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

Lecture 19: November 1

Lecturer: Vijay Garg *Scribe: Zijiang Yang*

19.1 Wait Free Consensus Hierarchy

Consensus number of a shared object is the maximum number of processes that can use that object to solve consensus problem. Different objects have different consensus numbers, as shown in the following table.

Object Type	Consensus Number
Read&Write	1
Queue(Stack), Test&Set, Swap, Get&Increment, (2,1)Objects	2
...	...
Check&Set	Infinity

Based on the hierarchy of consensus objects, we can always use objects with higher consensus number to build objects with lower consensus numbers.

19.1.1 Read-Modify-Write Objects

A Read-Modify-Write object is the abstraction of an general object that has the following structures:

```
private int value;
public int synchronized getAndModify(int x) {
    int prev = value;
    value = f(value, x);
    return prev;
}
```

Each Read-Modify-Write object has an associated value and an atomic operation. The operation will set a new value based on function f and return the old value.

There are a lot of consensus objects which can be viewed as Read-Modify-Write object. Each has its own f function definition:

- Test&Set: f(v, x) = 1
- Swap: f(v, x) = x
- Get&Increment: f(v, x) = v + 1

Read-Modify-Write objects can be characterized by the property of funtion f:

- A Read-Modify-Write object is commute if
  $f_i f_j = f_j f_i$
  where f_i is the funtion applied by process i

- A Read-Modify-Write object is overwrite if
  $f_i(f_j(x)) = f_i(x)$ *for any i, j*
  where f_i is the funtion applied by process i

19.1.2 Consensus Number of Read-Modify-Write Object

A **trivial** Read-Modify-Write object is an Read-Modify-Write object whose f(v, x) = v. It can be viewed as a simple read operation.

Theorem: Any non-trivial Read-Modify-Write object with either commuting or overwriting property has consensus number equal to two.

Proof:

1. Firstly we need to prove that consensus number for RMW object is at least 2.

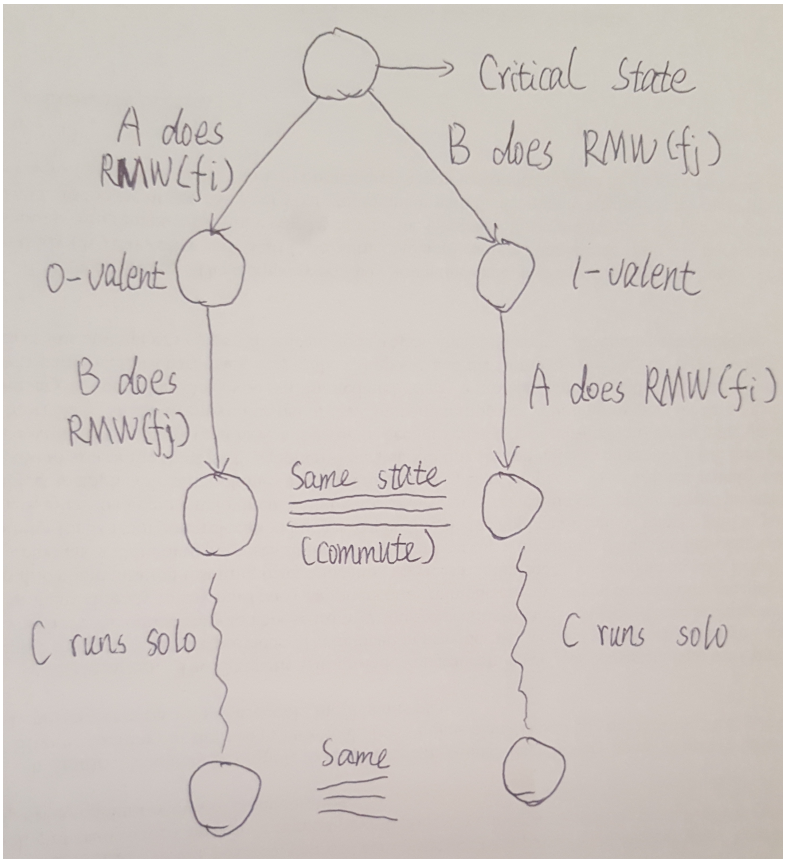
   It is easy to come up with an algorithm to solve consensus problem on 2 processes using the non-triviality property:

```
int[2] A, RMW r (init null)
Pi: write proposol on A[i]
     value = r.getAndModify(i)
     if(value == null) {
         I win
     } else {
         other process win
     }
```

   This algorithm works because the RMW object is non-trivial. Thus there exist v such that f(v, x) != v.

2. Then we need to prove that the consensus number of RMW object is at most 2.

   We can show that there is not a protocol that solves consensus for 3 processes using RMW objects.



In order to reach consensus, a protocol has to reach critical state. Suppose at critical sate A does RMW(fi) and reach to 0-valent state; B does RMW(fj) and reach to 1-valent state. Then at the 0-valent state B does RMW(fj); at the 1-valent state A does RMW(fi). After that, both path reaches the same state. Then C runs solo and will decide the same value from both state. But it contradicts the assumption that one state is 0-valent state and the other state is 1-valent state. Thus it is impossible to reach a critical state for RMW object on 3 processes.

Thus in conclusion, non-trivial Read-Modify-Write object with either commuting or overwriting property has consensus number equal to two.

19.1.3 2-Writes-1-Read Object

We already know that we cannot use atomic read-write register to solve consensus problem. However we can solve this problem if we are allowed to write 2 locations atomically. A 2-Writes-1-Read object (*(2, 1) object*) is an object which can write 2 locations and read 1 location atomically.

Following is a protocol that can solve 2 process consensus problem using (2, 1) object.

```
int[3] A, init null
P0: A[0] = a, A[1] = a (atomically)
P1: A[1] = b, A[2] = b (atomically)
whichever writes first wins
```

There are 4 possible configurations:

- [a, a, null] =>p0 wins
- [a, b, b] =>p0 wins
- [null, b, b] =>p1 wins
- [a, a, b] =>p1 wins

In this way we can solve consensus on 2 processes.

19.2 Universal Constructions

Deterministic Object: given a state and an operation, the resulting state is deterministic.

What we want to do is given some objects of consensus number m, construct a deterministic object with consensus number m' <= m.

An easy way would be to use a consensus object to decide who wins. However, in general consensus objects are not reusable.

The correct way would be use a linked list to record the invocation histories, and maintain a consensus object inside each node. When multiple processes wants to invoke methods, use consensus object of the head node to decide which thread wins.

The linked list is organized as follows:

```
Initial state -> apply method1 -> apply method2 -> apply method3
```

The invocation log is equivalent to the current state of the object.

Sudo code can be found in https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter7-consensus

19.3 Hashing

Hashing is a technology to map an object into an index. A HashTable uses hashing to decide where to store an item. The time complexity of HashTable is as follows:

Operation	**Time Complexity**
Insert	Average O(1)
Search	Average O(1)
Remove	Average O(1)

However, generally HashTable may has collisions which may damage the performance. We want to make collisions as infrequent as possible. There are different strategies in case of collision:

- **Chaining:** Store a linked list in each bucket. Add items into the linked list.
- **Open Addressing:**
  - Linear Probing: if the bucket h(i) is occupied, insert item into the next available bucket. This approach may cause clustering effect.
  - Quadratic Probing: if the bucket h(i) is occupied, then try $h(i)+1$. If still occupied, try $h(i)+2^2$, then $h(i)+3^2$, then $h(i)+4^2$ ...

We want to parallelize the chaining HashTable using lock. A naive solution is to hold a lock for all buckets and acquire the lock for all the operations. A better solution is to hold a lock for every bucket, thus we can parallelize the operations on different buckets. However this approach will waste a lot of spaces. An improvement would be hold a lock for every k buckets. Thus we can save a lot of space while maintain the parallel ability.