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

Lecture 21: November 8

Lecturer: Vijay Garg *Scribe: Shengwei Wang*

21.1 Transaction Memory

21.1.1 Transaction

For transcations:

- Atomicity
- Consistency
- Isolation
- Durability

For TM, we are looking at ACI.

Generally, a transaction can be described as:

```
begin transaction
        read(x);
        write(z);
        read(y);
        ...
end transaction
```

There are several read/write operations.

21.1.2 Serializability

The execution is equivalent to some series execution of all transactions.

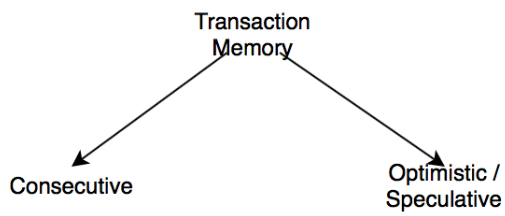
Databases use locks, but not necessary for programmers.

```
atomic {
        x = 2;
        while()
        ...
}
```

Former structure :

```
synchronized (this) {
                    ...
          }
```

Difference : By using synchronized key word, we are actually acquiring locks.



When we use locks, we may have problems:

- Deadlocks
- Composability Problem

Example for Composability Problem :

BLOCKING QUEUE q1, q2

To do : get item from either of the queue if not empty.

Problem : when we do q1.deq(), and q1 is empty, then blocking.

21.1.3 Opacity

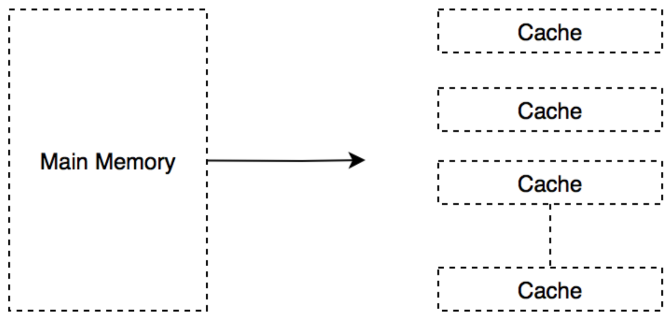
An execution satisfies opacity if all the committed transactions and the read prefix of the aborted transactions appear as if they have been executed serially in agreement with real-time occurrence order.

```
Var x, y init 0

atomic {
          x++;
          y++;
}

atomic {    //Zombie Transaction
          if (x != y) {
                    while (true);
          }
}
```

21.2 Software Transaction Memory

There are multiple algorithms.
Deuce STM (Software Transaction Memory)

In hardware TM:



we store data from memory to cache. When abort, we just throw away data in cache.

TL2

21.2.1 Key Notion

- fetch&add
- Clock
- Local copies
- Dates of local copies

Put timestamp for each transaction. If the birthday of transaction 1 if 75, and the birthday of transaction 2 is 80, then transaction 1 is completely earlier than transaction 2.

- read set of transaction : lrst
- write set of transaction : lwst
- local copy : lc (xx)

21.2.2 operations

Begin of transcaion

```
begint()
lrst <- {}; lwst <- {};
birthday(T) = clock + 1; //atomic
```

Read opreation: X.read()

```
if there is a local copy of xx
then
        return lc(xx).value
else
        lc(xx) <- copy of xx from shared memory
        if lc(xx).date < birthday(T)
        then
                lrst(T) <- lrst(T) U {x};
                return lc(xx).value
        else
                abort
```

Write operation: X.write(v)

```
if there is no local copy then allocate space for lc(xx)
loc(xx).value <- v;
lwst <- lwst U {x}
```

Commit operation : Commit()

```
lock all x in lrst, lwst
for all xx <- lrst do
        if xx.date >= birthdate (T)
        then
                release lock;
                abort
write_date <- clock.fetch&add();
for all xx <- lwst do
        xx <- (lc(xx), write_date);

unlock;
return commit;
```

The algorithm is slow because of making local copies When abort, start over again.

21.3 Stream Programming

It's like functional programming.
Parallelism is easy if objects don't change.

- lamda function : nameless function
- streams
- parallel stream

lamda Funtion :

```
//using lamda function as the second parameter
Arrays.sort(data, (a,b) -> (b - a));
```

Stream : Think in high level