MULTICORE PROGRAMMING

Concurrent Hash Tables

Lecture 6

Trevor Brown

LAST TIME

- Stack performance
- Difficulties in **using** ordered data structures
- Harnessing disorder: relaxed (multi-)queue

- This time:
  - Unordered **Set** data structures (similar to dictionaries / maps)

ORDERED SET OBJECT

- Operations
  - Search(key):
    - return true if key is in the set, and false otherwise
  - Insert(key):
    - if key is not in the set, add it and return true, otherwise return false
  - Delete(key):
    - if key is in the set, delete it and return true, otherwise return false
  - Successor(key):
    - returns the smallest key in the set that is larger than key
    - can use to **iterate** over the set contents

Requires strict ordering – intuitively limits concurrency

UNORDERED SET OBJECT

- Same as ordered set, but no Successor operation
  - Just: Search, Insert and Delete

Eliminates ordering constraints – possibly better concurrency

WHEN SHOULD WE USE EACH?

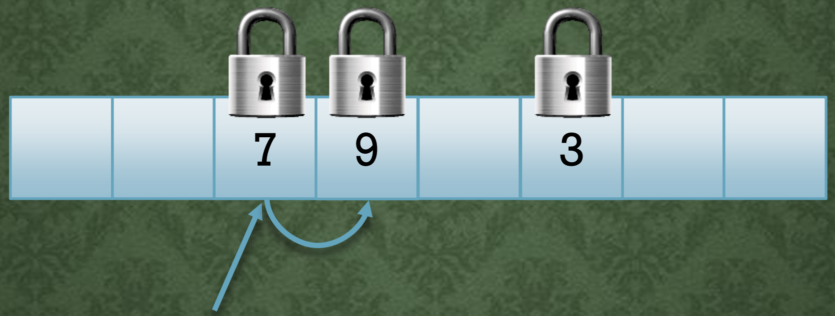
- Use an ordered set (search tree, skip list) if you **really** need the Successor operation
  - Ordered traversals that are concurrent with updates to the set
  - Operations that update predecessors/successors of keys
  - Certain types of spatial / geometric algorithms
- Use an unordered set (hash table, hash trie) otherwise
  - **Much** better performance if you don't need order
  - Also much easier to shard & distribute

BUILDING A CONCURRENT HASH TABLE

- Some things to think about
  - Supporting deletions or insert-only?
  - Using locking or lock-free techniques?
  - Using chaining or probing?
- Insert-only hash table object with a fixed **capacity**:
  - Contains(key): returns true if key is present, and false otherwise.
  - Insert(key): If key is present, return false,
    else if table contains **capacity** keys, return FULL,
    else insert key and return true.

SIMPLEST HASH TABLE POSSIBLE: LOCKING, INSERT-ONLY, PROBING, FIXED CAPACITY

- Shared array: data[]
- Each element is a **bucket**
- **bucket** has fields:
  - key (the key inserted there), m (a **mutex** to protect that bucket)



Insert(7) hashes to 2

Insert(3) hashes to 5

Insert(9) hashes to 2

- Simplest possible locking protocol:
  - Must lock a bucket before **any** access (read or write) to its key

IMPLEMENTATION AND LP SKETCH

```
int insert(int key)
int h = hash(key);
for (int i=0;i<capacity;++i) {
  int index = (h+i) % capacity;
  data[index].m.lock();
  int found = data[index].key;
  if (found == key) {
    data[index].m.unlock();
    return false;
  } else if (found == NULL) {
    data[index].key = key;
    data[index].m.unlock();
    return true;
  }
  data[index].m.unlock();
}
return FULL;
```

<u>Intuitive sketch</u> of how we choose LPs

- Important lemmas
  - The key in a bucket changes only **once**, from NULL to non-NULL
  - If we probe a sequence of buckets at times $\mathbf{t_1} < \mathbf{t_2} < \ldots < \mathbf{t_n}$, and see non-NULL keys, then we would see the **same keys** if we were to do all probing **atomically** at time $\mathbf{t_n}$ **<u>or later</u>**

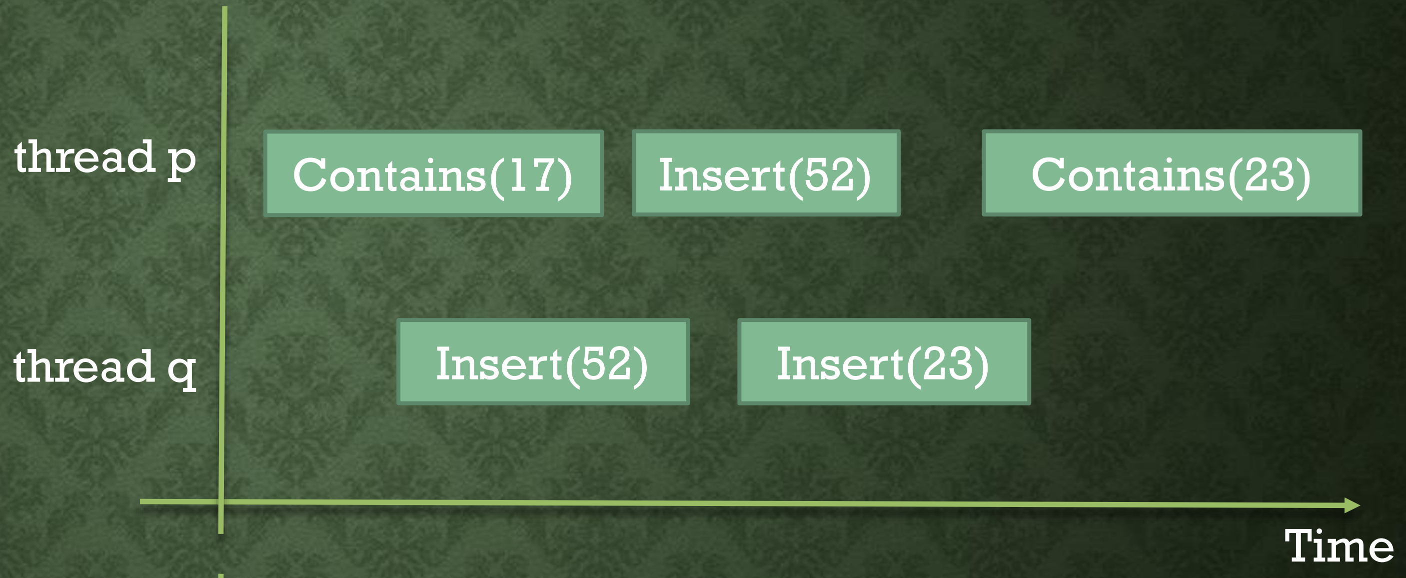
What do I mean by: "<u>if we were</u> to do all probing <u>atomically</u>?"

WHAT DO I MEAN BY:
"IF WE WERE TO DO ALL PROBING ATOMICALLY?"

We are **not** making a formal argument. Just building intuition about why the chosen LPs are **plausible**.

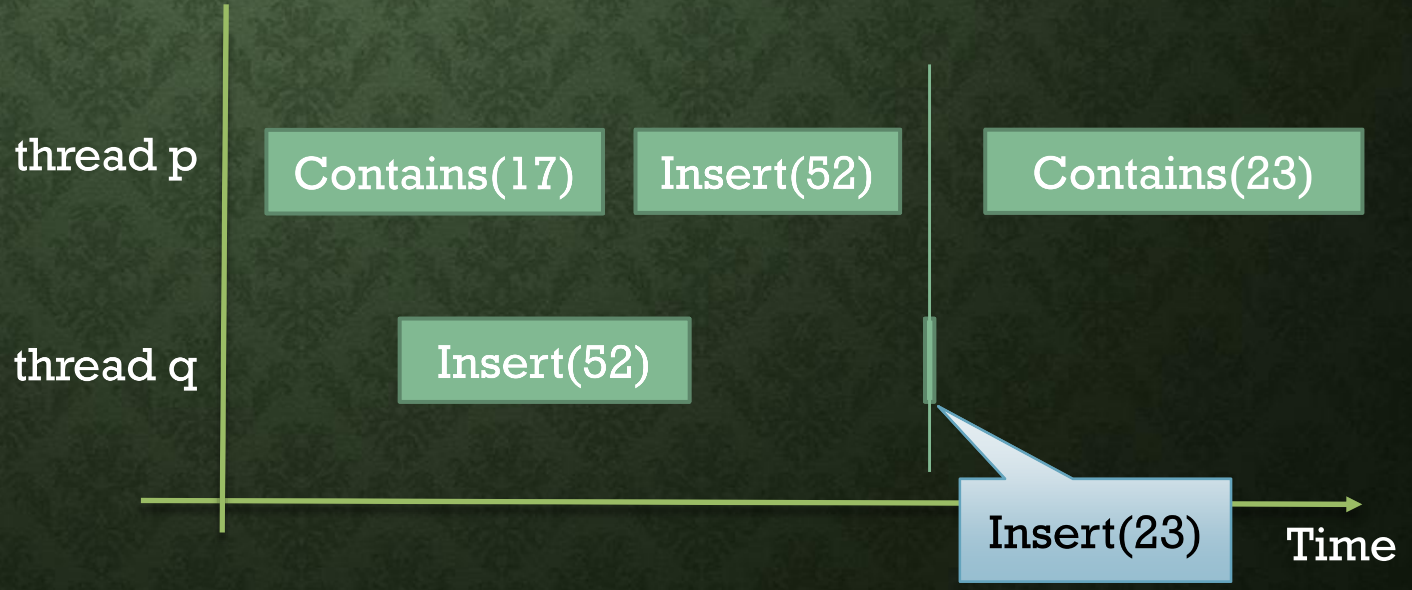
We are **imagining** some **concurrent** execution E

E



To try to find **linearization bugs** caused by our choice of LPs, I am imagining a similar execution E' where **some** insert happens **atomically at its LP**.

E'



IMPLEMENTATION AND LP SKETCH

```
int insert(int key)
1    int h = hash(key);
2    for (int i=0;i<capacity;++i) {
3      int index = (h+i) % capacity;
4      data[index].m.lock();
5      int found = data[index].key;
6      if (found == key) {
7        data[index].m.unlock();
8        return false;
9      } else if (found == NULL) {
10       data[index].key = key;
11       data[index].m.unlock();
12       return true;
13     }
14     data[index].m.unlock();
15   }
16   return FULL;
```

Intuitive sketch of how we choose LPs

- Important lemmas
  - The key in a bucket changes only **once**, from NULL to non-NULL
  - If we probe a sequence of buckets at times $t_1 < t_2 < \ldots < t_n$, and see non-NULL keys, then we would see the **same keys** if we were to do all probing **atomically** at time t_n **or later**
- **Linearization points (LPs)**
  - Return@8: last read of data[index].key
  - Return@12: write to data[index].key
  - Return@16: last read of data[index].key

LP CHOICE INTUITION/SKETCH

```
int insert(int key)
1   int h = hash(key);
2   for (int i=0;i<capacity;++i) {
3     int index = (h+i) % capacity;
4     data[index].m.lock();
5     int found = data[index].key;
6     if (found == key) {
7       data[index].m.unlock();
8       return false;
9     } else if (found == NULL) {
10      data[index].key = key;
11      data[index].m.unlock();
12      return true;
13    }
14    data[index].m.unlock();
15  }
16  return FULL;
```

- Case Return@16
  (LP: last read of data[index].key)
- Easy case
- We returned at 16 after seeing data[index].key != NULL for **every** index
- Non-NULL buckets don't change
- So, when we do the last read of data[index] .key, all buckets are full!
- So, our insert would return FULL if performed atomically at the LP

LP CHOICE INTUITION/SKETCH

```
int insert(int key)
1   int h = hash(key);
2   for (int i=0;i<capacity;++i) {
3     int index = (h+i) % capacity;
4     data[index].m.lock();
5     int found = data[index].key;
6     if (found == key) {
7       data[index].m.unlock();
8       return false;
9     } else if (found == NULL) {
10      data[index].key = key;
11      data[index].m.unlock();
12      return true;
13    }
14    data[index].m.unlock(
15  }
16  return FULL;
```

- Case Return@8 (LP: last read of data[index].key)
- Suppose we return@8 in the **k**th loop iteration
  - In the first k-1 iterations, we see data[index].key != NULL and data[index].key != key
- Claim: if our insert(key) ran **atomically** at the **LP**, it would perform k iterations, and see the same values as it sees in the concurrent execution
- In its kth it[...]key) reads data[in[...]ex] at the same ti[...]e as the **c[...]ncurrent** i[...]t(key), so it sees dat[...]ndex].key [...] key

[...]t return

...

Insert(key)

time

Iteration 0 reads data[index]

!= NULL, != key,
& does not change!

Iteration k-1 reads data[index]

!= NULL, != key,
& does not change!

LP: Iteration k reads data[index]

The values read in previous iterations are **unchanged**!

LP CHOICE INTUITION/SKETCH

```
int insert(int key)
1    int h = hash(key);
2    for (int i=0;i<capacity;++i) {
3      int index = (h+i) % capacity;
4      data[index].m.lock();
5      int found = data[index].key;
6      if (found == key) {
7        data[index].m.unlock();
8        return false;
9      } else if (found == NULL) {
10       data[index].key = key;
11       data[index].m.unlock();
12       return true;
13     }
14     data[index].m.unlock();
15   }
16   return FULL;
```

- Case Return@12 (LP: write to data[index].key)
- Argument is similar to the previous case:
- Suppose we return@12 in the **k**th iteration
  - In the first k-1 iterations, we see data[index].key != NULL and data[index].key != key
- Claim: if our insert(key) ran **atomically** at the **LP**, it would perform k iterations, and see the same values as it sees in the concurrent execution
- In its k^{th} iteration, the **linearized** insert(key) reads data[index].key at the same time as the **concurrent** insert(key), so it sees data[index].key == NULL
- So, it returns true

IMPLEMENTING CONTAINS

```
int contains(int key)
  int h = hash(key);
  for (int i=0;i<capacity;++i) {
   int index = (h+i) % capacity;
   data[index].m.lock();
   int found = data[index].key;
   if (found == key) {
    data[index].m.unlock();
    return true;
   } else if (found == NULL) {
    data[index].m.unlock();
    return false;
   }
   data[index].m.unlock();
  }
  return false;
```

- Quite similar to insertion
- Still lock before accessing buckets
  - Even though we are not changing anything… Why?
  - *Ex: is this necessary?*
- Linearization points?
  - Always last read of data[index].key

WHAT ABOUT DELETION?

- For example, in our lock-based, fixed size, probing hash table:

Insert(7) hashes to 2

Insert(3) hashes to 5

Insert(9) hashes to 2

Delete(7) hashes to 2

Delete(9) hashes to 2

Incorrect!

A WORKAROUND: TOMBSTONES

- Introduce a special key value called a TOMBSTONE
- To delete a key
  - Instead of setting key = NULL, set it to TOMBSTONE
- TOMBSTONE essentially acts like a regular key
  (that never matches a key you are trying to find/insert/delete)
  - Insertions must still probe past it to find NULL
  - Deletions must still probe past it to find the desired key

HOW TOMBSTONES FIX OUR PROBLEM

Insert(7) hashes to 2

Insert(3) hashes to 5

Insert(9) hashes to 2

Delete(7) hashes to 2

Delete(9) hashes to 2

DELETE WITH TOMBSTONES

```
bool erase(int key)
1    int h = hash(key);
2    for (int i=0;i<capacity;++i) {
3      int index = (h+i) % capacity;
4      data[index].m.lock();
5      int found = data[index].key;
6      if (found == NULL) {
7        data[index].m.unlock();
8        return false;
9      } else if (found == key) {
10       data[index].key = TOMBSTONE;
11       data[index].m.unlock();
12       return true;
13     }
14     data[index].m.unlock();
15   }
16   return false;
```

- Insert and contains are **mostly unchanged**
  - They already skip past keys that do not match the argument **key**
  - TOMBSTONE acts like such a key
- Linearization points?
  - return@8: last read of data[index].key
  - return@12: write to data[index].key
  - return@16: last read of data[index].key

DOWNSIDES OF TOMBSTONES?

- TOMBSTONEs are never cleaned up
- Cleaning them up seems hard
  - Cannot remove a TOMBSTONE until there are **no** keys in the table that probed past the TOMBSTONE when they were inserted

Probes when 9 was inserted

- Thought: if we do table expansion, there's no need to copy over TOMBSTONEs…

A LOCK-FREE ALTERNATIVE

Not necessarily much faster… but good to know about.

LOCK-FREE INSERTION

```
int insert(int key)
  int h = hash(key);
  for (int i=0;i<capacity;++i) {
    int index = (h+i) % capacity;
    int found = data[index];
    if (found == key) {
        return false;
    } else if (found == NULL) {
      if (CAS(&data[index], NULL, key)) {
        return true;
      } else if (data[index] == key) {
        return false;
      }
    }
  }
  return FULL;
```

- Instead of:
  - Locking data[index], then
  - <u>if</u> it is NULL, writing to data[index].key
- Use CAS to atomically change data[index] **from NULL** to **key**
- If we **fail**, someone else inserted there
  - If it's our value, we return false (because the value is now already present)
  - Otherwise, we go to the next cell and retry

Linearization points?

LOCK-FREE CONTAINS

```
bool contains(int key)
int h = hash(key);
for (int i=0;i<capacity;++i) {
  int index = (h+i) % capacity;
  int found = data[index];
  if (found == NULL) {
    return false;
  } else if (found == key) {
    return true;
  }
}
return false;
```

- Quite similar to insertion
- Linearization points?
  - Always last read of data[index]

LOCK-FREE DELETION

```
bool erase(int key)
  int h = hash(key);
  for (int i=0;i<capacity;++i) {
    int index = (h+i) % capacity;
    int found = data[index];
    if (found == NULL) {
      return false;
    } else if (found == key) {
      return CAS(&data[index], key, TOMBSTONE);
    }
  }
  return false;
```

If CAS **succeeds**, we deleted key

If CAS **fails**, **another** thread concurrently deleted key

(return false, since **we** did not delete it)

Linearization points?

RECAP

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