Cpts 483/580 – April 20, 2010

Non-blocking synchronization: – General topic

– : suspension of one or more threads does not stop other threads from progressing

– contrast locks: if a thread holding a lock is suspended – no other thread needing that lock can progress

Several defined levels of NB synch:

Most restrictive is

(1) Wait-free: every operation requires bounded number of operations

⇒ every thread makes progress

Very complicated to implement — requires threads to help each other to word completion.

② Lock-free :

Bound on # of steps between system-wide progress

allows : some threads do not make progress

again : threads help each other, but now with goal of enabling their own progress.

③ Obstruction-free :

- Progress guaranteed if only one transaction executes.
- If multiple transaction execute concurrently they may all abort.
- Relatively easy to implement

and claim: we can introduce a contention manager to reduce the amount of concurreny if too many aborts occur.

wait free ⇒ lock-free ⇒ obstruction free

STM

Last time we talked about TAS instr and LL/SC instruction pair.
atomic check Then act.

CAS compare and swap

```
bool CAS (int *a, int exp, int new) {
    if *a == expected {
        *a = new;
        return true
    } else return false.
}
```

(exp: expected; the { } block: atomic)

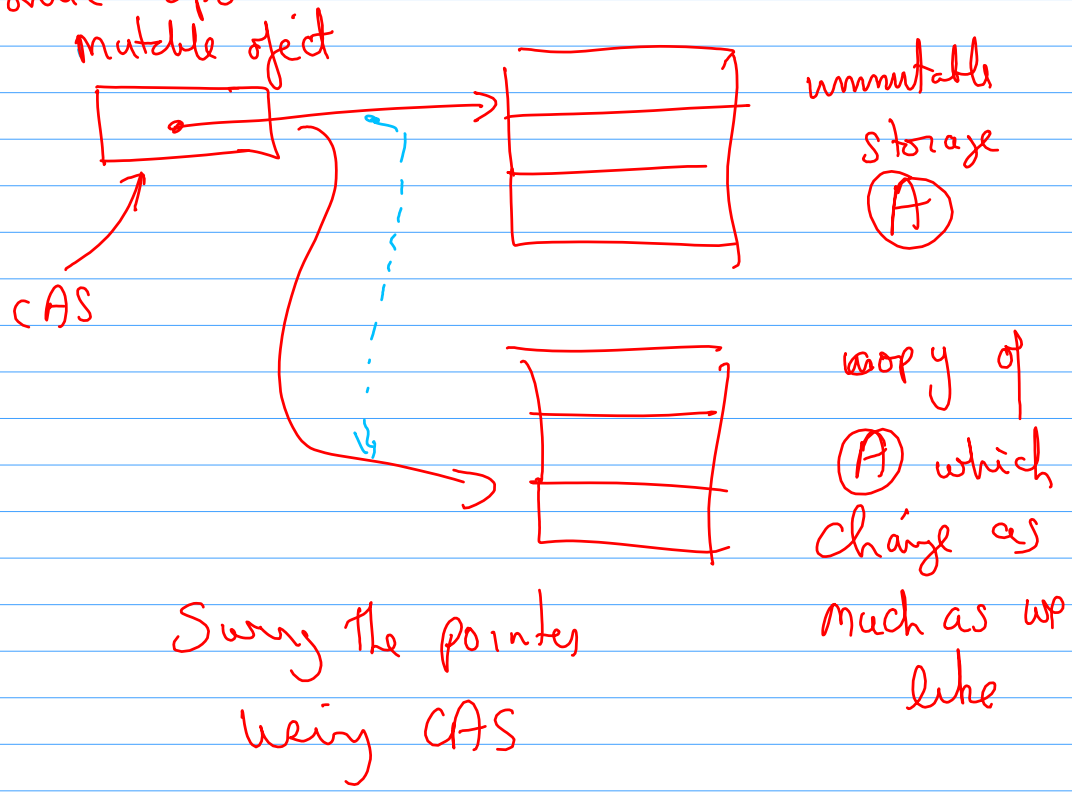
Basis for mechanisms of STM.

Up 'til early 2000s – instrs like these were used to implement specialized data structures trying to be WF, LF, OF.

Then MH et al said: Can we implement the TM ideas using instrs like these.

Yes.

Fundamental idea is to go from a one-word atomic update to a multi-word atomic update:



Not good enough: only change one object atomically

So people worked out ways to put all the interesting data for a transaction in one place.

DSTM - says how to update multiple objects atomically.

Implemented for Java by

1) extending Thread with a current transaction field.

2) Wrapping objects we want to update atomically in objects called TMObjects.

Wrapping – reference via TMObjects

An ordinary object. ⟹ Counter counter = new Counter(0);

TMObject myCounter = new TMObject(counter);

I should now forget I ever knew about counter.

but instead ...

Counter localCounter =
(Counter) myCounter.open(WRITE ~~READ~~)

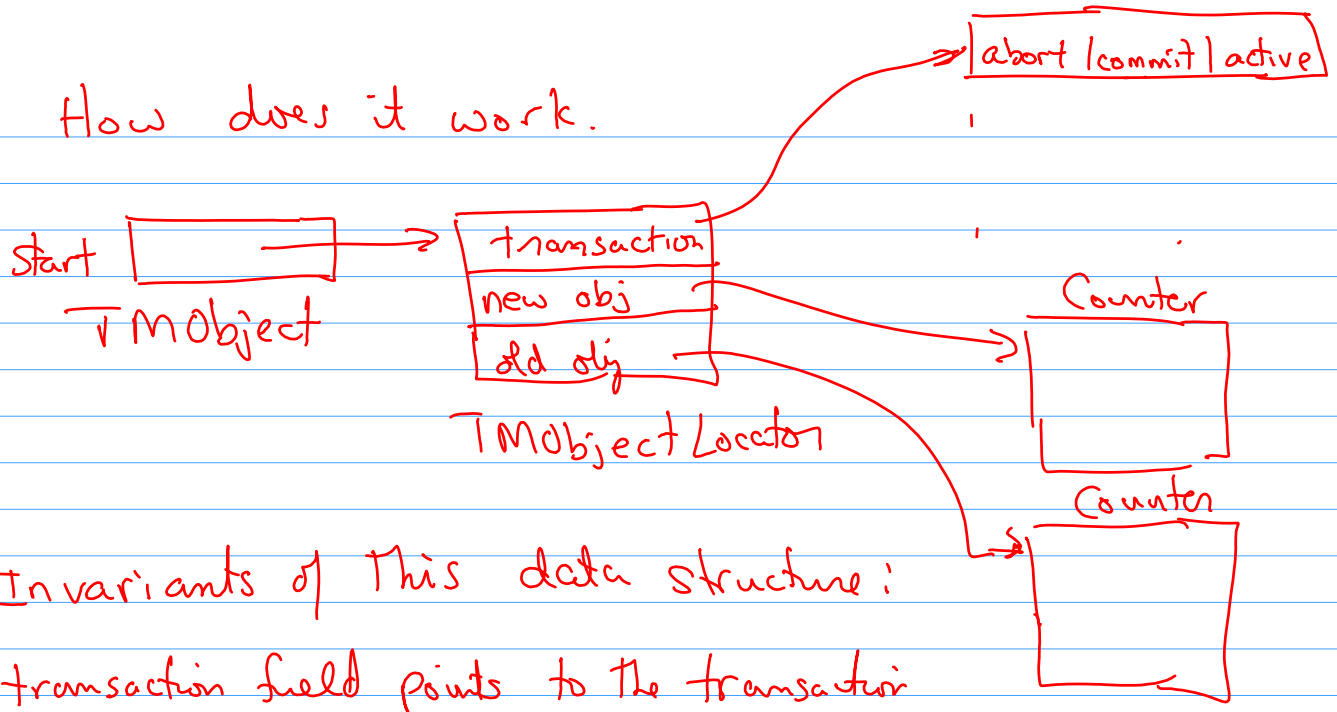
now localCounter refers to a copy of the storage of counter.

localCounter.inc() ~ changes content of the local copy

Thread.currentThread().commitTransaction().

atomically puts local counter and any other changed objects back into public view.

How does it work.



Invariants of this data structure:

transaction field points to the transaction that most recently opened the TM Object for WRITE

"current version" is defined to be
old obj if the transaction state is abort
new obj if the transaction state is commit
if active - old object

Cool thing: a CAS on the transaction state from active to commit atomically changes all the objects that reference that transaction from old to new

To make it practical:

- want to not copy the object when opening for read.
- two threads that both open for writing conflict — one has to abort (the older)
- if T1 aborts T2, then T2 doesn't find out about it while executing "normal code"
- Contention manager idea: each thread has a CM that it consults whenever it wants to abort another thread.
  - CM may tell it to abort itself instead; wait a while.

Performance

implemented basic CM that always allows aborts

and CM that does back off waiting.

Tests on perf. of set implementation using linked lists and red-black trees.