Concurrency and High Performance Reloaded

Disclaimer

Any performance tuning advice provided in this presentation.....

will be wrong!

Me

- Work as independent (a.k.a. freelancer)
  - performance tuning services
  - benchmarking
  - Java performance tuning course
- www.javaperformancetuning.com
- www.theserverside.com
- Nominated Sun Java Champion
- Other stuff

single-threaded, single-core

how did we get better performance?

concurrent programming is the norm


CPU Usage
CPU Usage History
PF Usage
Page File Usage History

multi-core is a fact of life!

we need to “deliver twice as much concurrency every 18 mounts”

hardware components are notsharable

access to shared data must be serialized

databases offer access to shared data

serialization limits scalability

- Maths to explain relationship between serialized execution and processor utilization

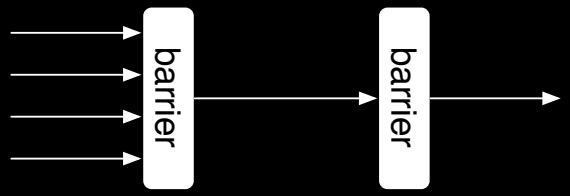
$$\frac{1}{F + \frac{(1 - F)}{N}}$$

  - F -> 0 number of utilized CPU -> N
  - F -> 1 number of utilized CPU -> 1

Amdahl's Law

serialization limits throughput

- Maths explaining the relationship between locking and throughput



$\lambda = 1 / \mu$

$\mu = 10ms, \lambda = 100$ tps

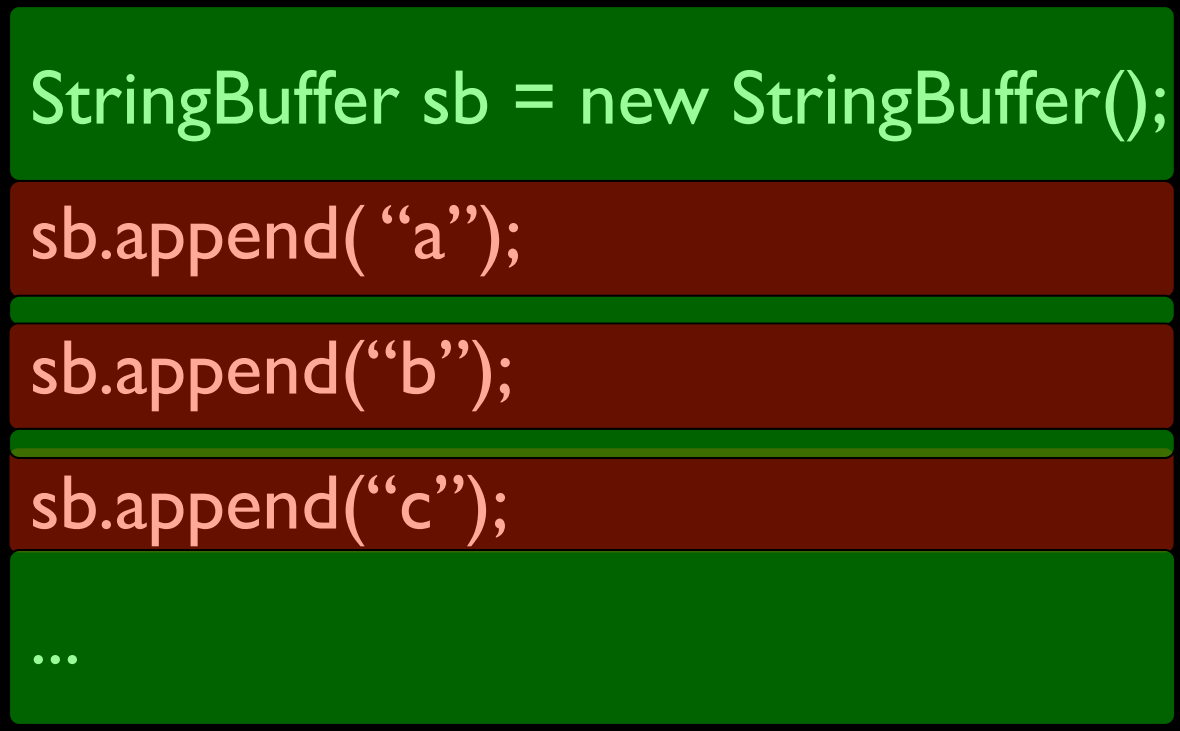
$\mu = 100ms, \lambda = 10$ tps

Little's Law

locking is pessimistic

getting better concurrency in the JVM

Java and system level locks



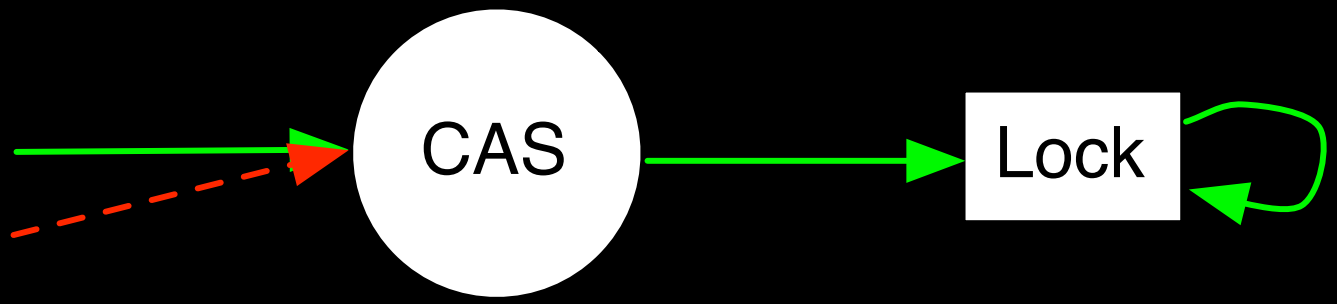
Lock Coarsening

```
StringBuffer sb = new StringBuffer();
sb.append( “a”);
sb.append(“b”);
sb.append(“c”);
...
```

Lock Coarsening

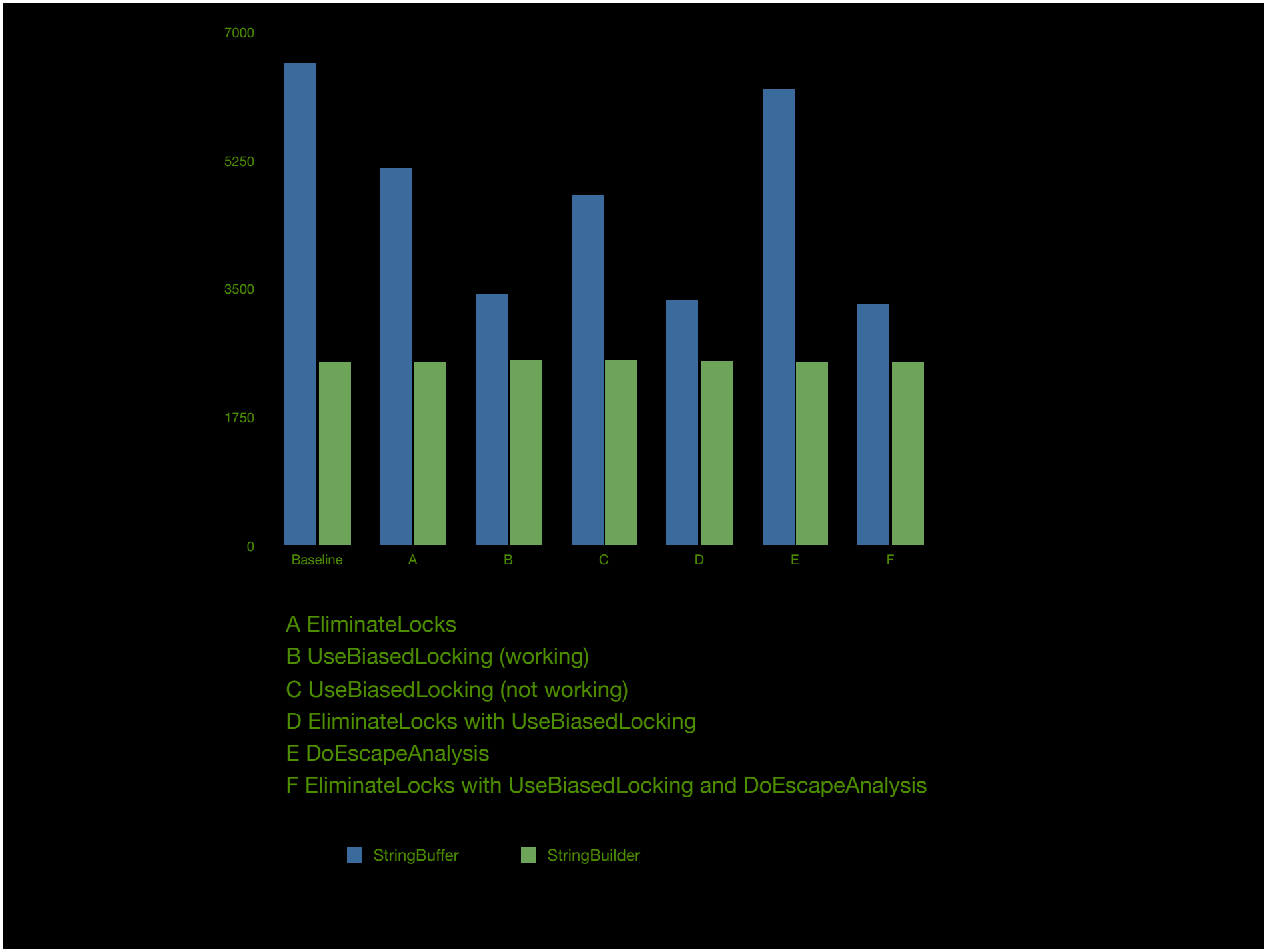
```
{
    StringBuffer sb = new StringBuffer();
    sb.append( “a”);
    sb.append(“b”);
    sb.append(“c”);
}
```

Lock Elision



Biased Locking

do these optimizations work?

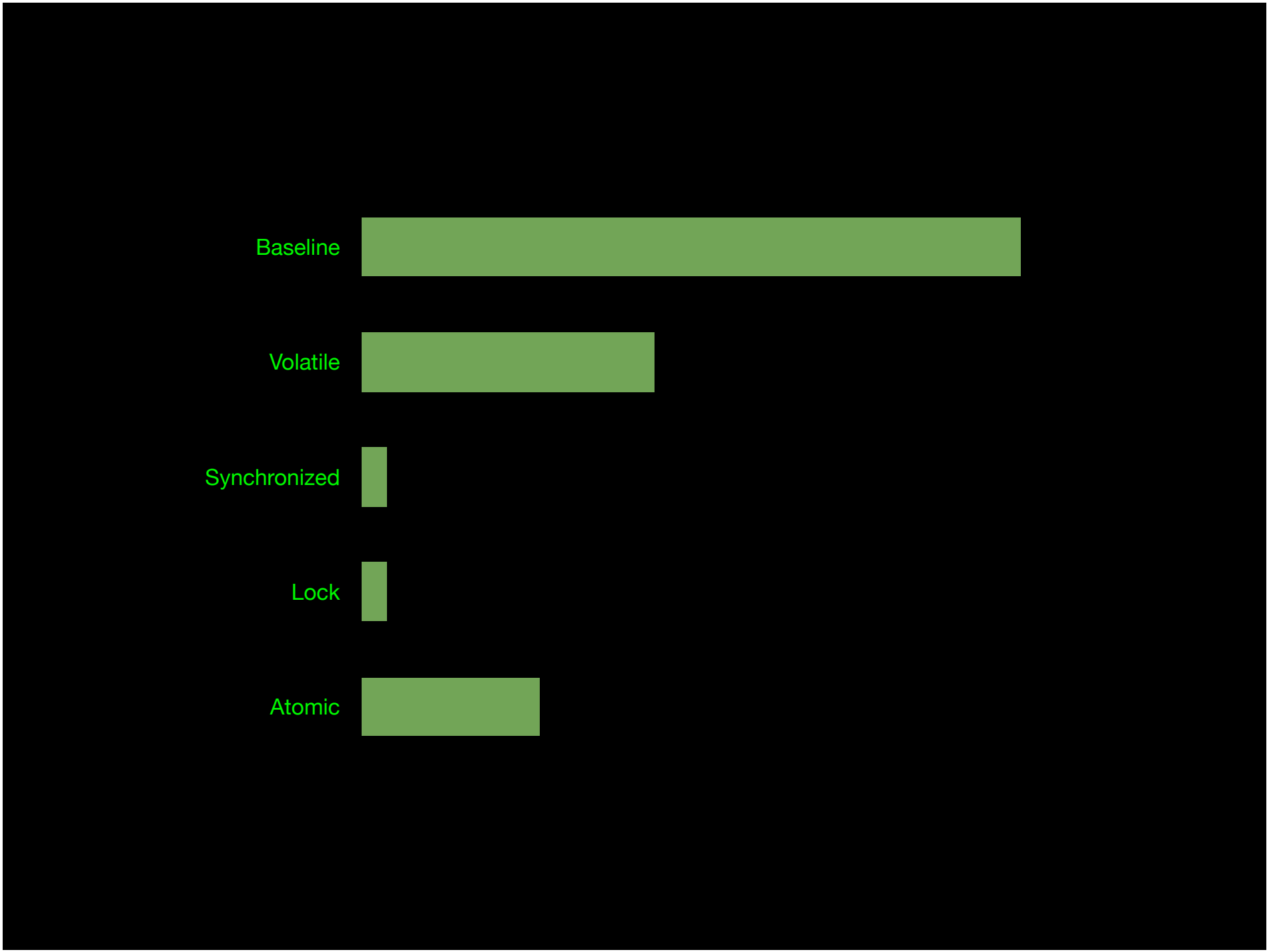

7000
5250
3500
1750
0
Baseline
A
B
C
D
E
F
A EliminateLocks
B UseBiasedLocking (working)
C UseBiasedLocking (not working)
D EliminateLocks with UseBiasedLocking
E DoEscapeAnalysis
F EliminateLocks with UseBiasedLocking and DoEscapeAnalysis
StringBuffer
StringBuilder

techniques we can use

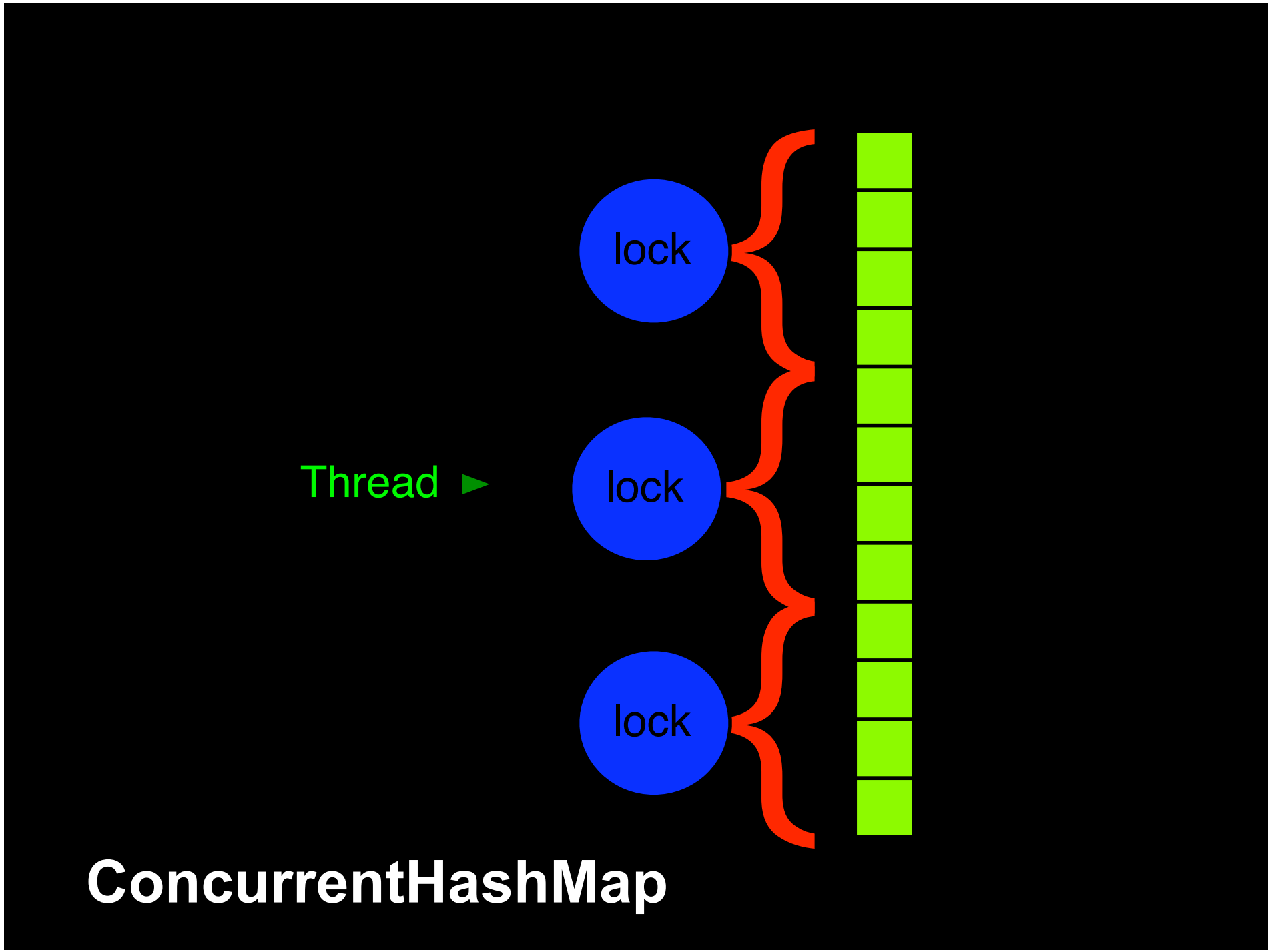
Atomics to reduce lock contention

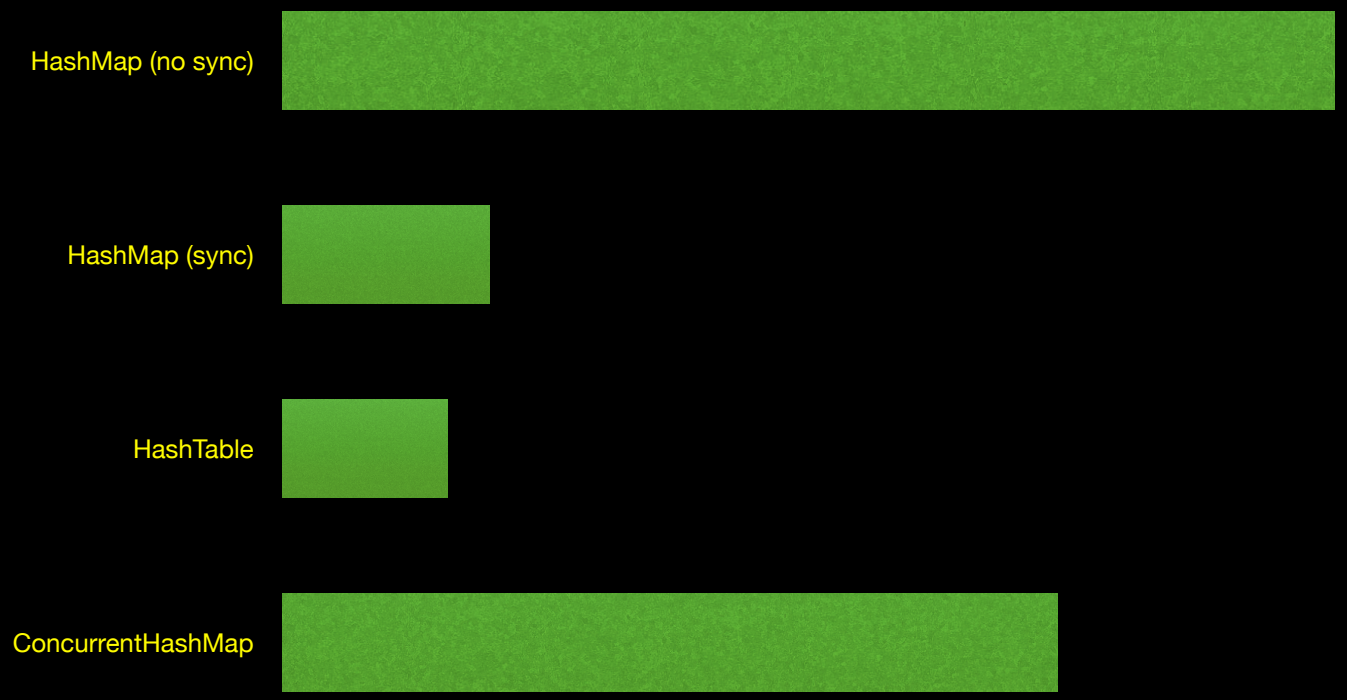
```
private int counter = 0;

Runnable mutator = new Runnable() {
    public void run() {
        long localCount = 0;
        while ( running) {
            counter++;
            counter--;
            localCount++;
        }
        addToTotalCount( localCount);
    }
};
```


Baseline
Volatile
Synchronized
Lock
Atomic

Lock striping

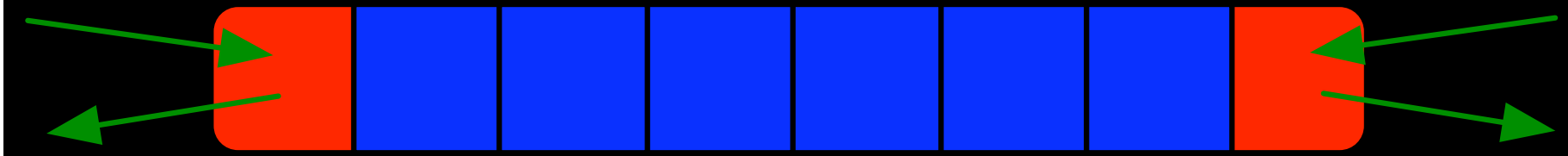

lock
Thread
lock
lock
ConcurrentHashMap


HashMap (no sync)
HashMap (sync)
HashTable
ConcurrentHashMap

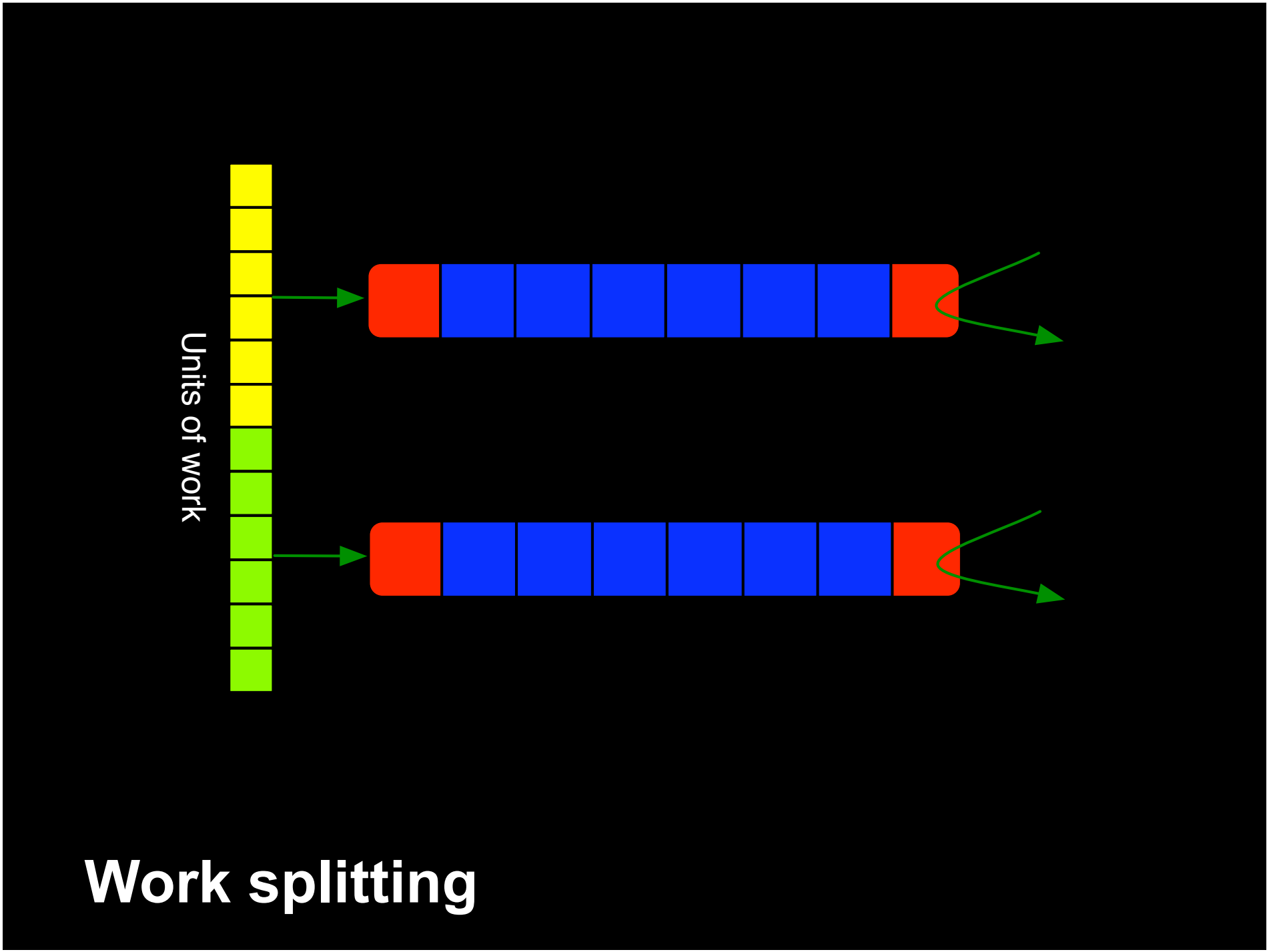
Blackboard

teaching threads to steal

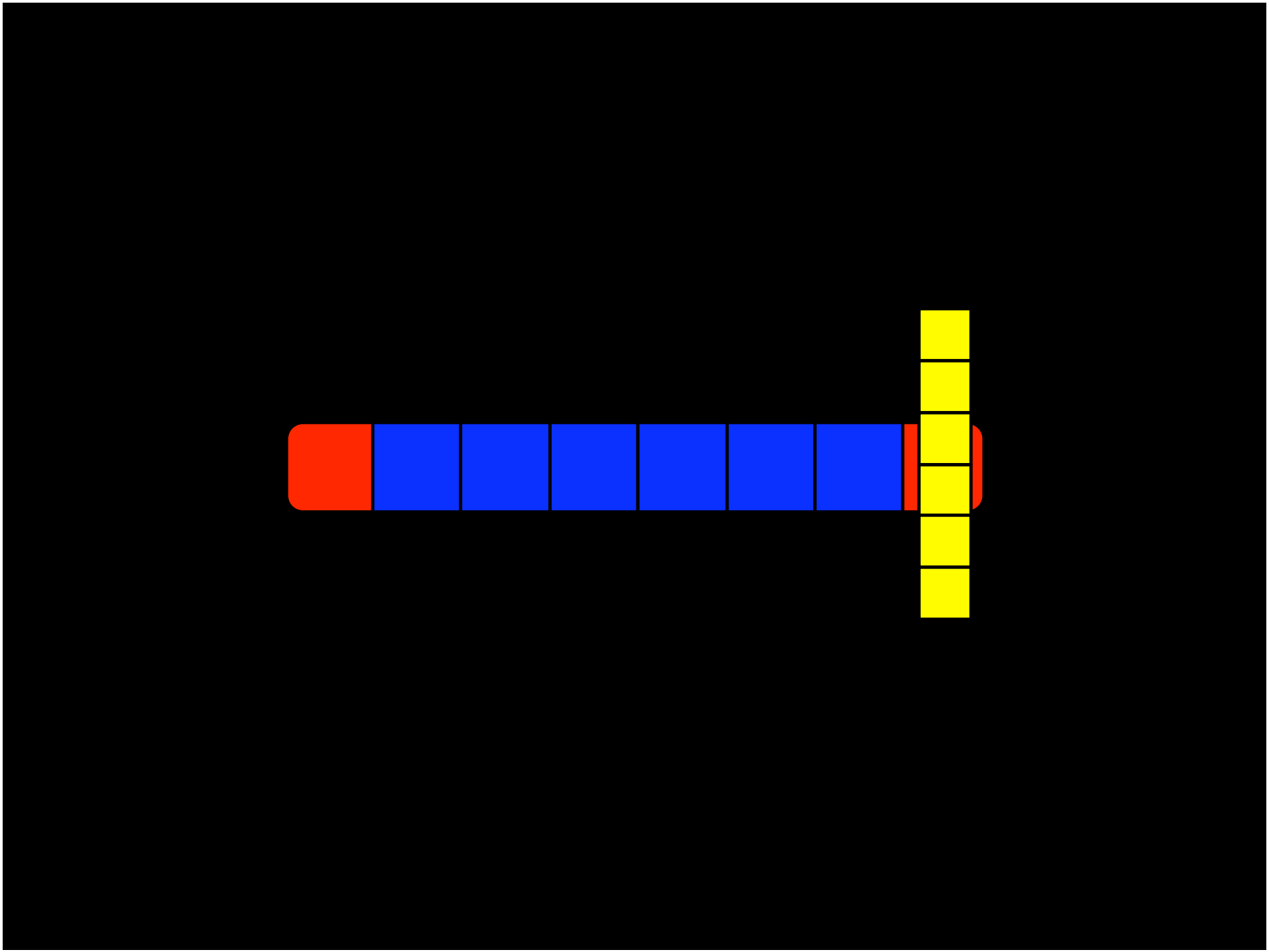
Fork-Join

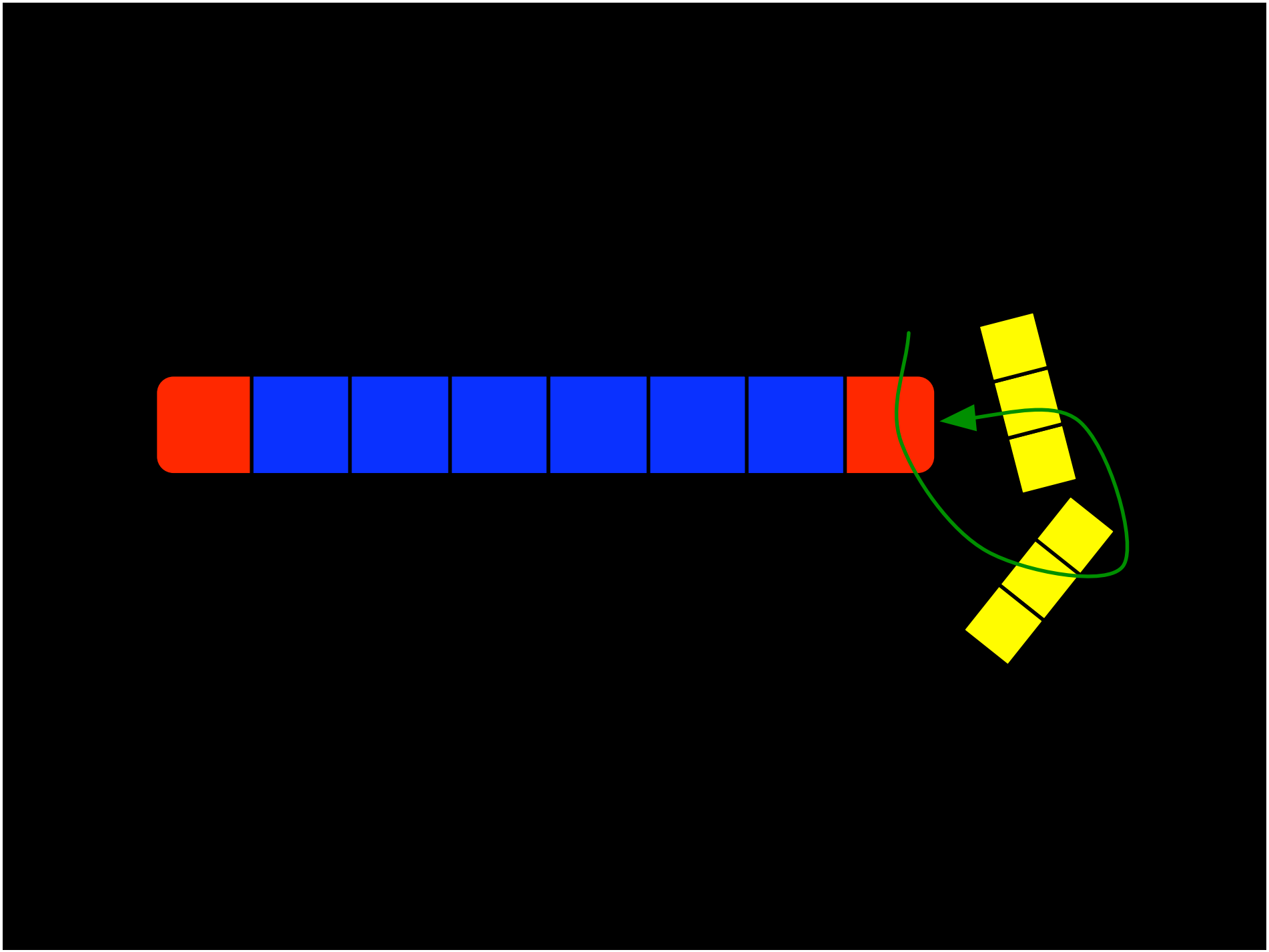


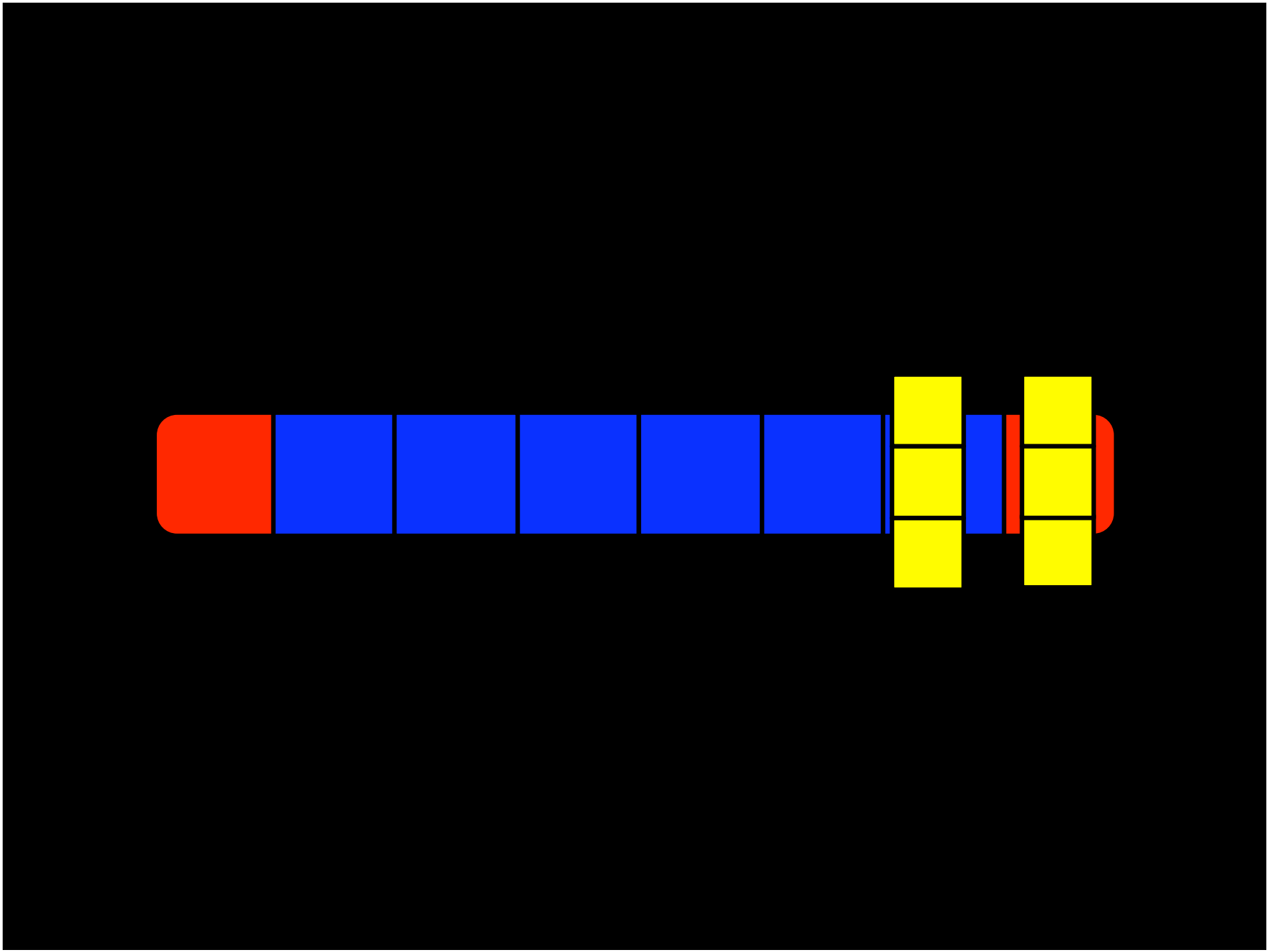
Dequeue

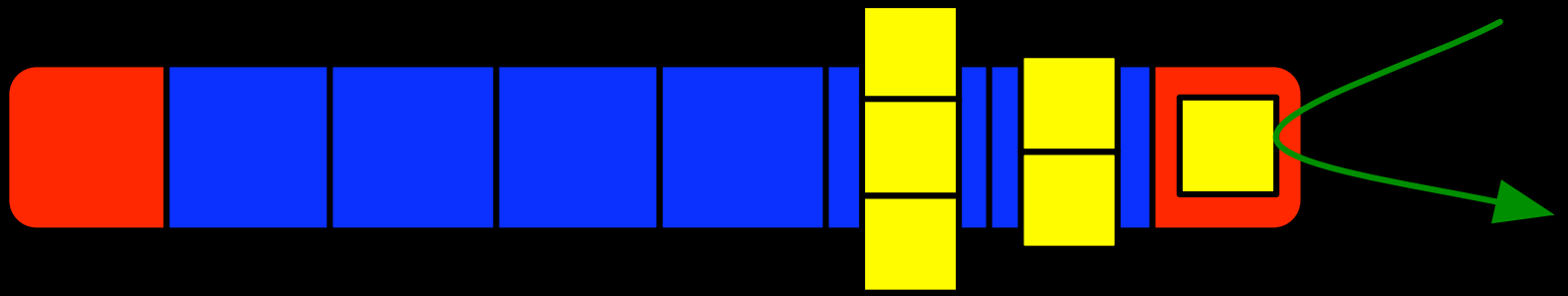


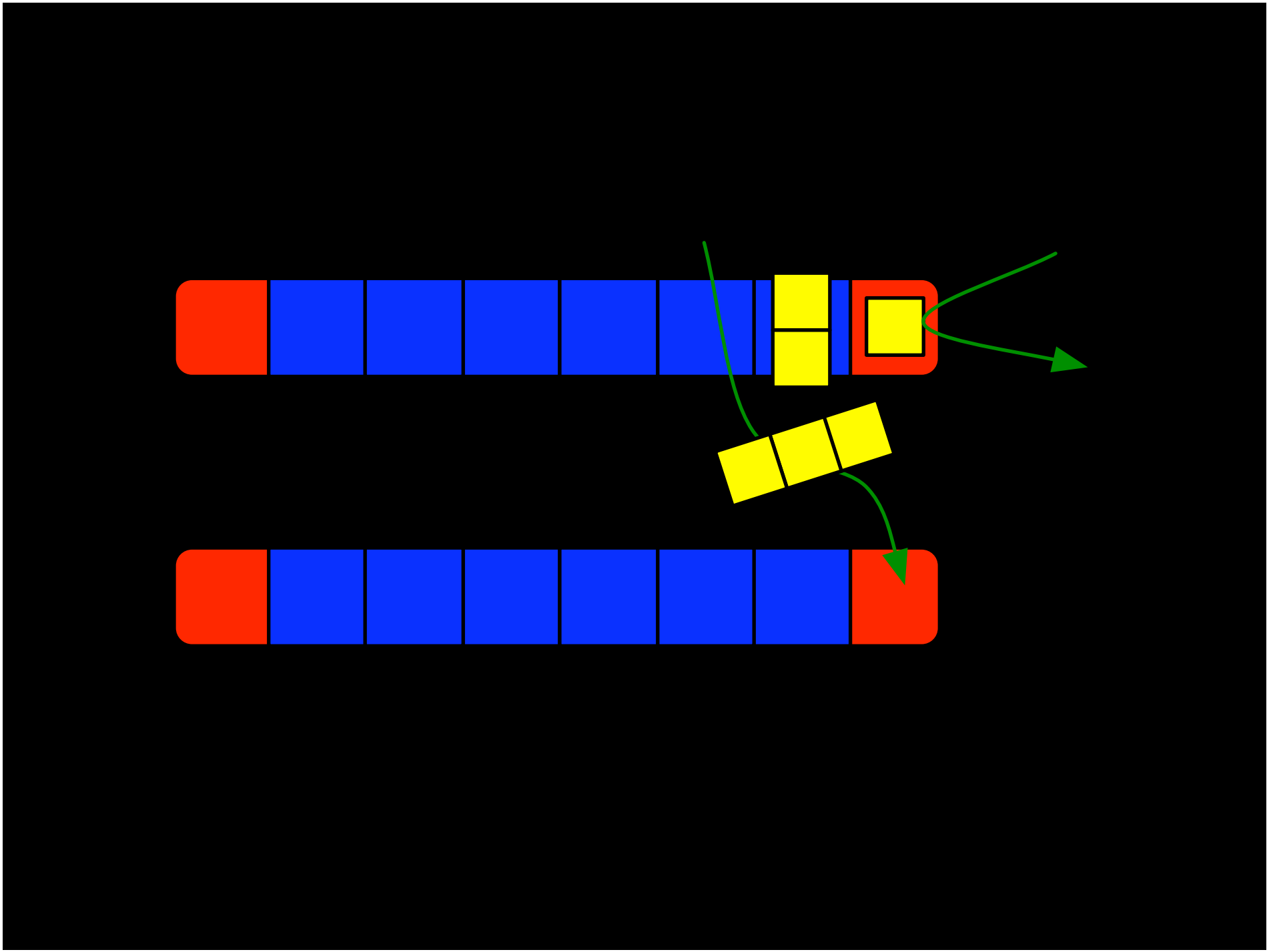
Work splitting











Degrees of Scalability

Lock free concurrency

Parallel reads, serialized writes

Reader/Writer lock with only readers
will not scale beyond 100 cpus

large arrays for concurrent

arrays to hold all data

resize cannot block

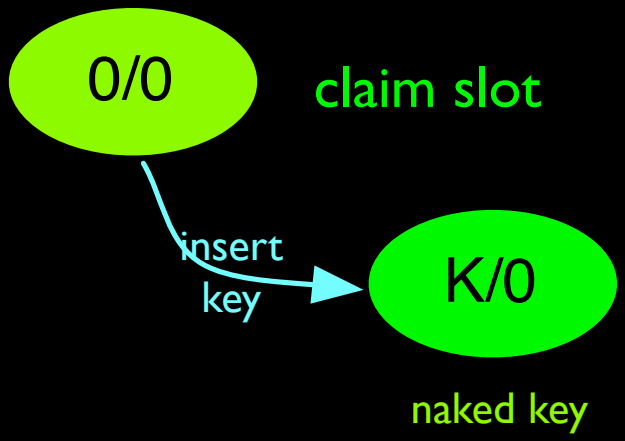
fully concurrent lock-less hashmap

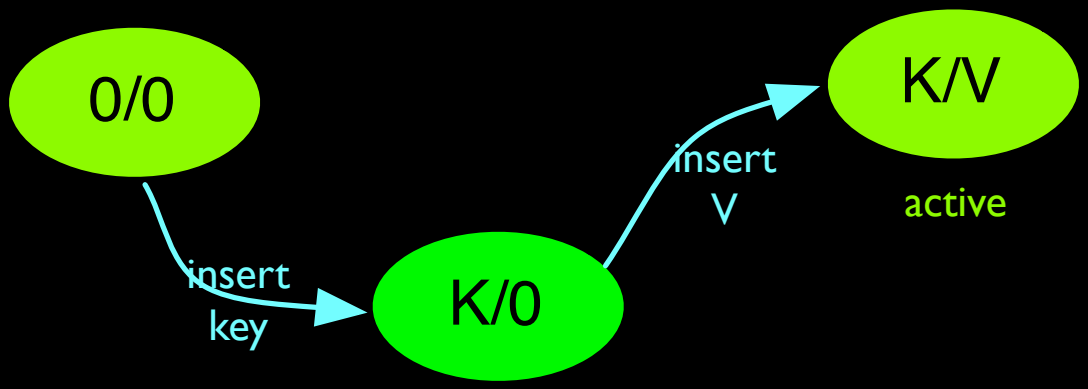
Things we need

- Large array to hold all data
  - alternating array of key value pairs
- state machine for pair of words
  - CAS to manage state transistion
- Tombstone to mark deleted words
- Box to mark a resize
  - allows read access but prevents update
- No single point of contention

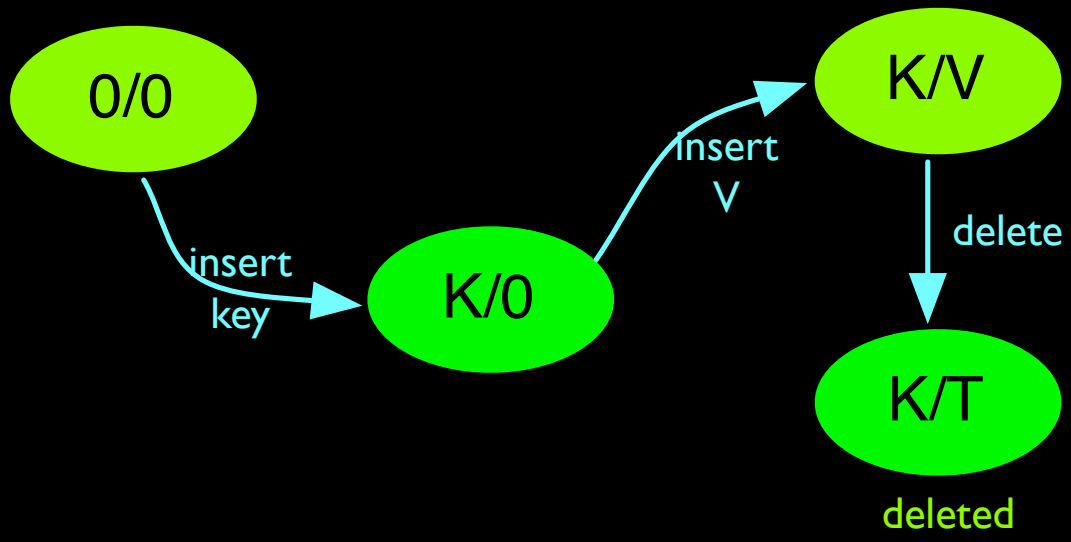
Initial

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

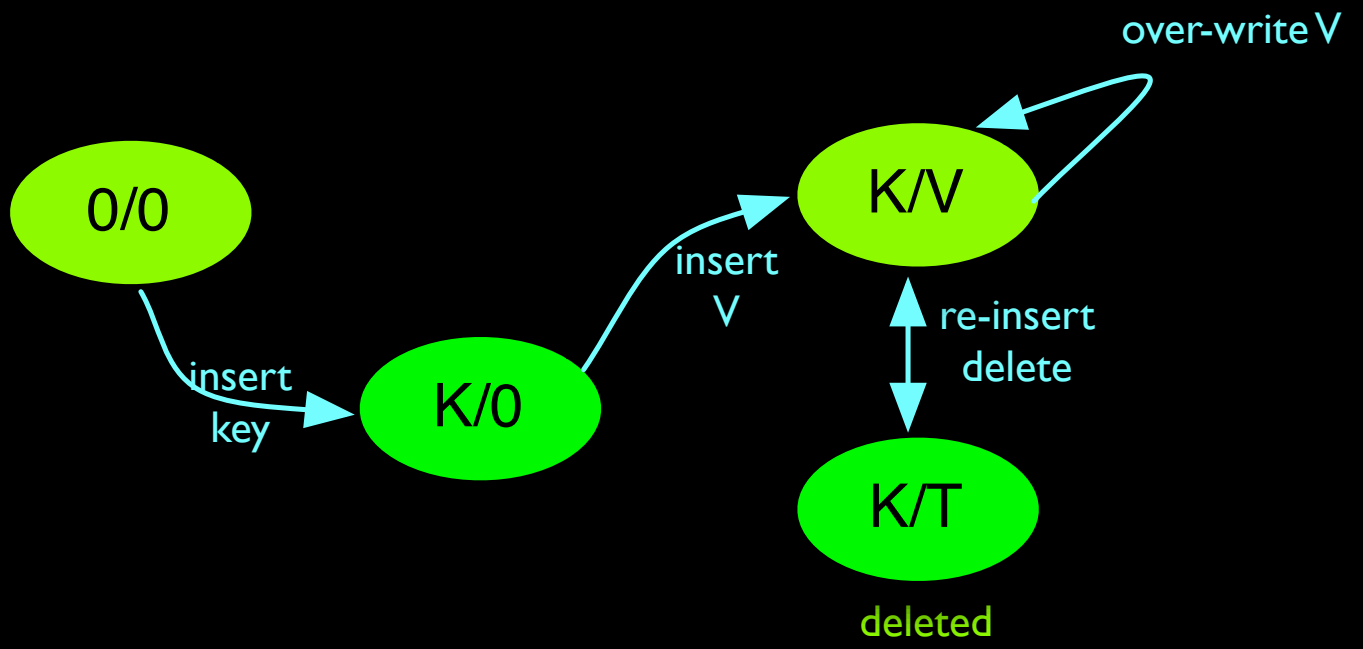

0/0
claim slot
insert
key
K/0
naked key



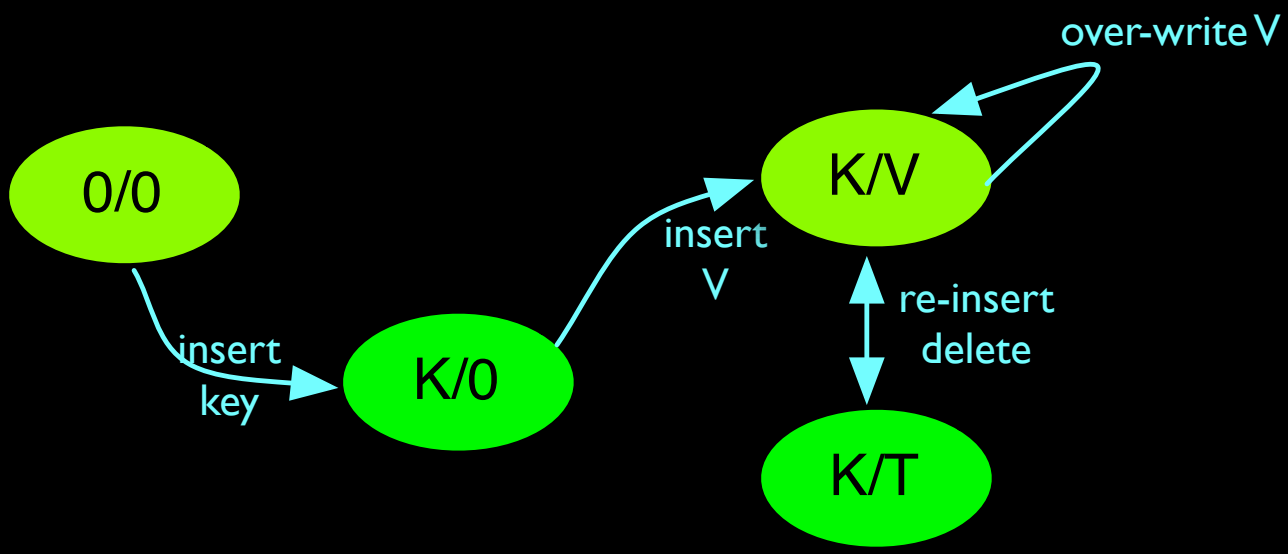
initial set of value

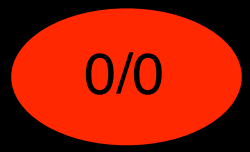


Tombstone marks delete, key remains

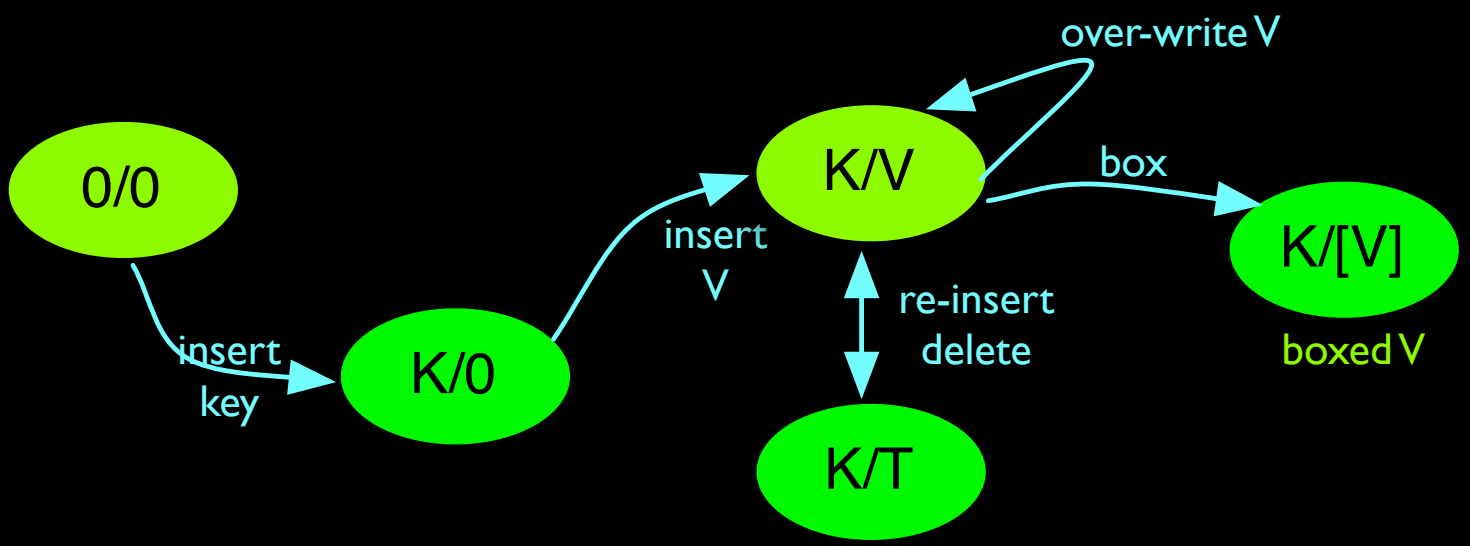


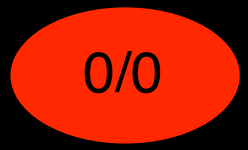
operations use same key slot



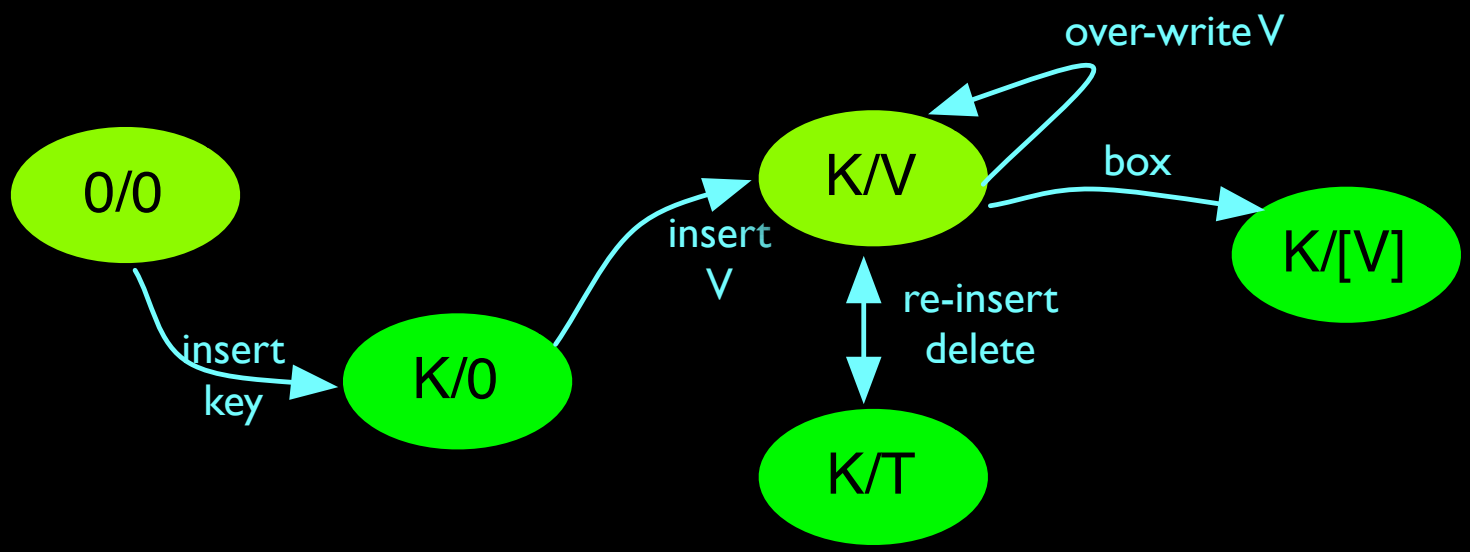


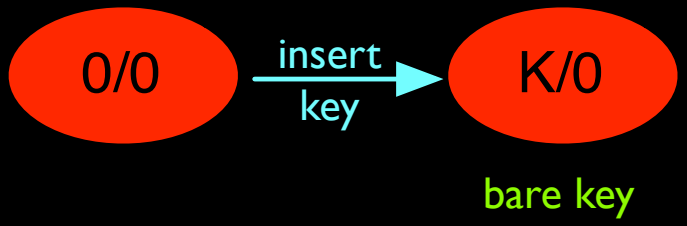
resize triggered
new array created



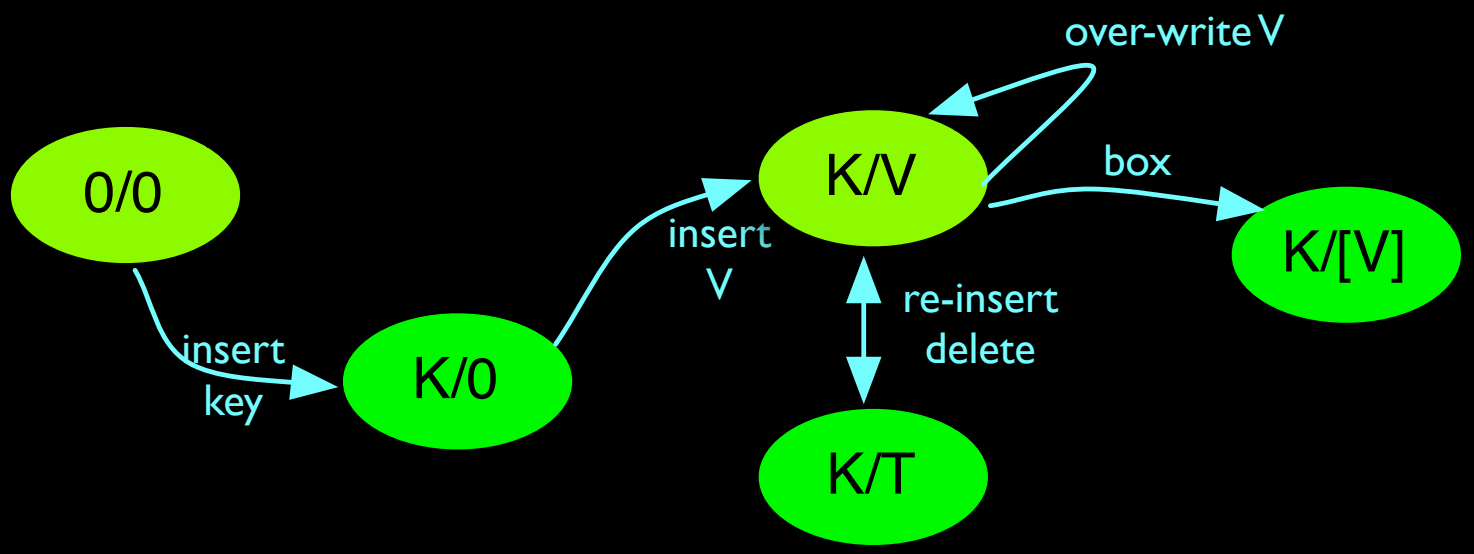


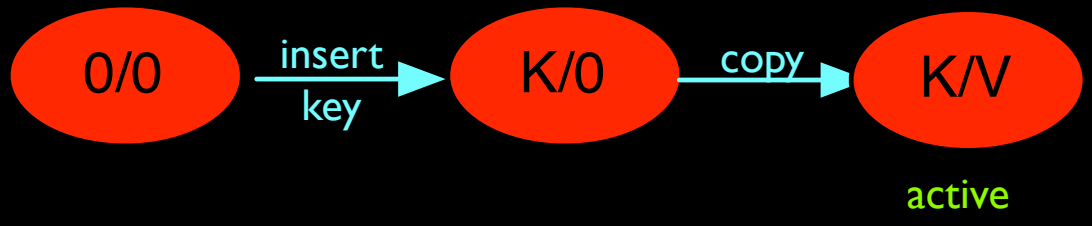
boxing prevents further changes



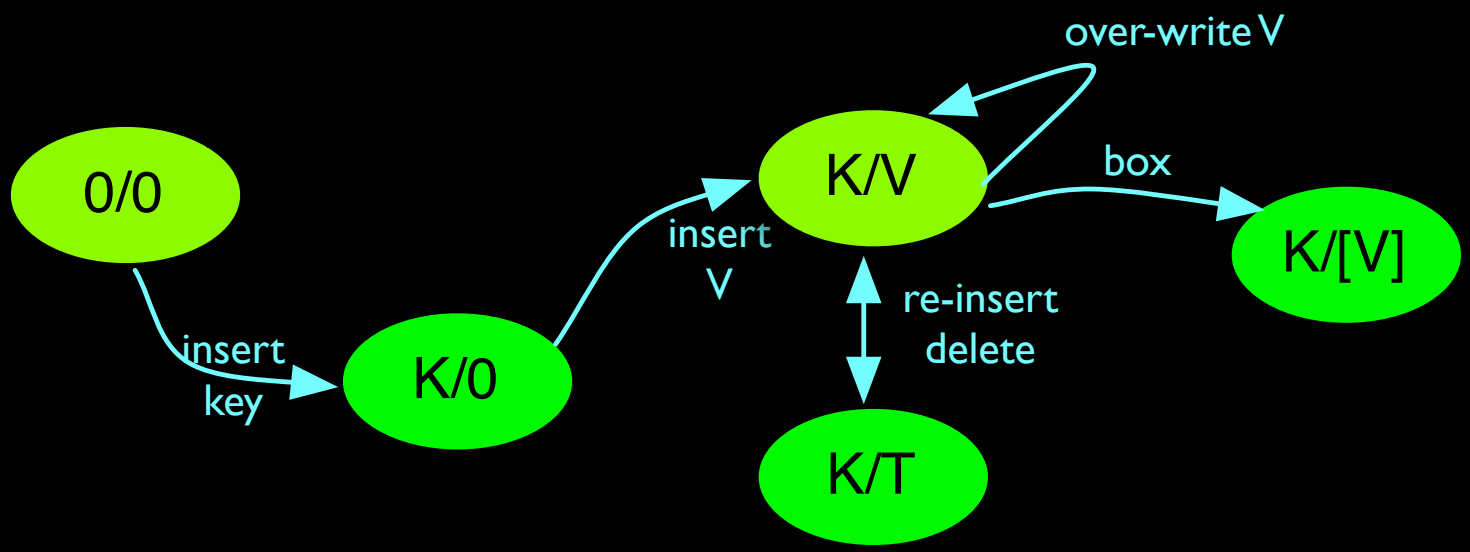


claim key slot in new table

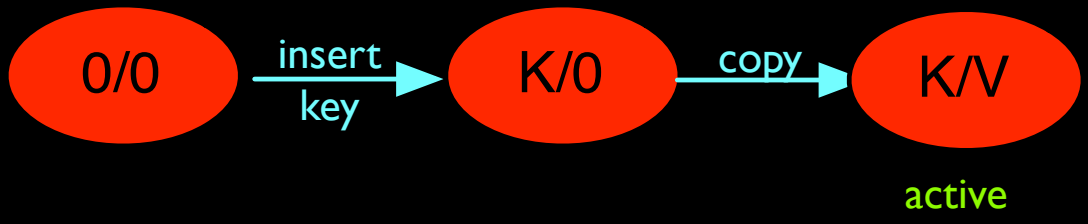




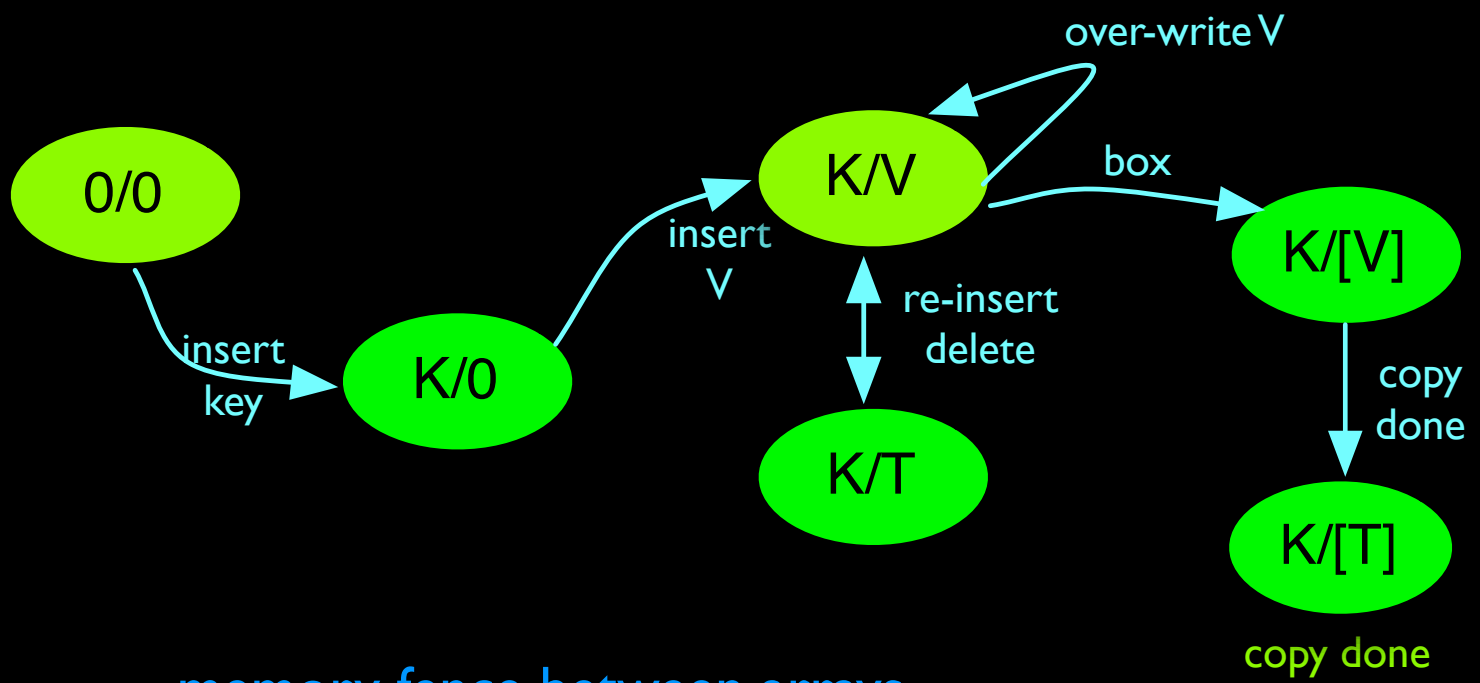
copy V without box



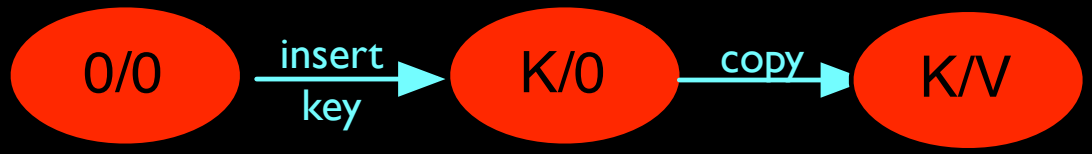
memory fence between arrays



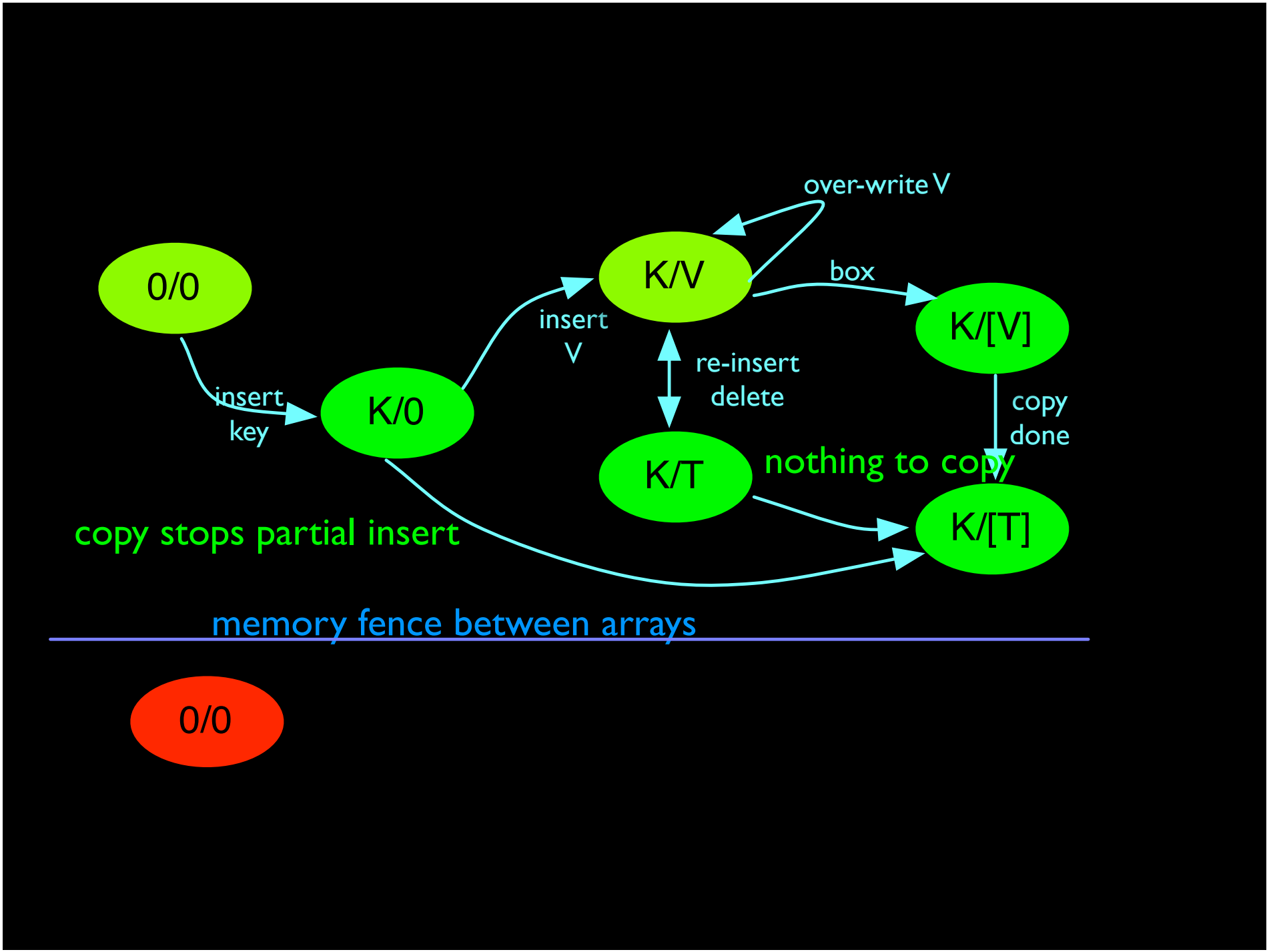
fence after writing new array and before copy done

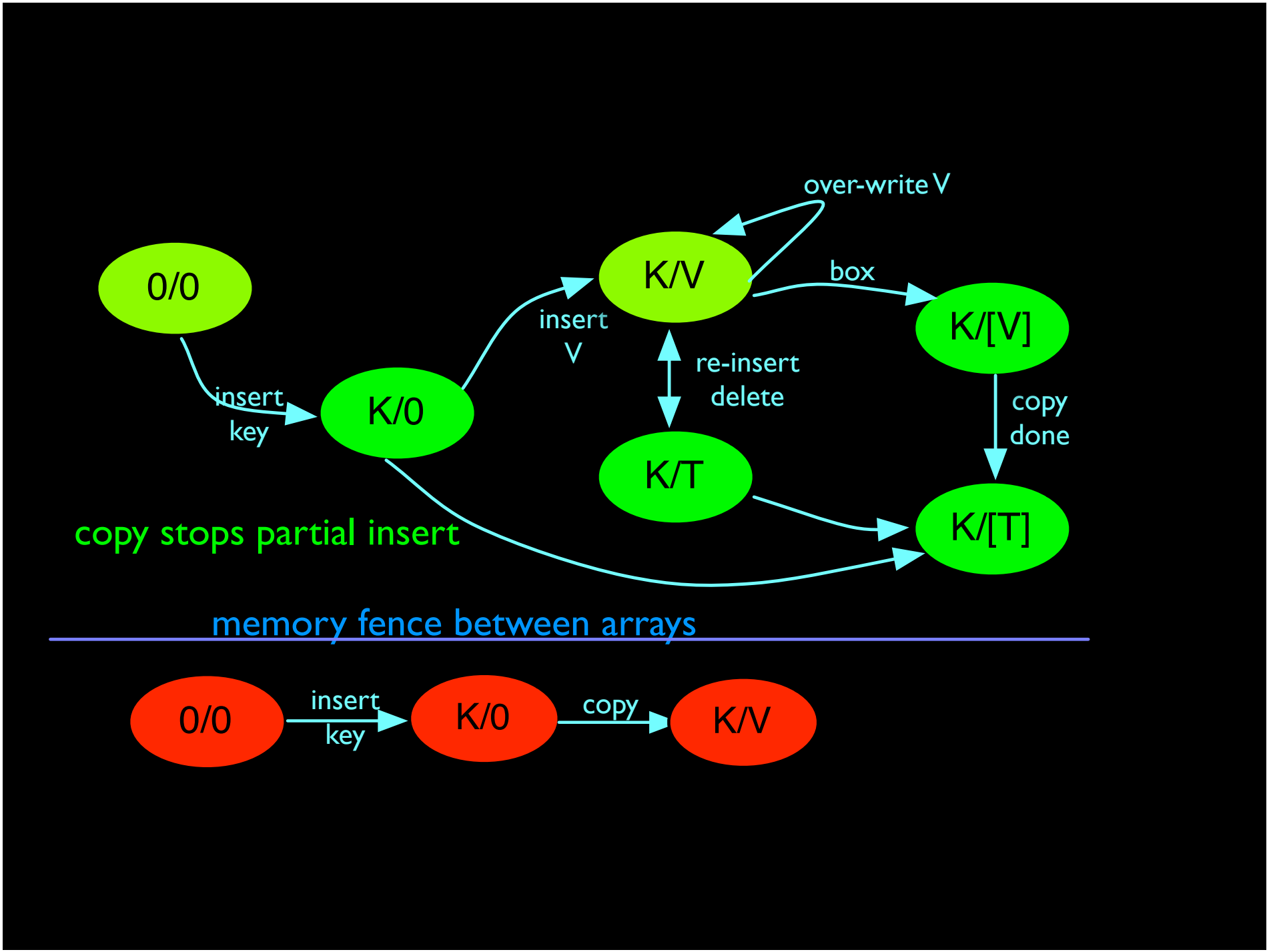


memory fence between arrays

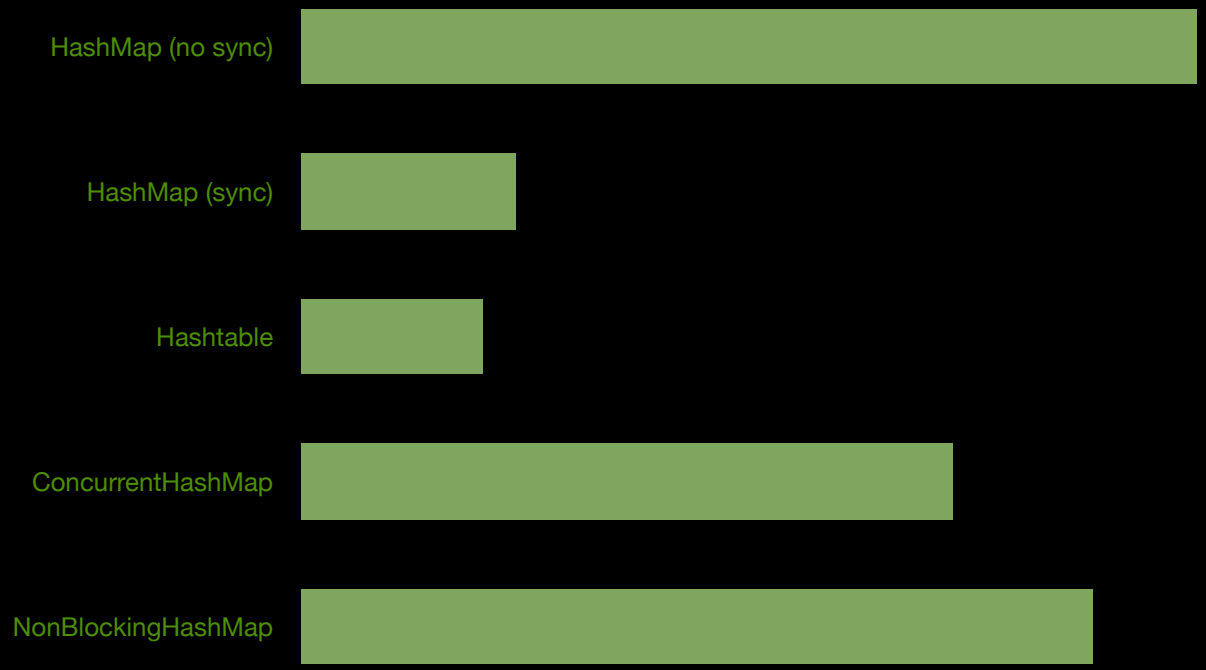


fence after writing new array and before copy done

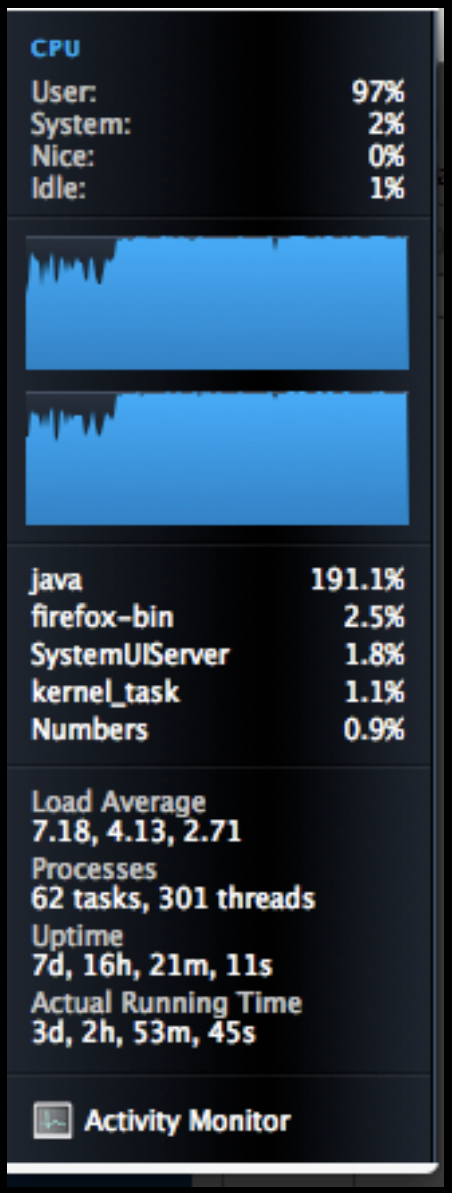

over-write V
0/0
K/V
box
K/[V]
insert
V
re-insert
delete
insert
key
K/0
copy
done
K/T
nothing to copy
K/[T]
copy stops partial insert
memory fence between arrays
0/0


over-write V
0/0
K/V
box
insert
V
K/[V]
re-insert
delete
insert
key
K/0
copy
done
K/T
K/[T]
copy stops partial insert
memory fence between arrays
0/0
insert
key
K/0
copy
K/V

Blackboard Reloaded



Blackboard Reloaded


CPU
User: 97%
System: 2%
Nice: 0%
Idle: 1%
java 191.1%
firefox-bin 2.5%
SystemUIServer 1.8%
kernel_task 1.1%
Numbers 0.9%
Load Average
7.18, 4.13, 2.71
Processes
62 tasks, 301 threads
Uptime
7d, 16h, 21m, 11s
Actual Running Time
3d, 2h, 53m, 45s
Activity Monitor

scales linerarly up too 1000 CPUs

Fully concurrent lock-less FIFO?

Stripe on queues and randomly pick one

stripe ad-absurdum

insert searchs for null CAS down value

read searchs for value and CAS down null

too large read spin, too small inserts spin

resize is earier, promote when entire array is tombstoned

Questions?