Challenge 3
The World's Simplest Lock-Free Hash Set

VerifyThis at ETAPS 2022
Organizers: Marie Farrell, Peter Lammich
Steering Committee: Marieke Huisman, Rosemary Monahan,
Peter Müller, Mattias Ulbrich

2–3 April 2022, TU Munich, Germany

Disclaimer: the programs may contain bugs. If you find any, fix them and mention the fix in your submission!

How to submit solutions: send an email to `verifythis@googlegroups.com` with your solution in attachment. Remember to clearly identify yourself, stating your team's name and its members.

Problem Description

This challenge is inspired by Jeff Preshing's blog entry: The World's Simplest Lock-Free Hash Table.

This simple hash-set has a fixed capacity, only supports insert and membership query operations, and uses linear probing for collision handling. However, it is thread-safe and implemented lock-free. At the core, the insert operation uses a compare-and-swap (CAS) operation to find a free spot for adding the key to be inserted. An optimization replaces expensive CAS operations by cheaper load operations when the table fills up.

The hashtable works with a key type K. It has a special value `key_invalid :: K` that is used as placeholder for empty slots, and cannot be used as key. Moreover, there is a function `get_hash(size_t,K) :: size_t`, such that `get_hash(`n`,`k`)` returns a hash-code for k, in the range $[0, n)$.

The compare and swap operation that we use returns the value stored in target after the operation (whether swapped or not). You are free to replace it with whatever similar operation your tool supports:

```
T compare_and_swap(T &target, T oldv, T newv) {
  T result;
  atomic {
    if (target == oldv) target=newv;
    result=target;
  }
  return result;
}
```

```
typedef hset = array<K>

hset empty(size_t n) {
  hset t = new K[n];
  for (size_t i = 0; i < n; ++i) t[i]=key_invalid;
  return t;
}

// Assumes k ≠ key_invalid
// returns true if key was inserted, false if table is full
bool insert(K k, hset t) {
  size_t n = t.length;
  size_t i0 = get_hash(n,k);
  size_t i = i0;

  do {
    { // Optimization: probe for potentially free spot
      key kk = atomic_load(t[i]);
      if (kk == k) return true; // Key already in table
      if (kk ≠ key_invalid) { // Spot taken, try next index
        i = (i+1) mod n; continue;
      }
    }

    // Maybe i is still free when we try to put our key there
    key k' = compare_and_swap(t[i],key_invalid,k);
    if (k' == k) return true; // We (or someone else) stored our key
    i = (i+1) mod n; // Someone else interfered with us, try next index
  } while (i ≠ i0); // Stop if we went one full round

  return false; // Table is full
}

// Assumes k ≠ key_invalid
bool member(hset t, key k) {
  size_t n = t.length;
  size_t i0 = get_hash(n,k);
  size_t i = i0;

  do {
    key k' = atomic_load(t[i]);
    if (k' == k) return true; // found the key
    if (k' == key_invalid) return false; // found empty entry. Key not in.
    i = (i+1) mod n;
  } while (i ≠ i0);
  return false; // Table full, our key is not in
}
```

Tasks

If your tool does not support concurrency, and you also cannot model concurrency on a more abstract level, implement and verify the sequential version of this hash-set as last resort.

Implementation task. Implement the above hash-table. Use whatever atomic operations your tool supports, but try to stay lock-free. If you have to use locks, use a fine-grained lock around `compare_and_swap`, rather than locking the whole table for the duration of an `insert`/`member` operation.

Verification tasks. Verify the following properties:

1. `empty(`n`)` creates an empty set with capacity n
2. `member(`k`)` $==$ `true`, if `insert(`k`)` has been executed before (and returned true)
3. `member(`k`)` $==$ `false`, if no `insert(`k`)` that returned true can have been executed
4. termination
5. every key is contained in the table at most once
6. if insert returns false, the table is full

You are free to re-phrase properties 2 and 3, but make sure you can prove things like:

```
insert(1) || insert(2) || insert(3);
{ assert(member(1) and !member(42)) } || insert(4) || insert(5);
```