MULTICORE PROGRAMMING

Hash Table Expansion, Linked Data Structures

Lecture 8

Trevor Brown

LAST TIME

- Probing vs chaining
- Hash function quality
- Started hash table expansion

- This time:
  - Finishing hash table expansion
  - Starting **linked data structures**

HASH TABLE EXPANSION

Clarifying and finishing up after last time

RECALL: ROUGH IMPLEMENTATION **SKETCH**

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

Atomic pointer to current table struct

```
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<int> chunksDone;
9    char padding1[64];
```

old stays around so expansion can be done…

Erratum: changed since last lecture!

Initial table struct:
old = NULL
oldCapacity = 0
chunksClaimed = 0
chunksDone = 0

Note total number of chunks in old is 0, so chunksDone = 0 means expansion is "done."

Result of createNewTableStruct(t):
old = t->data
oldCapacity = t->capacity
chunksClaimed = 0
chunksDone = 0

RECALL: CODE FROM LAST TIME

```
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)

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

Clarifying the last lecture

```
bool hashmap::expandAsNeeded(t, i)
  helpExpansion(t);
  if (t->approxSize->get() > t->capacity/2) or
     (i > 10 and t->approxSize->getAccurate() > t->capacity/2) {
    startExpansion(t);
    return true;
  }
  return false;

void hashmap::helpExpansion(t)
  int totalOldChunks = ceil(t->oldCapacity / 4096);
  while (t->chunksClaimed < totalOldChunks) {
    int myChunk = FAA(&t->chunksClaimed, 1);
    if (myChunk < totalOldChunks) {
      migrate(t, myChunk);
      FAA(&t->chunksDone, 1);
  } }
  wait until (t->chunksDone == totalOldChunks)

void hashmap::startExpansion(t)
  if (currentTable == t) {
    t_new = createNewTableStruct(t);
    if not CAS(&currentTable, t, t_new) delete t_new;
  }
  helpExpansion(currentTable);
```

Note: for initial table struct, this is a no-op

What about this?

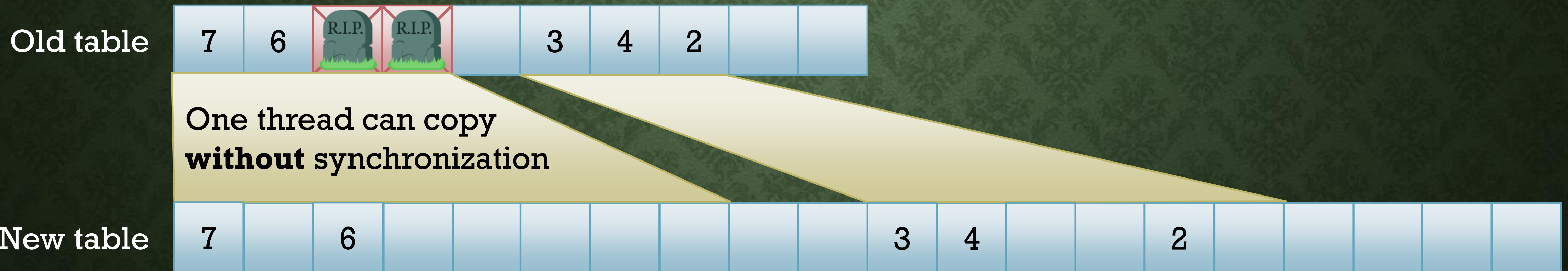
Important! Last time I made a mistake… Actually **cannot** let threads insert into **the new table** until after expansion is **done!**

Wait until expansion is finished before returning!

MAKING MIGRATION MORE EFFICIENT

- **Typical index function** to get a **bucket** index from a key:
  - **index = hash(key) % capacity**
  - If capacity doubles, indexes of keys are **scrambled**
    - Hash 23 in array of size 12: bucket 11 → in array of size 24: bucket 23
    - Hash 13 in array of size 12: bucket 1 → in array of size 24: bucket 13
- **Scaled index function**
  - **index = floor(hash(key) / largestHashPossible * capacity)**
  - If capacity doubles, indexes of keys are **doubled**
    - In array of size 12: bucket 11 → in array of size 24: bucket 22
    - In array of size 12: bucket 1 → in array of size 24: bucket 2
- With predictable indexes, can expand more efficiently!

IDEA



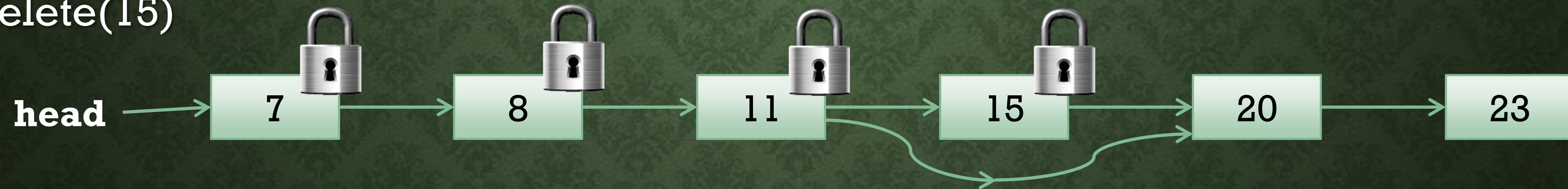
MORE COMPLEX DATA STRUCTURES

WHAT ELSE IS WORTH UNDERSTANDING?

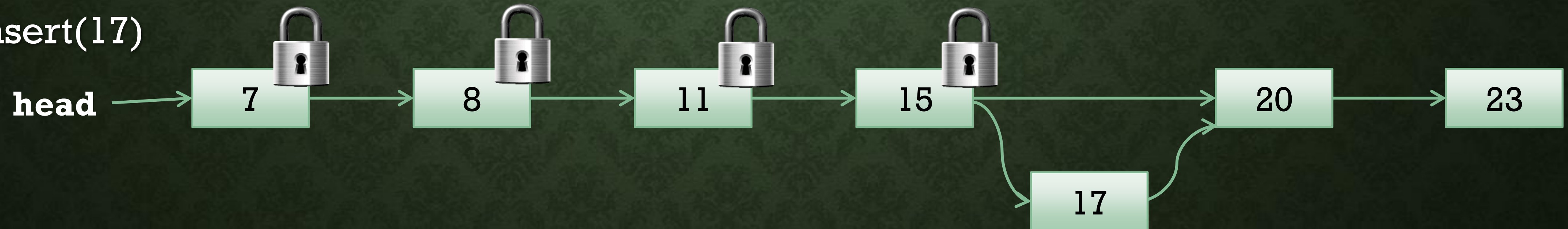
- We've seen hash tables…
- What about node based data structures?
  - (That aren't just a single pointer like stacks, or two pointers like queues)
- Singly-linked lists, doubly-linked lists, skip-lists, trees, tries, hash tries, …
- New challenges:
  - Nodes get deleted when threads might be trying to work on them
  - Operations may require atomic changes to multiple nodes

LOCK-BASED SINGLY-LINKED LISTS

- Ordered set implemented with singly-linked list
- Hand-over-hand locking discipline:
  - must lock a node before accessing it
  - Can only acquire a lock on a node:
  - **if** it is the list **head**, or
    if you already hold a lock on the previous node
- Delete(15)



- Insert(17)



Is this a good approach?

Locking causes **many** cache invalidations, even for searches!

Should **avoid locking** while searching/traversing the list!