A report for lock-free data structures and algorithms

Synchronization is a regular problem in current programming. Although mutual exclusion(adding locks) gives us a solution to this problem. But it will waste resources sometimes, which causes delays.

Lock-free data structures use lock-free algorithms on the abstract data types.

One of the algorithms that handles these structures is atomic synchronization primitives, such as compare and set.

Here is the pseudo code.

```
COMPARE& SWAP(a : address, old, new : word)
        returns boolean
        BEGIN ATOMIC
        if *a != old
                *a = new
                return TRUE
        else
        return FALSE
        END ATOMIC
```

Next, we will use linked list(queue) as an example to illustrate how to construct a lock-free queue.

```
Enqueue(x)
        q = new record
        *q.value = x
        *q.next = NULL
         repeat
                p = tail
                succ = Compare&Swap(*p.next, NULL, q)
                if succ != TRUE
                        Compare&Swap(tail ; p; *p.next)
        until succ = TRUE
        Compare&Swap(tail ; p; q)
end
```

```
Dequeue()
        repeat
                p = head
                if *p.next = NULL
                        error queue empty
        until Compare&Swap(head ; p; *p.next)
        return *p.*next.value
end
```

However, when we do the following operations,

When the first node is dequeued, and is recycled, and we place the enqueue element in it.

After then, we do the COMPARE& SWAP operation, it will be successful, but the data structures will be corrupted.

This is called ABA problem.

To solve this problem, we can use a double-word version of COMPARE& SWAP or STORE-CONDITIONAL primitive.

But here we present a saferead operation,

```
SafeRead(q)
        loop:
        p = *q.next
        if p = NULL then
                return p
                Fetch&Add(*p.refct; 1)
                if p = *q.next then
                        return p
                else
                Release(p)
        goto loop
end
```

That’s it.

Difficulties and Challenges

1, Lock-free algorithms are not efficient for general data structures, even compared with spinlocks, we must design specific algorithms for concrete data structures.

2, We need to verify programs' correctness by formal methods.

3, we must test the efficiencies after implementations.

Some research hotspot

The software implementation is very slow compared to locks, some studies used hardware to implement these algorithms, while it makes the hardware very complex.

Some researchers were studying transactional memory which is based on lock-free primitives.

Reference

[1] Valois J D. Lock-free data structures[D]. Rensselaer Polytechnic Institute, 1995.

[2] Valois J D. Lock-free linked lists using compare-and-swap[C]//Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. 1995: 214-222.

[3] Barnes G. A method for implementing lock-free shared-data structures[C]//Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures. 1993: 261-270.

[4] Valois J D. Implementing lock-free queues[C]//Proceedings of the seventh international conference on Parallel and Distributed Computing Systems. 1994: 64-69.

[5] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 2 (May 1993), 289–300. DOI:https://doi.org/10.1145/173682.165164