ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2017 Prof. R. Wattenhofer

Distributed Systems Part II

Solution to Exercise Sheet 10

Quiz

1 Quiz

a) By acquiring locks in the order the affected nodes occur in the list – which should remain fixed at all times.

b) Hash functions ...

- ... map variable size inputs to constant size hashes.
- ... ideally are fast. (perhaps unless used in cryptography)
- ... ideally have a uniform distribution of outputs, independent of the distribution of inputs. I.e., collisions should be rare!
- ... are hard to invert. (at least in cryptographic contexts)

c) This is acceptable if the integer keys are distributed uniformly. (Of course, if the keys' remainders mod x are distributed uniformly, this is also sufficient.) Note that the size of the resulting hashes is only $\log_2 x$ bits!

d) One could use a binary search tree or any other search data structure within each bucket. This causes some overhead, but may save a lot of space compared to growing the hash map, if many buckets are filled far less. However, this scenario should only rarely occur given a good hash function with a uniform hash distribution.

e) Attach a linked list instead of a single item to each key. Allowing the hash map itself to hold multiple items with the same key can cause severely imbalanced buckets, no matter how good the hash function is, if a lots of values with the same key are inserted.

Advanced

2 Old Exam Question: Fine-Grained Locking

a) Coarse grained locking locks the entire tree at once for each operation, regardless of the location the change occurs. This can be achieved by always locking the same location for each operation. We could for example always lock the root by issuing LOCK(1) and UNLOCK(1).

b)

Algorithm 1 Insert value

```
1:  .....
2:  i = 1
3:  LOCK(i) .....
4:  while A[i] != null do
5:      .....
6:      if A[i] > value then
7:          .....
8:          exchange A[i] and value
9:          .....
10:     end if
11:     .....
12:     next = smallestChild(i)
13:     LOCK(next); UNLOCK(i) .....
14:     i = next
15:     .....
16: end while
17: .....
18: A[i] = value
19: UNLOCK(i).....
```

Algorithm 2 Remove smallest value

```
1:  LOCK(1) .....
2:  ret = A[1]
3:  .....
4:  A[1] = ∞
5:  .....
6:  i = 1
7:  .....
8:  while A[i] != null do
9:      .....
10:     next = smallestChild(i)
11:     LOCK(next) .....
12:     exchange A[i] and A[next]
13:     UNLOCK(i) .....
14:     i = next
15:     .....
16: end while
17: .....
18: A[i] = null // Mark as not used
19: UNLOCK(i).....
20: return ret
```

c) Both functions acquire locks in the same ordering: locks closer to the root are taken before the locks on descendants. This means that there is a global lock order, hence there cannot be a deadlock as seen in the lecture. Alternatively one might observe that descending in a tree is exactly the same as walking down a linked list, hence the same analysis from the lecture applies.

d) We could use optimistic locking to lock only at the subtree that is going to be modified. We would walk down from the root to the location the current value will be inserted at, lock that location and then check the consistency by walking down once more. We'd then start hand-over-hand locking at that location to propagate the changes down to the leafs. This has the advantage of not locking the root in the case of an insert. Notice that this would still lock the root in the case of a remove since the first modified location is the root.