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

Lecture 21: November 8

Lecturer: Vijay Garg *Scribe: Edward Clinton, TEC553*

21.1 Agenda

This lecture deals with transaction memory. We will focus on 4 characteristics of transactions:

- A - Atomicity
- C - Consistency
- I - Isolation
- D - Durability

Transaction memory: we will look at Atomicity, Consistency and Isolation, not worried about Durability

21.2 Introduction

A transaction is a sequence of events that appears indivisible and instantaneous to an outside observer and has four properties:

- Atomicity: All constituent actions in a transaction complete successfully, or none of these actions appear to start executing.
- Consistency: A transaction can modify the state of the world, i.e. data in a database or memory. These changes should leave this state consistent. If the transaction completes successfully, then the system will be in a valid state. If an error occurs then any change will be rolled back. For example, when transfering money from account A to account B, the system is consistent if the total of all accounts is constant. If an error occurs after removing money from account A but before adding it to account B, the system is no longer consistent since the total would have changed (money disappeared). By rolling back the removal from account A, the system will be back in a consistent state.
- Isolation: Each transaction produces a correct result, regardless of which other transaction are executing concurrently. (i.e. the system state would be obtained if transactions were executed serially)
- Durability: Once a transaction commits, its result is permanent (stored in a durable media such as disk)

Transactions is a language construct that discharges programmers from the management of synchronization issues. (synchronization constructs such as locks are hidden from the programmer).

21.3 Serializability

The correctness condition in transactions is called serializability. Serializability states that the concurrent execution of transactions is equivalent to some serial execution of all transactions. In other words, it appears that one transaction finishes before the next transaction starts. For example, if T_1, T_2 and T_3 execute concurrently, the net result should be equivalent as if these transactions ran sequentialy such as: $T_2->T_1->T_3$ or $T_3->T_2->T_1$ etc.. (there are n! possibilities).

21.4 Transaction construct

In the database world, the programmer does not have to worry about using locks for synchronization. Synchronization is all taken care of by the underlying system. To enable transactions we add one additional construct to import in a programming language as follows:

```
atomic{
//do whatever you want
          x:=2;
     while(){
          //do stuff
     };
}
```

This atomic construct assures that the block is executed as one single step such that a process running concurrently can only observe a state that is either before or after the atomic block executes.

At first this construct may seem equivalent to the java synchronized block. The difference is that the synchronized block locks the object implementing the synchronized method such that operations in the synchronized blocks cannot be interleaved. However in transactions, its OK to interleave operations inside an atomic block as long as the effect is the same as if each block executed separately.

Another difference is that transactions use speculative (optimistic) execution to execute multiple transactions in parallel (hoping that nothing bad happens). If contention is low then there is a good chance that nothing bad will happen. If there is a conflict between two atomic blocks the system will resolve it using aborts and roll-backs.

Some issues of locks vs transactions:

- **Deadlocks** , There is no notion of deadlocks in TM. If there is a conflict between some transactions (atomic blocks). The system will resolve it using aborts. Aborts are not available in Java synchronization.
- **Composability problem**. Given two data structures Q1 and Q2 we perform following operation atomically:

  ```
  atomic{
  x = q1.deq();
  q2.enq(x);
  }
  ```

  Invariant: No process shall find a state where x is not in q1 nor q2 (i.e. X has to be in q1 or q2 and not in both). In TM, this is achieved trivially by using atomic block. If using locks, this problem is non-trivial. Locks cannot be easily composed. In Queue implementations based on monitors, each method acquires the lock internally, so it is essentially impossible to combine two method calls this way.

21.5 Opacity

Opacity is a correctness condition for TMs. 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. Opacity can be viewed as an extension of serializability with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states.

For example:

```
x = 1;
y = 0;

atomic{
x++;
y++;
}

atomic{
z = z/(x - y) //<- can raise a divide by zero exception
//zombie transaction, read values that are impossible (ex x == y)
//if all transactions occured serially

}
```

Opacity guarantees that x != y since x != y is true in all the consistent global states of the multiprocess program (no transaction can read values from an inconsistent global state).

21.6 Implementation and Algorithm

There are multiple implementations (ex Deuce STM , Hardware Transaction memory (HDM) for small transactions). Current state of the art : no algorithms are currently used because of their high overhead (higher than locking , or lock free). But there is optimism that transactions will become useful someday.

In HDM, the idea is that since in a normal machine all updates are stored in the cache and not in main memory (cache block is written-back to main memory before getting evicted, only true for write-back cache) the abortion of a transaction can be implemented by not writing the cache block(s) back to main memory.

We will focus on an STM (software transaction memory) system called TL2 (transactional locking 2) due to D. Dice, O. Shalev, and N. Shavit (2006). This STM system statisfies the opacity consistency condition. The TL2 algorithm is posted on canvas.

The TL2 algorithms uses locks (lock usage is minimized). Locks are used when a transaction is about to be committed to check if the states are consistent and to determine if the transaction needs to be committed or aborted. TL2 also uses atomic instructions.

Properties and control variables of TL2:

- CLOCK: clock is a fetch and add register initialized to 0 and is used as a logical clock to measure the progress of the system. It represents the number of transactions that have been committed so far.
- All registers are MWMR atomic registers. Each register has two fields: *value*, which contains the value of the register, *date* which contains the date of its last update. A lock is associated with each register.
- Each process has two local variables *lrs(T)* and *lws(T)* where T is the transaction currently executing. *lrs(T)* (local read set) contains the names of all registers read by T until now. *lws(T)* (local write set) contains all the registers written up to now.

We will look at an implementation of a *begin* , *end* transaction and how to read and write objects.

```
begin()
{
lst(T) <- {} ; rst(T) <- {} ;
birthdate(T) <- CLOCK + 1 ;
return;
}
```

When reading an object we first read it into a local copy

```
X.readT()
{
if(there is a local copy lc(XX) of XX){
        then return(lc(XX).value)
    }
    else lc(XX) <- copy of XX read from the shared memory;
    if(lc(XX).date < bitrhdate(T)){
        then lrst <- lrst U {X}; return(lc(XX).value)
        }
        else return(abort)
}
```

If the date of the register is not less than the birth date of the transaction then we must abort the transaction to guarantee opacity. A transaction cannot read from the future.

```
X.write(V)
{
if(there is no local copy of XX) then allocate local space lc(XX) for a copy
lc(XX).value <- v ; lwstt <- lwst U {X};
return(ok) //write never aborts
}
```

If a transaction reaches its last statement without having been previously aborted, it invokes the **commit** operation. That operation decides the fate of T by returning commit or abort.

```
try_to_commit()
{
lock all the objects in (lrst U lwst);
for each X in lrst do
{
        //the date of XX is read from the shared memory
    if XX.date >= birthdate(T) then release all the locks; return abort
}
write_date <– CLOCK.fetch&add();
for each X in lwst do XX <– (lc(XX).value, write_date)
release all the locks; return commit


}
```

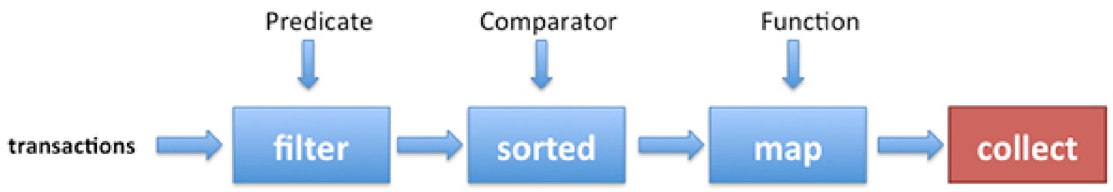
If the read validation succeeds (as explained above), the process has to compute the new date that has to be associated with all the writes issued by T. All written local copies are then stored to shared memory.

This algorithm is slow due to all the copy overhead and local copy write back. When using locks, there is isn't a copy overhead.

21.7 Intro to Java Streams

Stream is a JAVA abstraction introduced to JAVA SE 8. A stream is a sequence of aggregate operations as presented below.

Figure 21.1:



```
List<Integer> transactionsIds =
     transactions.stream()
                    .filter(t –> t.getType() == Transaction.GROCERY)
                    .sorted(comparing(Transaction::getValue).reversed())
                    .map(Transaction::getId)
                    .collect(toList());
```

Figure 21.1 illustrates the Java SE 8 code. First, we obtain a stream from the list of transactions (the data) using the stream() method available on List. Next, several operations (filter, sorted, map, collect) are chained together to form a pipeline, which can be seen as forming a query on the data. Stream operations return streams themselves

To parallelize the code, we can use parallelStream() instead of Stream().

References

[R12] M. Raynal, Concurrent Programming: Algorithms, Principles and foundations (2012), pp. 277-289.