

Problem

In many problems, it is natural to use Hash table as their data structures. How can the hash table be efficiently accessed among multiple units of execution (UEs)?

Context

Hash table is used when storing the key-value pairs. Given a key, the hash function maps it to the index of the table entry where it stores the value. This data structure is especially useful when the number of values to be stored is large and the keys are some objects other than small integers that cannot be used directly as index like strings. For example, suppose we want to store a telephone book where tons of pairs of a name and its associated telephone number are listed. The names are keys and telephone numbers are values as illustrated in Figure 1. Having values in plain sequential list, even if it can fit into the available memory size, would cost prohibitively large amount of the time for the lookup as it would have to inspect all the values in the sequential list. On the contrary, hash table enables fast lookup by using a hash function resulting in the constant overhead for finding the entry index.

Hash table are commonly used in all kinds of in-memory tables. They are used to implement associative arrays like symbol tables, caches like URL caching, data base index, and so on.

When a hash table is shared by multiple concurrent units of executions, like other shared data structures, it has to be accessed safely so that data should not get corrupted.

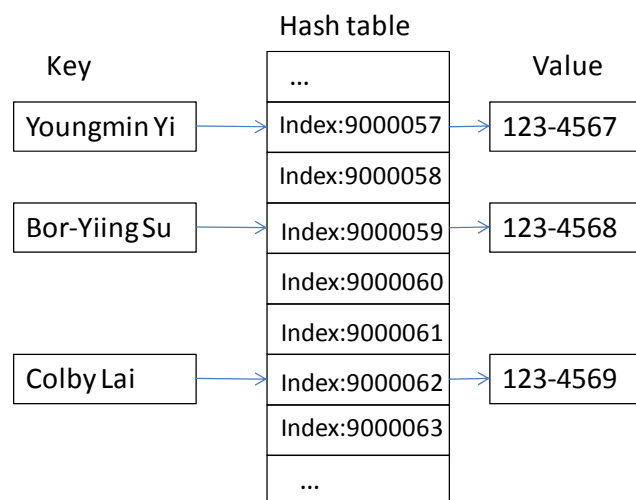


Figure 1. A hash table for a telephone book

A shared hash table on distributed memory can be constructed from multiple local hash tables in distributed memory by mapping a global index that has been obtained from a global hash function into a processor ID and the index for the local hash table. The scheme inherently ensures the exclusive access to the hash table because once the global index has been decoded to a processor ID and the local index then the processor makes a request to the appropriate processor and if there are multiple requests for the same processor ID and local index, the processor that owns the entry will serialize the requests.

If a global hash table is implemented on shared memory where every processor can directly access each entry in the table, the safe access should be enforced using locks.

Forces

- Shared hash table on distributed memory without locks vs. shared hash table on shared memory with locks

Distributed memory based shared hash table does not require any locking but it requires messages communications between processors. On the contrary, shared memory based shared hash table does not require message communication between processors but the locking is essential.

- Coarse-grain locks vs. fine-grain locks

One easy way of ensuring the safe access to hash table on shared memory is to have one single lock for the table. This would require minimal memory overhead but the single lock would be a source of high contention and would not be scalable. On the contrary, one can have one lock per table entry allowing for the concurrent accesses to the table as long as they access the different entries. However, as the number of locks increases, the memory overhead for the lock gets larger and overhead in managing locks in an operating system would become larger.

Solution

Depending on the number of keys, the number of units of executions, and the how the next keys to look up is determined, you would want to build a shared hash table in one of the following ways.

1. Single global hash table
2. Multiple local hash tables
3. Hybrid approach of 1. and 2.

1. **Single global hash table** is used when the next keys to look up is determined in a pretty random fashion. In such a case, a single global shared hash table that all the threads can read and update concurrently will result in better performance. The advantage of the single global hash table is that its implementation is very similar to the conventional hash table implementation in sequential machine except that it has to ensure the atomic operation of the accesses in order to prevent the data corruption.

Depending on the number of the units of executions, small number of locks (or even a single lock to the entire hash table) would suffice, especially if the ratio of the subsequent computation time after retrieving the value from the hash table is large to the hash table lookup time. As there are only a few locks to manage, there is less possibility of dead lock, which in general can occur frequently when implementing locks for shared data. However, only one or a few locks could restrict the available concurrency as the number of keys or the UEs in the system increases, becoming a single source of high contention.

On the other extreme, you would associate as many locks as the number of entries in hash tables. This will support as much concurrency as the system has by distributing the contention over all the entries in the hash table. However, maintaining a lock may not be cheap. It involves overhead in terms of memory size as it requires at least some bits to indicate the current status of the entry: whether the associated lock is currently occupied or available. Also, it involves run-time overhead as acquiring a lock and releasing a lock may invoke operation system calls which have some overhead. For such a reason, a user-level library using atomic instructions not invoking an operating system call is preferred to pthread API (pthread_mutex_lock) in terms of the performance. Therefore, it is always desirable not to use locks when it is possible as unnecessary locks will degrade the performance.

As a compromise of these two cases, you would want to associate one lock per multiple entries. Finding the right number of entries for efficient locking will be depend on the number of units of executions and the number of table entries.

Regarding to how to maintain and implement a lock, one can find a related pattern in **Mutual Exclusion Pattern** in Concurrent Execution Pattern and **Shared Data Pattern** in Implementation Strategy Pattern.

2. **Multiple local hash tables** approach constructs a logically global view of shared hash table from multiple local hash tables. In this approach, each UE has its own private hash table and interacts with the other UEs to read or update the hash entries. The advantage of this approach over single global hash table one is that it does not require explicit locking and there is no need for single contiguous large memory space for the hash table. In case of growing hash table design where if a certain threshold is reached (load factor) the hash table is dynamically resized so that it can still maintain fast lookup, all the UEs are blocked from any access to the shared hash table while the resizing is completed. In multiple local hash table approach, even if one local hash table is resizing, UEs can access the other hash tables that are not resizing. However, the disadvantage of this approach is that UEs cannot directly access another UE's local hash table and it has to access only via message passing or interprocess communication. This message passing or interprocess communication is implemented based on signal mechanism where a UE sends an request signal to the other UE that has the value to look up and the UE will be interrupted when receiving the signal. The concurrent requests are implicitly ensured to have consistent access with signal mechanism. Signals or messages are kept in the queue of the receiver and gets serialized. As a related pattern for the message passing or signals, there are **Message Passing Pattern** in Concurrent Execution Pattern and **Event-based, implicit invocation Pattern** in Structural Pattern.

One important design issue in this approach is how to map an index of a logically global shared

hash table into a pair of the UE ID and the index of local private hash table. Assigning UE ID from the global index can be static or dynamic. Static mapping requires ownership of each key and which UE owns which keys is determined a priori. The values of the keys that a UE owns are stored in its private hash table. The advantage of this solution is that it can have good cache locality as we can control the ownership of the keys in such a way that the keys accessed by a UE reside in the private hash table of that UE. On the contrary, it may be difficult to maintain a good load balance for the application that shows dynamic behavior.

Another way of assigning UE ID from global index is dynamic mapping, where there is no ownership of keys but the global hash function dynamically determines which UE should store the key. Since the dynamic mapping would consider good load balancing as its first-priority objective, this would fail in maintaining good cache locality compared to static mapping approach.

A related pattern for mapping global index of shared array to the local index of local arrays is **Distributed Array Pattern** in Implementation Strategy Pattern.

3. **Hybrid approach** is to mix the two approaches explained above. First, there are multiple local hash tables each of which is owned by a process, and in turn the local tables are shared by the threads in the same process. This approach aims to achieve the advantages of each approach explained: supporting more parallelism efficiently by having small number of locks and less message passing or inter process communication overhead.

Invariants

In case of multiple local hash table approach, hash function should cover all index space of each processor's local hash tables.

Also, the hash function should return a single index given the same key.

Examples

The code shown below illustrates a lock-free hash table implementation of Azul System's Highly Scalable Java Utility [1]. Instead of OS level lock, it uses user level library that uses atomic operation like CAS (Compare-And-Swap) instruction. Two library calls are denoted as CAS_key() and CAS_val() in the example. The code put() writes a pair of a key and value to the table.

First it obtains the hash or index of the shared table using hashing function, hashCode(). Note that mod (%) operation is a good hash function but mod operation is relatively expensive operation. So it replace the mod operation by AND (&) operation inside the subsequent loop. Then, it checks if there is an existing key in the entry with idx. If there is no key in that entry, it tries to insert a key using CAS_key(). If there is already a key in the entry but if it is the same key

then it considers that the key has been inserted. If there is already a key in the entry and it is a different key then, it means it encountered a collision. In this example, it employs linear probing policy [2].

After it inserted a key, it tries to insert the new value, newval by using CAS_val().

```
HashMap::put(key, newval) {
    idx = hash = key.hashCode();
    while( true ) {
        idx &= (size-1);          // & operation is efficient than %
        k = get_key(idx);
        if( k == null && CAS_key(idx,null,key) )
            break;              // Key inserted
        h = get_hash(idx);      // get memoized hash
        if( k == key || (h == hash && key.equals(k)) )
            break;              // Key already there
        idx++;                  // reprobe
    }
    oldval = get_val(idx)
    if( CAS_val(idx,oldval,newval) ) {
        ...                      // Value inserted
    }
}
```

Known Uses

- java.util.concurrent.HashMap : uses 16-way locks by default
- Azul Systems' Lock-free Hash table in Highly Scalable Utilities: uses CAS(Compare-And-Swap) like atomic update instruction[1]

Related Patterns

- Structural Patterns
 - Event-based, Implicit Invocation
- Implementation Strategy Patterns
 - Shared Data
 - Distributed Array
- Concurrent Execution Patterns
 - Mutual Exclusion
 - Message Passing

Reference

- [1] Cliff Click, "Lock-free Hash table", 2007 JavaOne Conference
- [2] T. Cormen et al. "Introduction to Algorithms", McGraw Hill

Author

Youngmin Yi