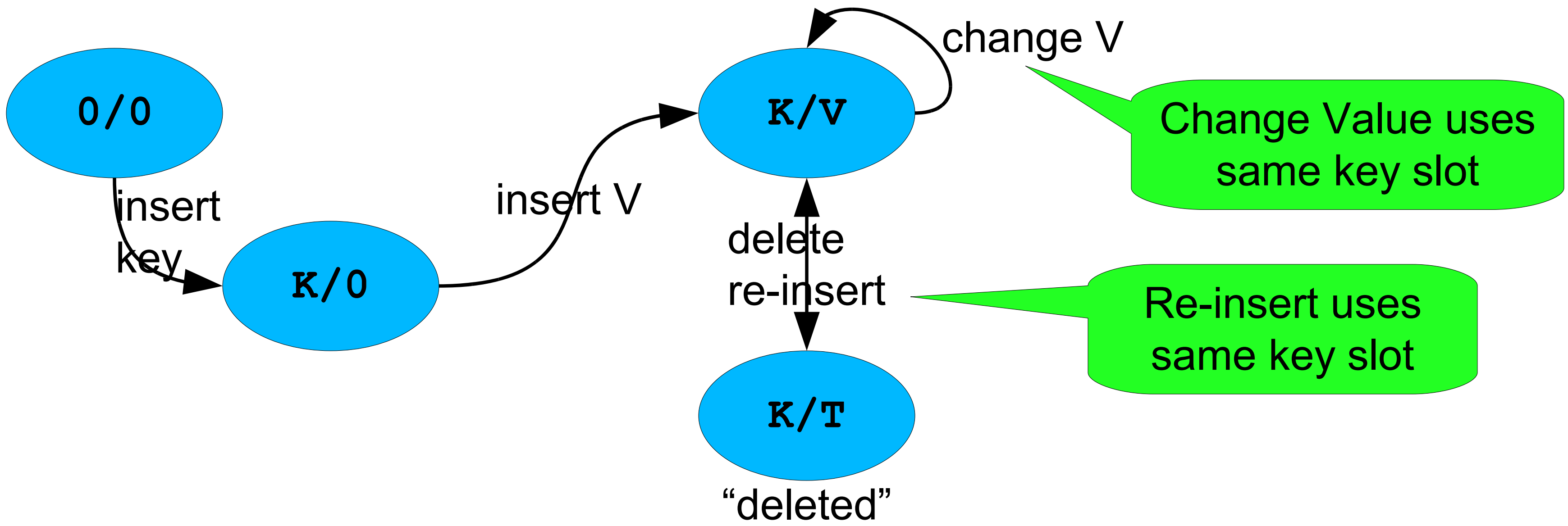
HashTable State Machine



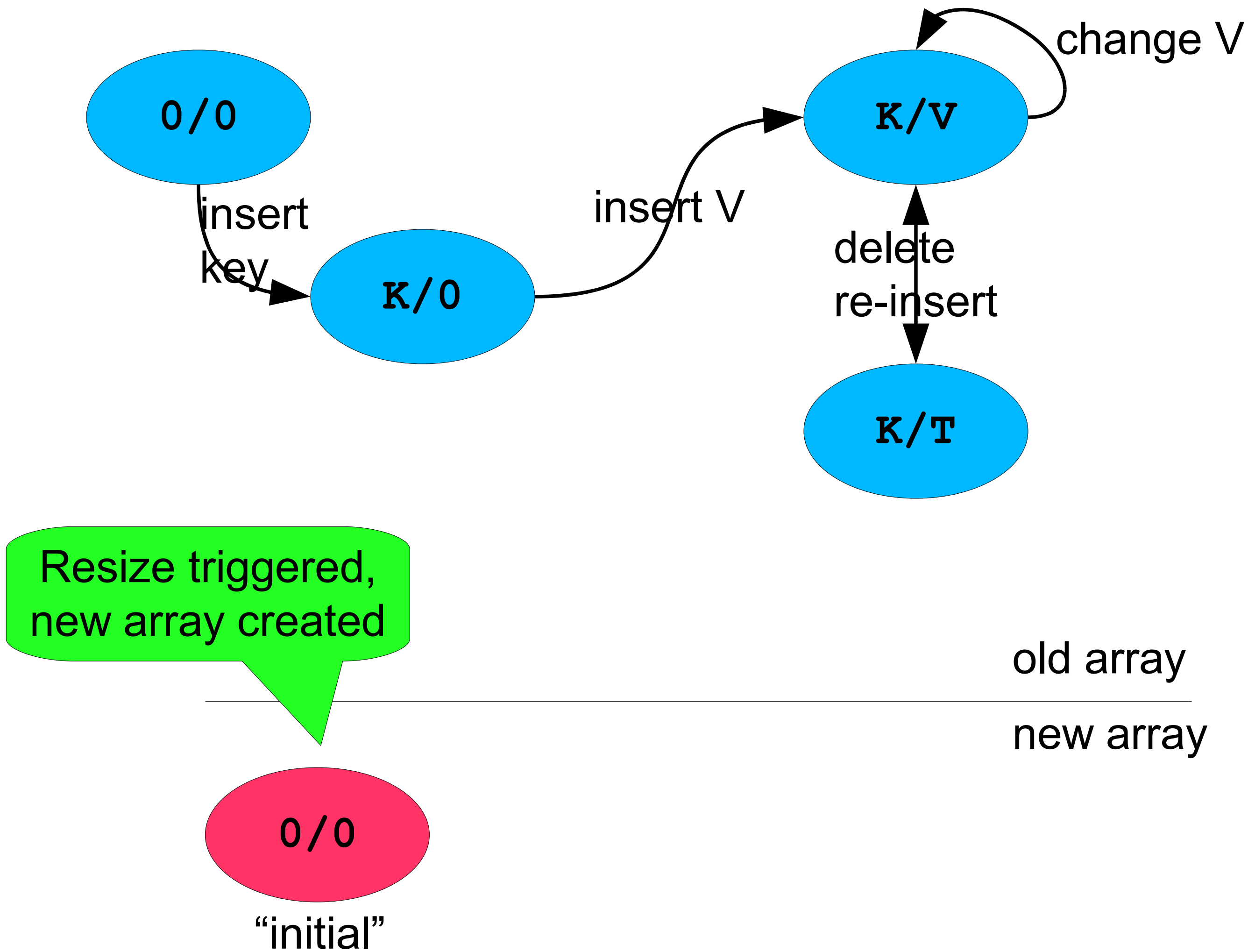
HashTable State Machine



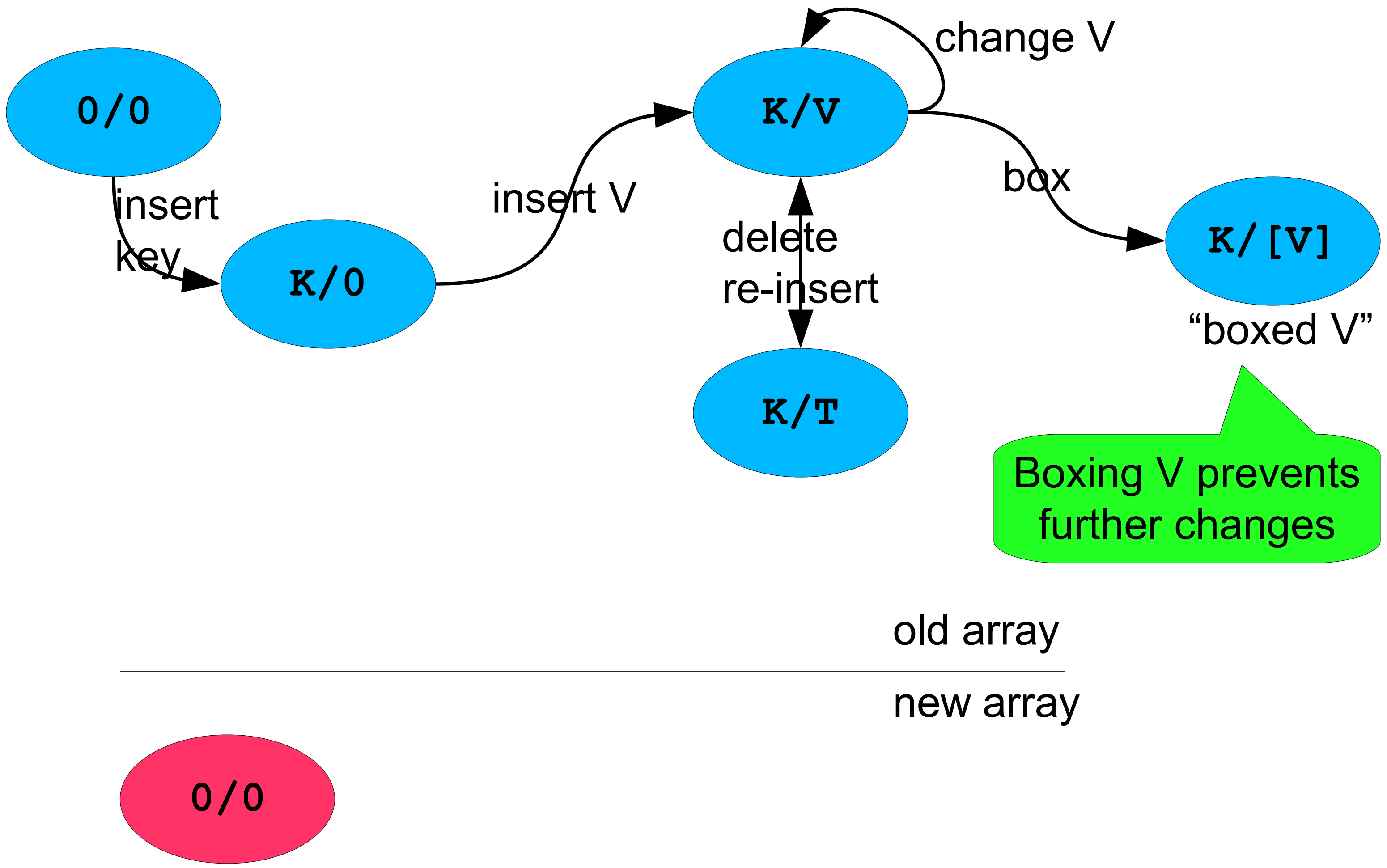
HashTable State Machine



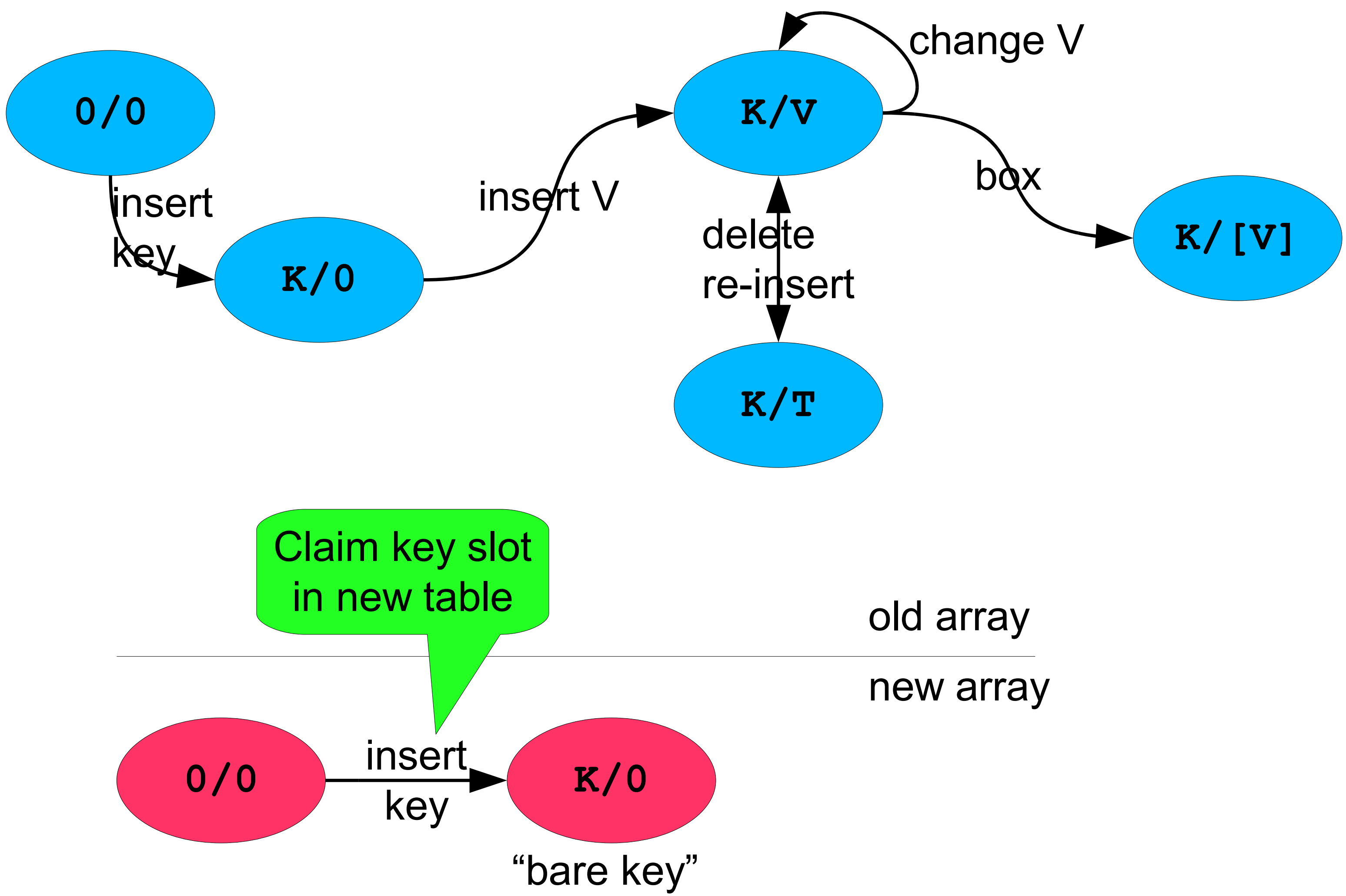
HashTable State Machine



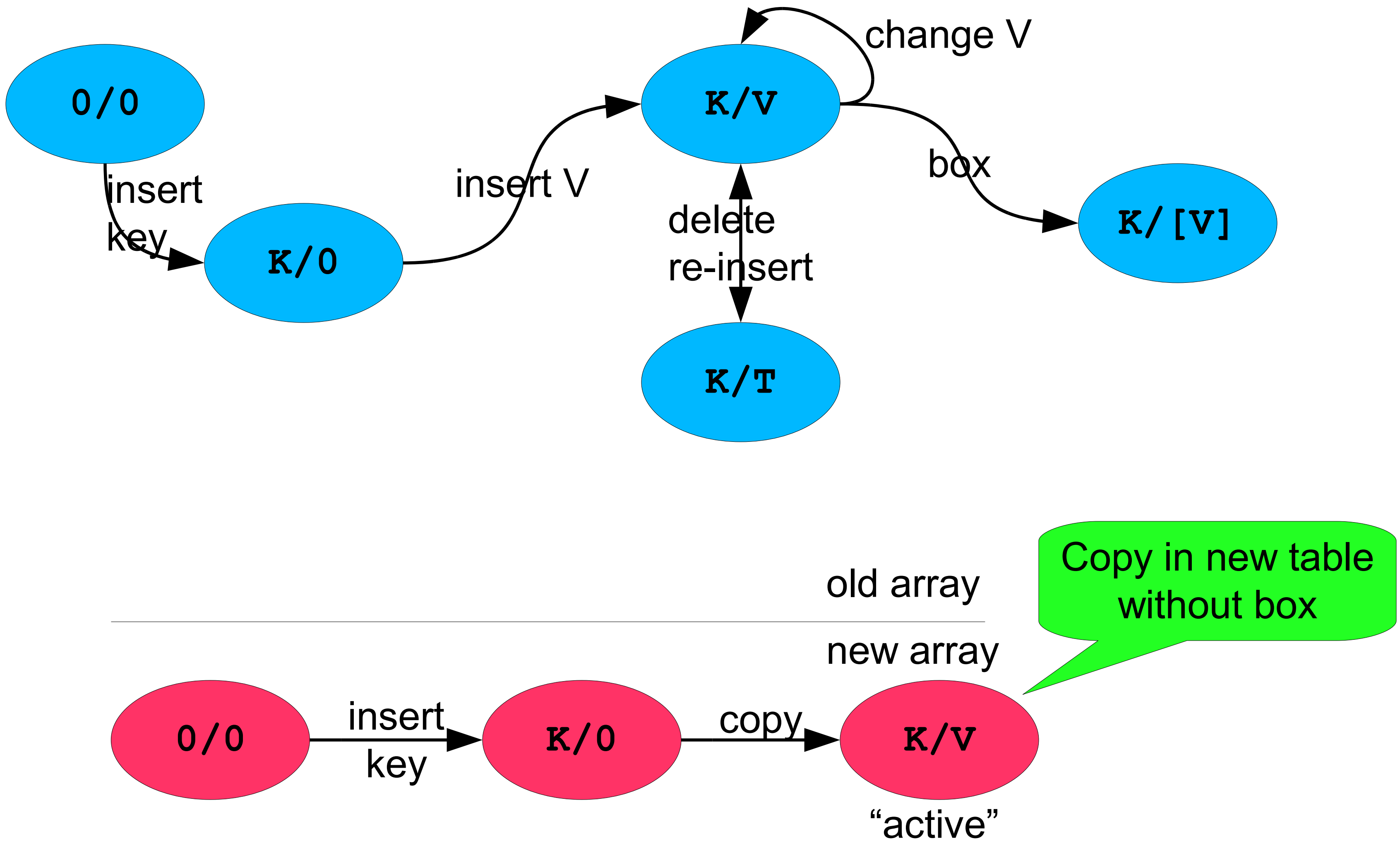
HashTable State Machine



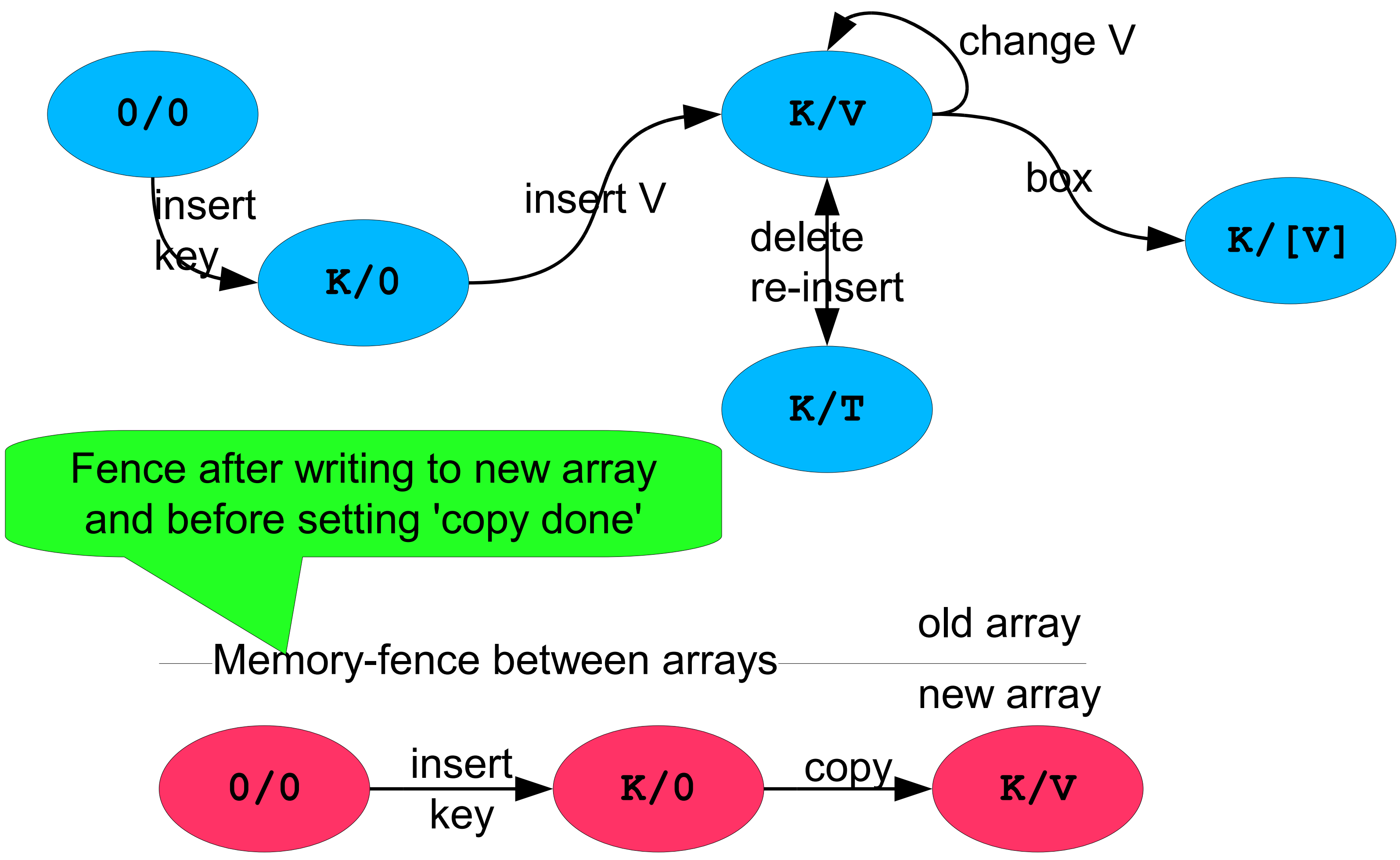
HashTable State Machine



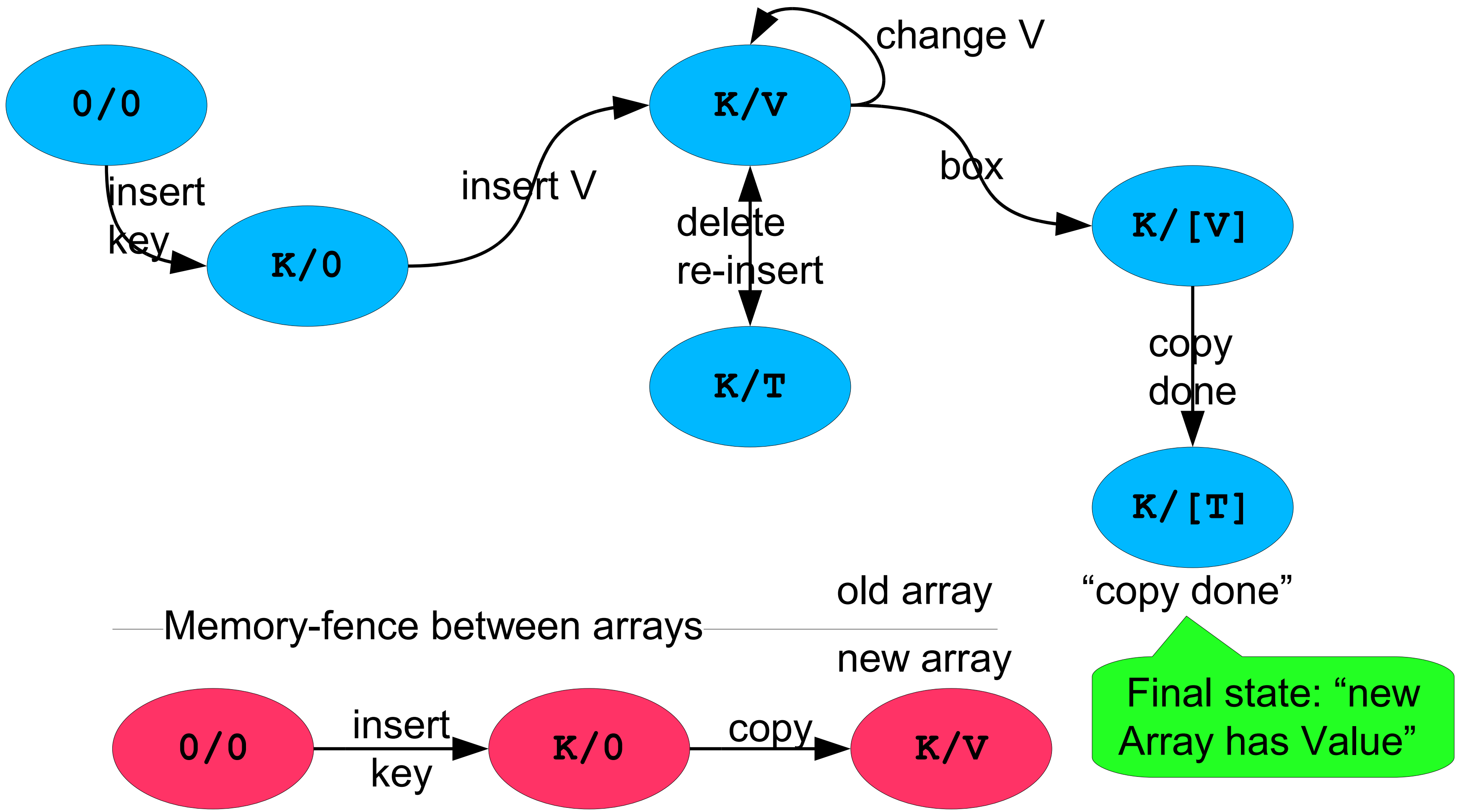
HashTable State Machine



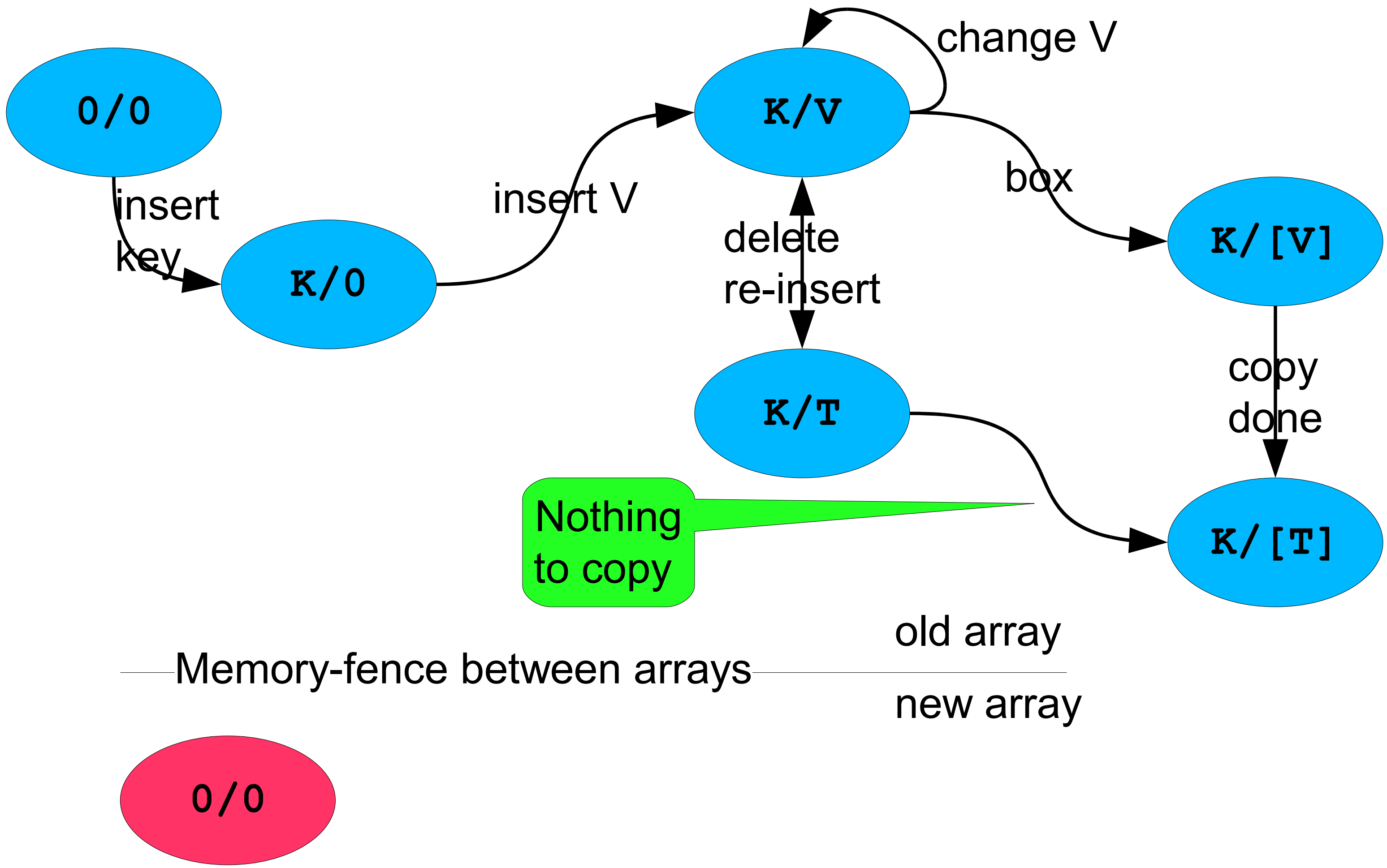
HashTable State Machine



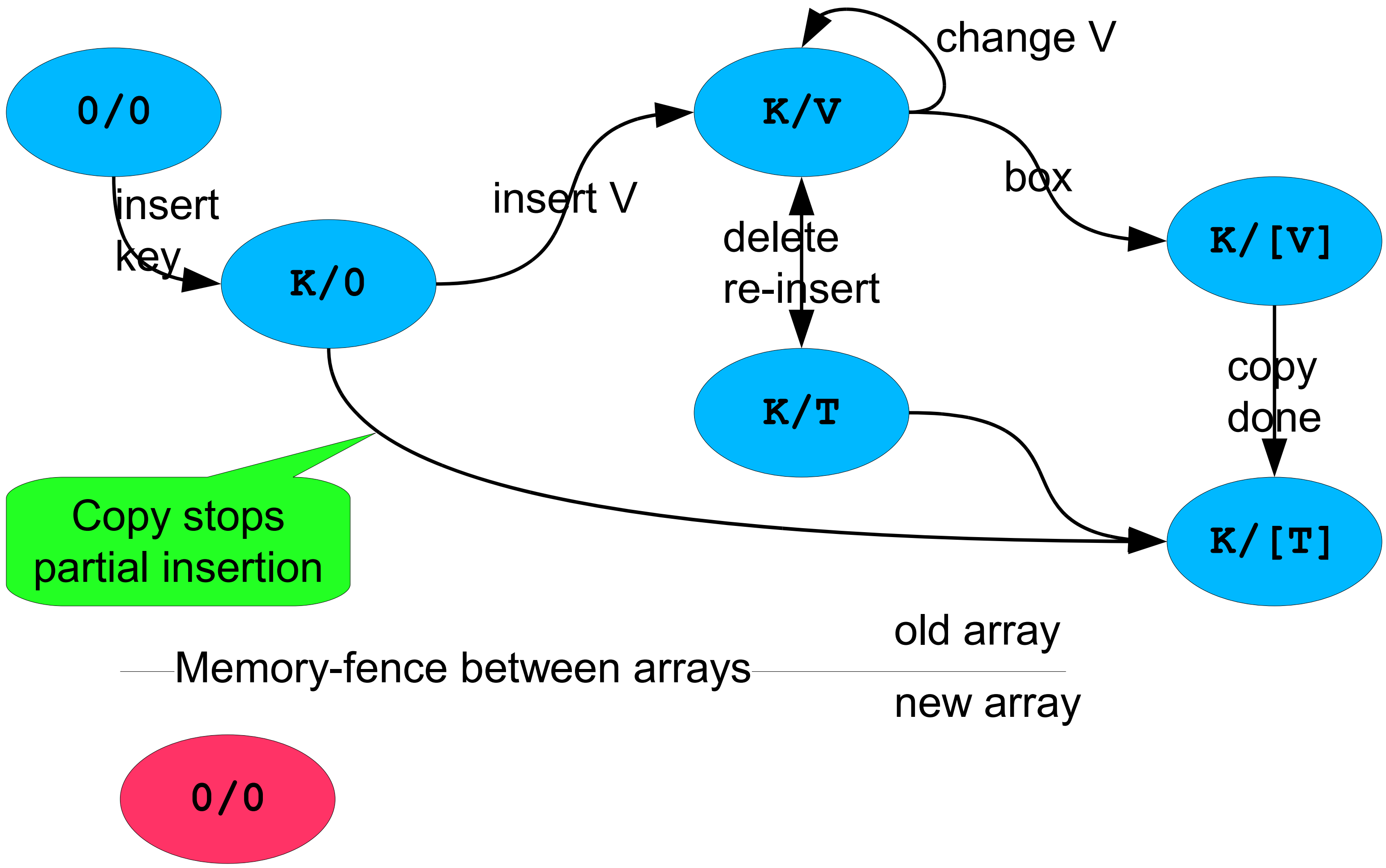
HashTable State Machine



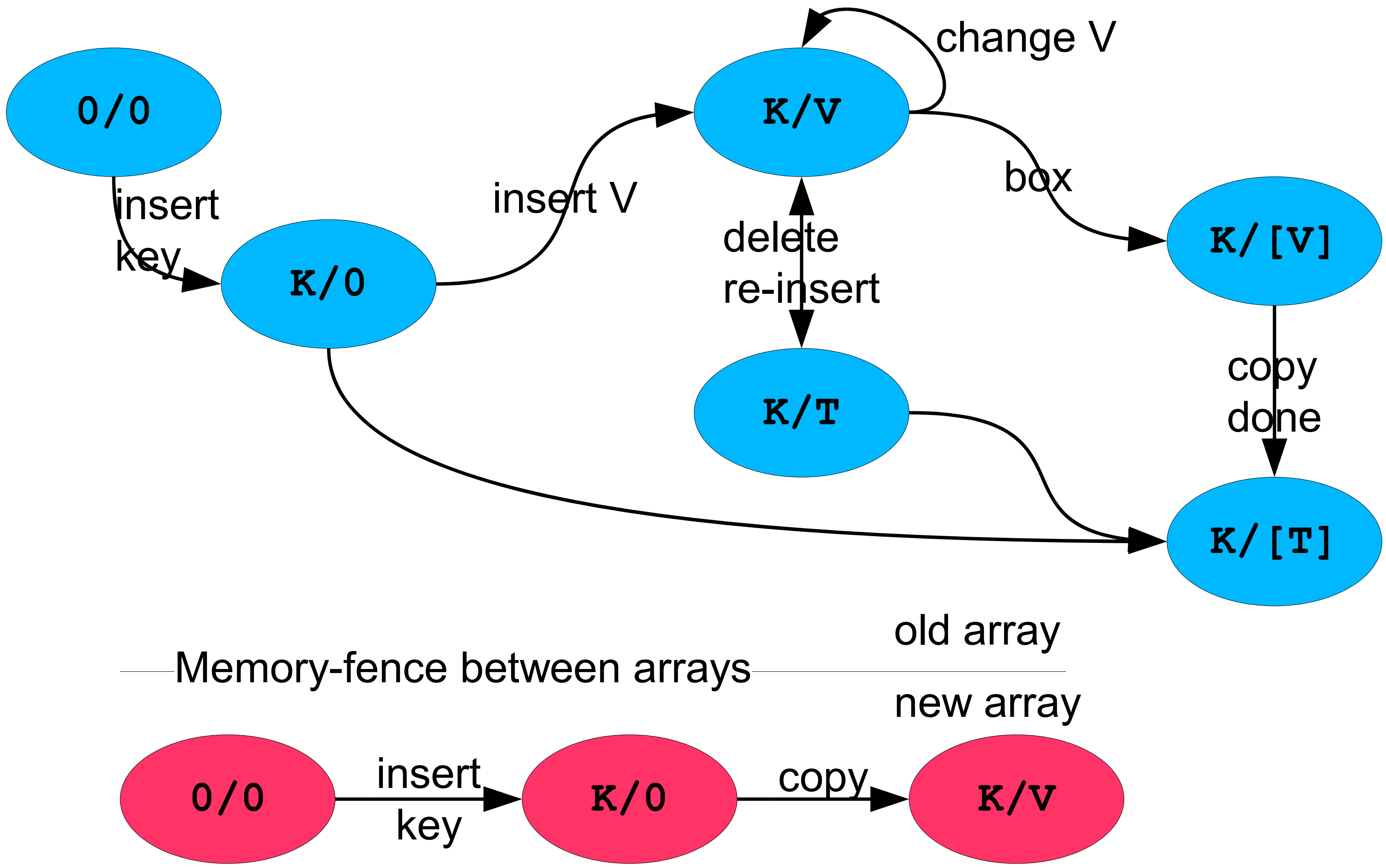
HashTable State Machine



HashTable State Machine



HashTable State Machine



Agenda

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

Example 3: Nearly FIFO Queue

- 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
- Could stripe Queue:
  - Many short Queues
  - Select random Queue
  - Many different locks or many different words to CAS
    - Less contention
  - Pick at random to push or pop
  - Must search all queues for not-full or not-empty

Example 3: Nearly FIFO Queue

- 1000's of CPUs need 1000's of Queues
  - Stripe Ad-Absurdum
  - Queues get ever-smaller
  - Get down to Queues of 1 entry
- Single-entry Queue: either full or empty
  - Implement as a single word
  - Either **null** or value
- Need 1000's of single-entry Queues
  - 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
  - Or might neglect tasks left in other slots
  - Means every 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 mostly 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 is entire 'stoned
- Still need feedback mechanisms on P/C threadpools

Example 3: Nearly FIFO Queue

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

Agenda

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

Summary

- Lock-Free
- Highly scalable (proven scalable to ~1000 CPUs)
- Use large array for data
  - Allows fast parallel-read
  - Allows parallel, incremental, concurrent copy
- Use Finite 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

http://e2e.azulsystems.com

THANK YOU

Dr. Cliff Click, Distinguished Engineer

Azul Systems

http://blogs.azulsystems.com/cliff