**EE 382C/361C: Multicore Computing** **Fall 2016**

# Lecture 20: November 3

*Lecturer: Vijay Garg* *Scribe: Muhammad Raza Mahboob*

## Agenda

This lecture focused on Hashing and introduction of Transactional Memory. Major concepts discussed -:

- Hashing basics
- Resizing Problem for Closed Addressing
- Chained Hashing
- Transactional Memory

## 20.1 Introduction

Link list provides us with operations having O(n) complexity. In order to reduce the time complexity to constant time, we use Hashing. Hashing function is applied to the object and the result is an index (integer) and the item should be stored at that index in the bucket array. While Hashing we assume that the items are uniformly distributed so that each item most likely have a different hash value. Hashing comes in 2 different flavors:

- Closed Addressing: Each item has a fixed bucket in the table and each bucket can contain several items in the form of a linked list.
- Open Addressing: Each item could end up in a different bucket in the table and each bucket contains at most one item.

## 20.2 Resizing Problem

Resizing is to allocate a new bigger bucket array and redistribute the existing items using the updated hash function which uses the new size. For closed addressing, we require resizing when the items in the individual buckets increases over a certain threshold which causes a linear search on the bucket list for the Hash operations. There are 2 different criteria for resizing:

- Global threshold: When the size of X number of buckets increases over a certain threshold. (i.e X = 1/4)
- Bucket threshold: When the size of any bucket increases over the threshold value.

The new size of the bucket array is generally twice the original size in order to amortize the cost of copying the existing items in the new bucket array.

There are 3 different techniques of resizing: coarse grained locking, fine grained locking and lock free resizing.

### 20.2.1 Coarse Grained Locking

The main idea of Coarse grained locking is very simple, there is one lock for the entire bucket array. Acquire that lock and start the resizing process which is to create a new bucket array and copy over all the existing elements in the new array using the updated hash function. Coarse grained lock resizing is very simple and easy to implement but it has the issue of sequential bottleneck. Only one thread can operate at any time which causes performance issues.

### 20.2.2 Fine Grained Locking

In fine grained lock resizing, there is a separate lock associated with each bucket which has to be acquired before performing any operation (i.e. insert) on that bucket. So while resizing, the first step is to acquire all the locks in ascending order and make sure that the table reference didn't change between the resize decision and the lock acquisition. After acquiring the locks, allocate the new super sized table and start transferring the existing elements into the new table. As shown in **??**, we take individual locks on each bucket, create a new bigger bucket array, copy over the existing elements and then map any new elements to the new array.

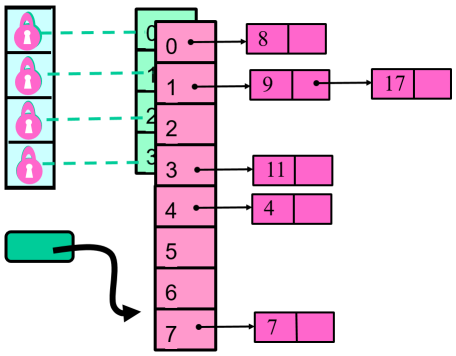


Figure 20.1: Fine Grained Locking

*Striped Locks:* Instead of allocating new locks for the new array, we use the existing locks. Since the size of the new array is double the size of original array, we use the each existing lock to cover one more bucket in the new array based on the symmetry.

*Read Locks:* We leverage the insight that most common operations is search on the hash table and so we keep 2 different kinds of locks: read lock and write lock. So multiple threads can take the read lock and start searching on the hash bucket.

### 20.2.3 Lock Free Resizing

The idea of resizing is to remove the existing element and put it at a new location in the new array. These are 2 different operations but CAS(CompareAndSwap) only allows us to do one operation so we either need

a double-compare-and-swap operation or a new idea for lock free resizing. The idea is to keep the items fixed in an ordered linked list and move the buckets. All the items are reachable from the top bucket but the additional buckets are shortcuts that allow us to achieve constant time retrieval.
We keep all the elements in a linked list with special order. The order will enable us to easily point new buckets in the linked list. We use the Recursive Split Ordering meaning that we recursively divide the existing bucket into two based on the least significant bit of the item. The ordering is based on the least significant bits. Starting from the least significant bit, start ordering and until all the elements are ordered. An example is given in **??** where each element is first sorted on the least significant bit and then on the second last significant bit. So if you have, two keys: a and b, a precedes b if and only if the bit reversed (least significant i bits) value of a is smaller than the bit reversed value of b as shown in **??**.

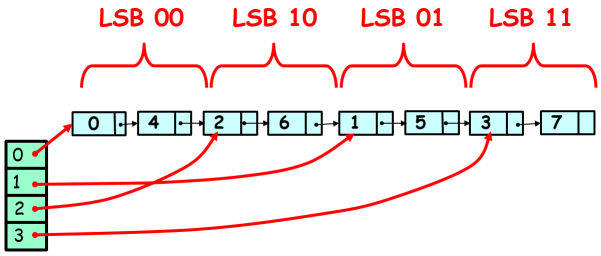


Figure 20.2: Special Ordering of items

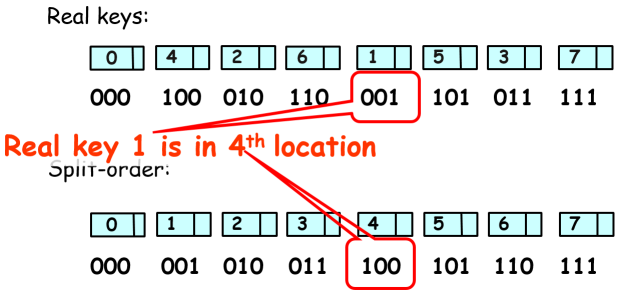


Figure 20.3: Reverse Ordering

Each bucket is mapped according to the reverse order of the bits of the first element in the linked list found by traversing from an existing bucket. But this causes an issue when removing a node having both the bucket pointer and the previous node pointer in the linked list using a single CAS operation. In order to cater for this problem, we use Sentinel Node for each bucket and initialize them whenever we need to point a bucket. So first create the Sentinel node and point it to the real node in the list. After that adjust the previous pointer to point to this sentinel node and finally add the bucket pointer to the sentinel.

## 20.3 Closed Hashing

### 20.3.1 Linear Probing

In open addressing we keep the bucket array which hold the actual items instead of pointing to a new list. Whenever an item x needs to added, we apply the hash function on the item and try to put the item at H(x)

index. If that index is already occupied then we move forward from that index until we find an empty spot to allocate x. For the Contain(x) operation, we start from H(x) and then search linearly until we find x or there is an empty bucket.
**Advantages:**

- Good Locality which means less Cache misses

**Disadvantages:**

- As the number of total items ratio to bucket size increases (M/N) more cache misses because we maybe searching in several unrelated buckets
- Clustering effect of keys into neighboring buckets

### 20.3.2 Cuckoo Hashing

Cuckoo hashing leverages the **power of 2** concept and creates 2 bucket arrays instead of one and keep two different hash functions, one for each array. For Insert(x) operation, we take the H1(x) and look at the resulting index in first array. If it's empty then just place x there else if there is an element y at H1(x) then take H2(x) and look at the resulting position in the second array. If it is empty then place x there else evict y, place x at H1(x) and recursively insert y at H2(y) and so on. The same idea is used for searching purposes.
**Advantages:**

- No clustering issue

**Disadvantages:**

- 2 tables
- The hash functions could be complex
- As M/N increases, the relocation cycle increases

### 20.3.3 Hopscotch Hashing

In Hopscotch hashing we have a single array and a simple hash function but we define neighborhood of original bucket. While adding an item, probe linearly to find an open slot and move the empty slot via sequence of displacements into the hop-range of H(x). For Contain(x) operation, search in at most H buckets (the hop range) based on the hop-info bitmap.
**Advantages:**

- Good locality and cache behavior
- Good performance as table density increases
- Optimize the common operation (Contains) at the expense of Insert

**Disadvantages:**

- Does not work well in all scenarios like Cuckoo Hashing

## 20.4 Trasactional Memory

Unitl now we have seen 2 different mechanisms to achieve mutual exclusion i.e. Lock based and lock free. Transactional memory is another concept used in this regard.

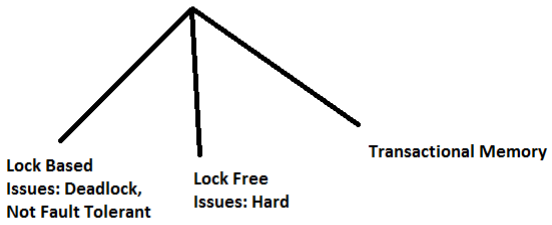


Figure 20.4: Different Concepts for Mutual Exclusion

The concept of Transactional memory is derived from Databases where we have similar issues like Concurrency and Failures. Jim Gray pioneered the concept of transactions for databases. The main construct of a Transaction is that we have *begin.transaction* at the start followed by several reads, writes and abort operations and then finally *end.transaction* which is also called commit transaction. Transactions in databases provide 4 guarantees which are commonly known as ACID:

- Atomicity : All or nothing
- Consistency : transaction will bring the database from one valid state to another
- Isolation : concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
- Durability : persistence even in the events of power loss or error

## References

[1] Vijay K Garg, Introduction to Multicore Computing

[2] Maurice Herlihy, Nir Shavit, Hashing and Natural Parallism, Companion slides for The Art of Multiprocessor Programming