MULTICORE PROGRAMMING

Concurrent Hash Tables

Lecture 7

Trevor Brown

ANNOUNCEMENTS

- A3 due soon
- A4 to be released just after
- **Two** week assignment
- Topic is hash tables (similar to last class & today, but **less fancy** expansion)

LAST TIME

- Ordered vs unordered sets
- Lock-based, fixed size, probing, insert-only hash tables
- Deletion via **tombstones**
- A lock-free implementation

- This time:
    - Probing vs chaining
    - Hash function quality
    - **Hash table expansion**

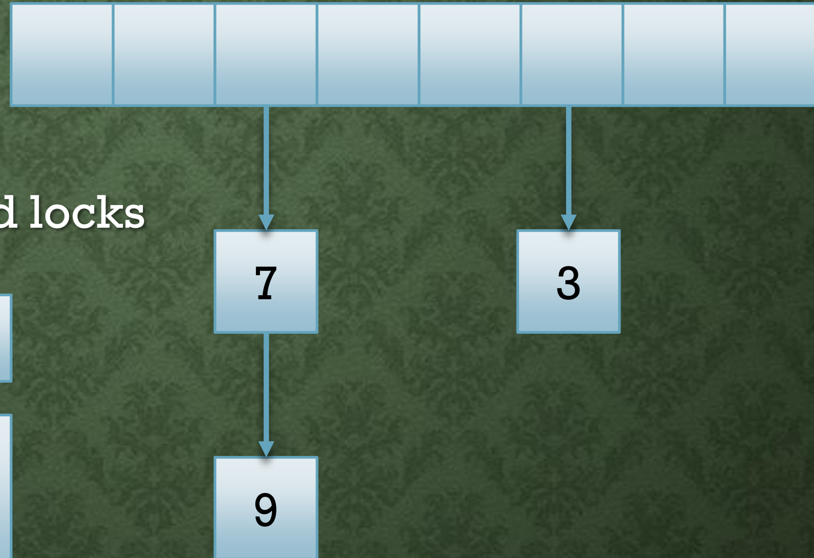
CHAINING: AN ALTERNATIVE TO PROBING

- Array of **concurrent linked lists**
- (Any concurrent linked list works)
  - Can use sequential + coarse grained locks

Advantages? Disadvantages?

Reduced need for **table expansion**
Deletion is easy (handled by the **list**)

Lists add overhead (vs ~one CAS)
Cache misses!



Insert(7) hashes to 2

Insert(3) hashes to 5

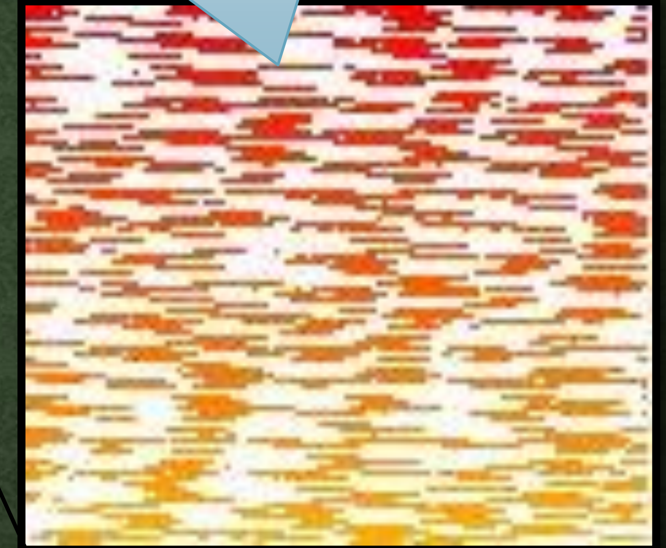
Insert(9) hashes to 2

BAD HASH FUNCTIONS

Idea: insert keys into a **probing** hash table, and visualize the array (row-by-row) as PNG; white = empty bucket, color = key in a bucket

Hashing numeric strings with:
SDBM

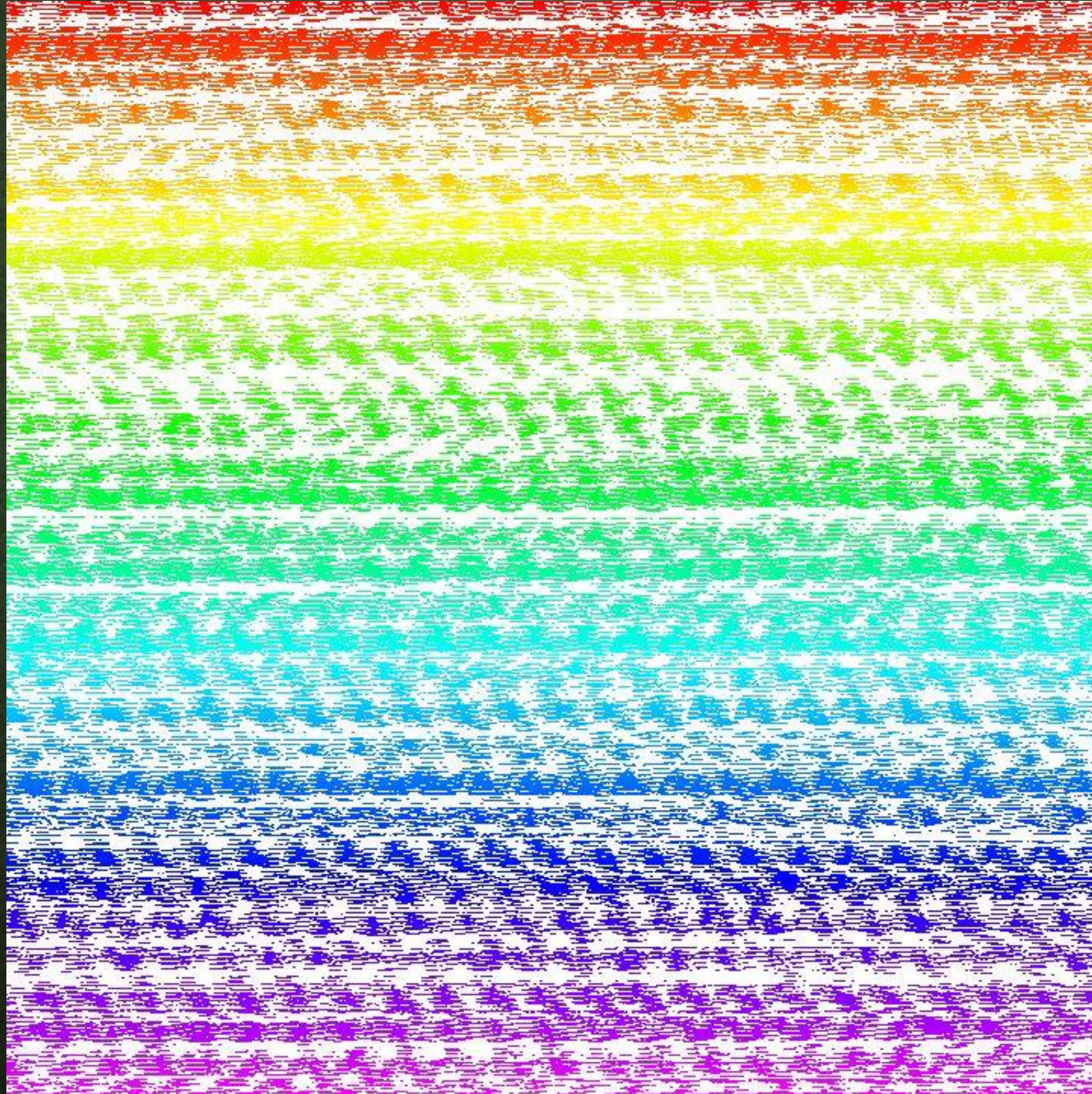
Likely represents many keys that **hash to the same bucket**, causing **excessive probing**



https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

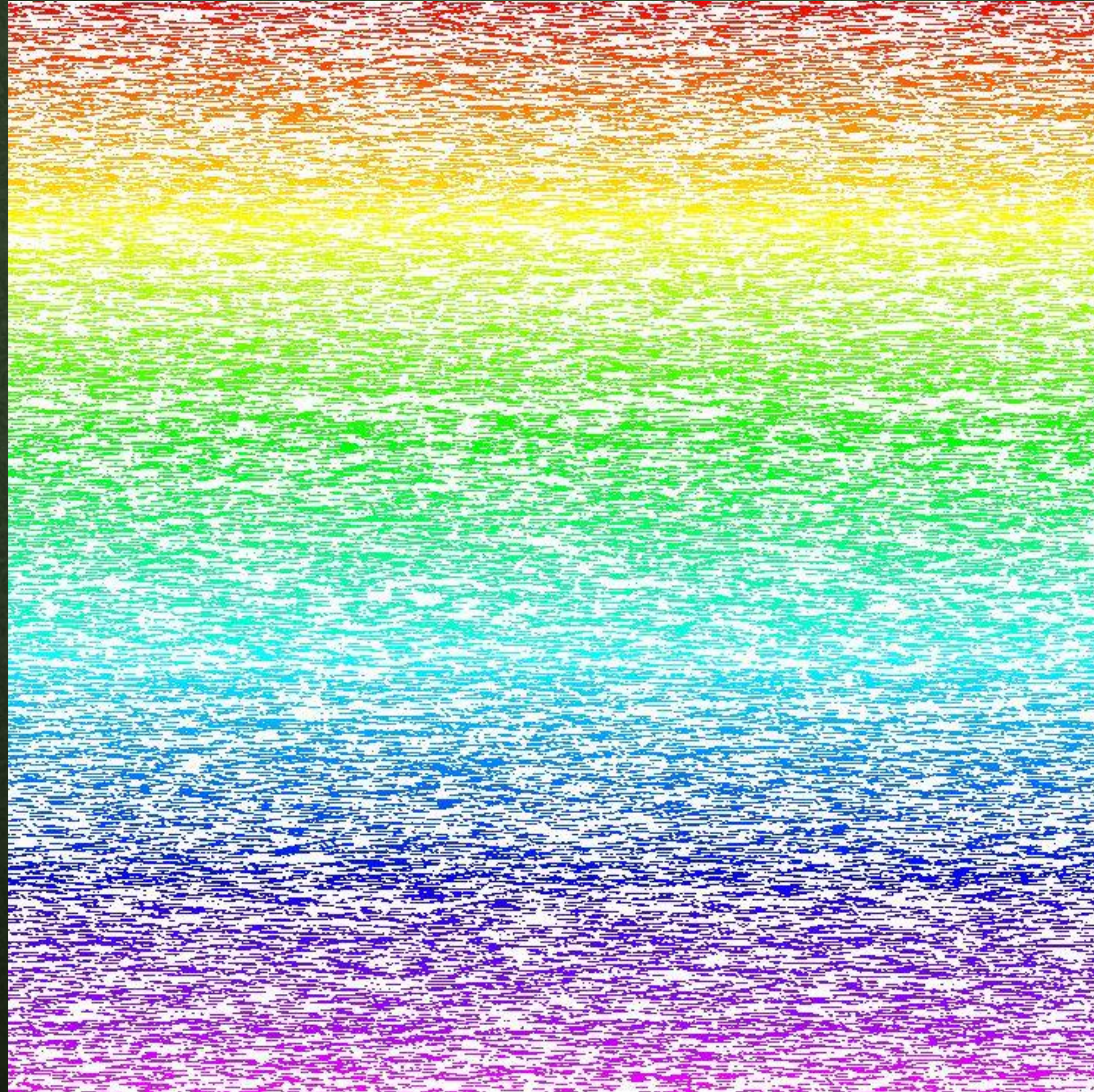
BAD HASH FUNCTIONS

Hashing numeric strings with:
DBJ2A



BAD HASH FUNCTIONS

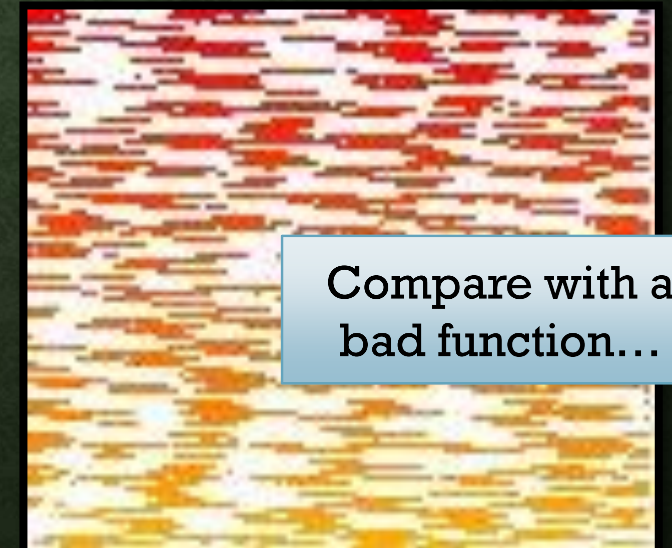
Hashing numeric strings with:
FNV1



BETTER HASH FUNCTIONS

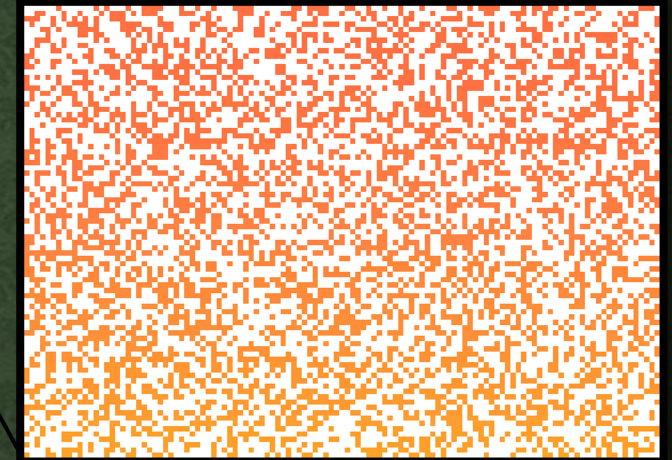
Hashing numeric strings with:
FNV1-A

Weird anomaly…
generated numbers
often alternate:
odd-even-odd-even !



Compare with a
bad function…

GOOD HASH FUNCTIONS



Hashing numeric strings with: **Murmur2**

Newer version of this exists: **Murmur3**…

Can go further: **cryptographic** hash functions (but they are **slow**!)

EFFICIENTLY EXPANDING A HASH TABLE

CONCURRENT HASH TABLES: FAST AND GENERAL(?)! [MSD2016]

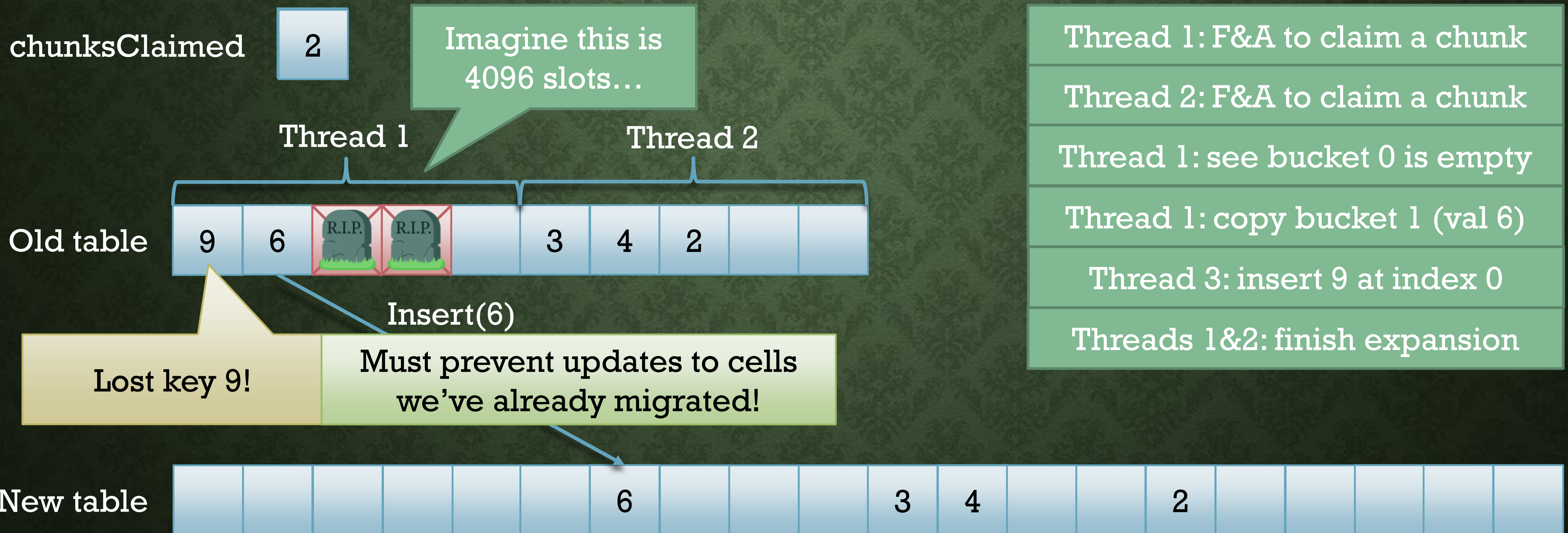
Teaching something *close* to this…

- Issues to solve
  - When to expand?
    - Too much probing? Estimate of number of keys in the table vs capacity?
  - Which threads will perform the expansion?
    - Enslaving user threads that access the table? Custom thread pool?
  - How to expand?
    - Block updates when expansion is happening? Always allow updates?
    - Hard to always allow updates
    - "we are fine with small amounts of (b)locking, as long as it improves overall performance" (a healthy attitude)

HIGH LEVEL IDEA

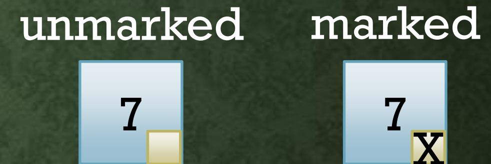
- When estimated number of keys > 50% of table capacity,
  expand to ~4x estimated number of keys & migrate contents into the new table
- Naïve idea: global lock on table, copy with one thread --- inefficient!
- Partition old table into chunks (default size 4096)
  - Numbered 0, 1, 2, …, floor(capacity / 4096)
- Threads use fetch & add to "claim" chunks to migrate
- Migrate a chunk to the new table by performing insert() on each key

WHAT IF EXPANSION IS CONCURRENT WITH UPDATES?

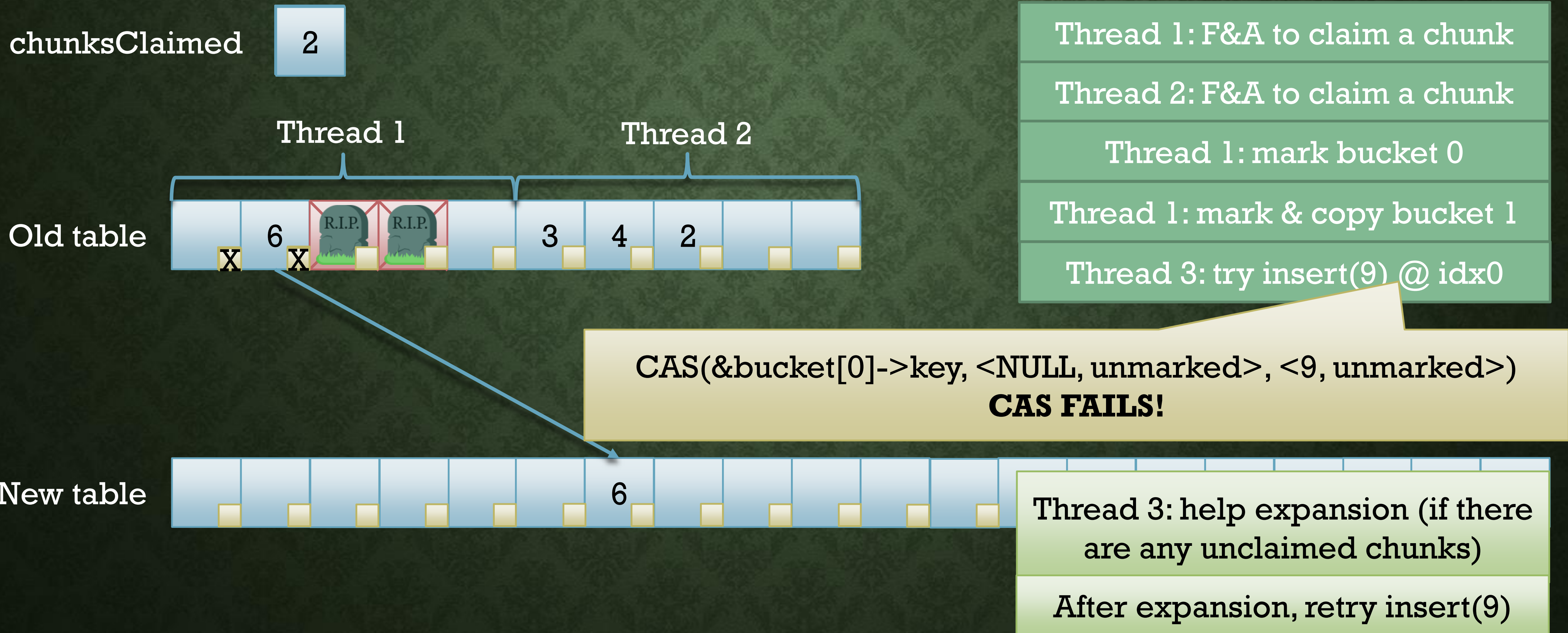


SOLUTION: MARKING

- Steal a bit from each **key** field
- Use it to store a **mark**
- Invariant: a bucket with a **marked** key **cannot** be modified
  - In expansion, **mark** each bucket before migrating it
    - currKey = bucket->key
    - CAS(bucket->key, currKey, currKey | MARKED_MASK)
  - In insert/delete, before performing CAS(bucket->key, currKey, newKey), must verify that currKey is **not marked** (otherwise, help with expansion)



HOW MARKING HELPS



DETECTING TABLE > 50% FULL

- Using the approximate counter
- Recall: how large can the error be in the approximate counter?
  - $Error = c \cdot numThreads^2$
  - Insignificant for large tables (for c=1, 100 threads, error is 10000: 1% of 1M keys)
- Could be a problem when the table is small…
  - What happens if error ~= table capacity?
  - Won't realize table is full until many more operations are done!
- Suggestions to fix this?
  - Modify approximate counter to add a slow "accurateGet()" operation
  - If we do "too much" probing in an operation (i.e., we are already paying a high cost), run accurateGet() to check if we should expand

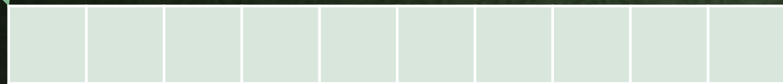
ROUGH IMPLEMENTATION **SKETCH**

Without expansion

```
struct hashmap
1    char padding0[64];
2    atomic<uint64_t> * data;
3    int capacity;
4    char padding1[64];
     /* code for operations ... */
```

data

Data is a pointer
(to the start of an array)



Each slot is an
atomic<int>

With expansion

Atomic pointer to
current table struct

```
struct hashmap
1    char padding0[64];
2    atomic<table *> currentTable;
3    char padding1[64];
     /* code for operations ... */
```

```
struct table
1    char padding0[64];
2    atomic<int> * data;
3    atomic<int> * old;
4    int capacity;
5    int oldCapacity;
6    counter * approxSize;
7    atomic<int> chunksClaimed;
8    atomic<bool> expandingNow;
9    char padding1[64];
```

old stays around
so expansion can
be done…

Can cause some
false sharing?

Erratum: changing
this in next lecture

IMPLEMENTATION SKETCH

```
int hashmap::insert(int key)
  table * t = currentTable;
  int h = hash(key);
  for (int i=0; i < t->capacity; ++i) {
    if (expandAsNeeded(t, i)) return insert(key);

    int index = (h+i) % t->capacity;
    int found = t->data[index];
    if (found & MARKED_MASK) return insert(key);
    else if (found == key) return false;
    else if (found == NULL) {
      if (CAS(&t->data[index], NULL, key)) return true;
      else {
        found = t->data[index];
        if (found & MARKED_MASK) return insert(key);
        else if (found == key) return false;
  } } }
  assert(false);
```

Check if we need to expand, and start expansion as necessary, or help ongoing expansion. If we start or help expansion, retry our insert (in the new table)

More details on next slide…

Found evidence of expansion… restart to help / get into the new table

Could fail CAS because another thread **marked** or **inserted**

Sketch of expandAsNeeded(t, i)

Clarifying and expanding this description in the next lecture!

- Check if t->expandingNow
- If so, call **helpExpansion(t)** to try to help the ongoing expansion
  - (**helpExpansion(t)**: repeatedly until all chunks in **t** are claimed: FAA(t->chunksClaimed, 1), and if return value was a valid chunk, migrate its contents from t->old to t->new **via insert()**)
- Else, check t->counter->**get()** to see if we should expand
- If we should expand, call **startExpansion(t)**
  - (**startExpansion(t)**: create new **table struct** with larger data[] and **CAS** it into **currentTable**; if CAS fails, another thread started expansion, and we will help it upon retrying our insert)
- Else, check probing to see if it's "excessive" (i.e., check if **i** is "very large" – heuristic!)
- If probing is "excessive," check t->counter->**getAccurate()** to see if we should expand
- If we should expand, call **startExpansion(t)**