Transactional Memory

R. Guerraoui, EPFL
Locking is "history"

Lock-freedom is "difficult"
Wanted

A synchronisation abstraction that is simple, robust and efficient
Transactions
Back to the sequential level

- accessing object 1;
- accessing object 2;
Back to the sequential level

atomic {
    accessing object 1;
    accessing object 2;
}
Semantics (serialisability)

Every transaction appears to execute at an indivisible point in time (linearizability of transactions)
The TM Topic has been a VERY HOT topic

- Sun/Oracle, Intel, AMD, IBM, MSR
- Fortress (Sun); X10 (IBM); Chapel (Cray)
The TM API
(a simple view)

- `begin()` returns `ok`
- `read()` returns a value or `abort`
- `write()` returns an `ok` or `abort`
- `commit()` returns `ok` or `abort`
- `abort()` returns `ok`
Two-phase locking

To *write* or *read* O, T requires a *lock* on O; T *waits* if some T’ acquired a *lock* on O

At the end, T *releases* all its locks
Two-phase locking (more details)

Every object $O$, with state $s(O)$ (a \textit{register}), is protected by a lock $l(O)$ (a \textit{c&s})

Every transaction has local variables $wSet$ and $wLog$

Initially: $l(O) = \text{unlocked}$, $wSet = wLog = \emptyset$
Two-phase locking

Upon $\text{op} = \text{read()}$ or $\text{write}(v)$ on object $O$

if $O \notin \text{wSet}$ then

\[ \text{wait until unlocked} = l(O).c&s(\text{unlocked}, \text{locked}) \]

$\text{wSet} = \text{wSet} \cup O$

$\text{wLog} = \text{wLog} \cup S(O).\text{read()}$

if $\text{op} = \text{read()}$ then return $S(O).\text{read()}$

$S(O).\text{write}(v)$

return ok
Two-phase locking (cont’d)

Upon `commit()`
cleanup()
return ok

Upon `abort()`
rollback()
cleanup()
return ok
Two-phase locking (cont’d)

Upon *rollback*(())
for all \(O \in wSet\) do \(S(O).write(wLog(O))\)
\(wLog = \emptyset\)

Upon *cleanup*(())
for all \(O \in wSet\) do \(l(O).write(unlocked)\)
\(wSet = \emptyset\)
Why two phases? (what if?)

To **write** or **read** O, T requires a **lock** on O; T **waits** if some T’ acquired a **lock** on O

T **releases** the lock on O when T is done with O
Why two phases?

T1

read(0)

O1

write(1)

O2

T2

read(0)

O2

write(1)

O1
Two-phase locking (read-write lock)

- To **write** O, T requires a **write-lock** on O; T **waits** if some T’ acquired a **lock** on O

- To **read** O, T requires a **read-lock** on O; T **waits** if some T’ acquired a **write-lock** on O

- Before committing, T **releases** all its locks
Two-phase locking
- better dead than wait -

To **write** O, T requires a **write-lock** on O; T **aborts** if some T’ acquired a **lock** on O

To **read** O, T requires a **read-lock** on O; T **aborts** if some T’ acquired a **write-lock** on O

Before committing, T releases all its locks

A transaction that aborts restarts again
Two-phase locking - better kill than wait -

- To **write** O, T requires a **write-lock on** O; T **aborts** $T'$ if some $T'$ acquired a **lock** on O

- To **read** O, T requires a **read-lock** on O; T **aborts** $T'$ if some $T'$ acquired a **write-lock** on O

Before committing, T releases all its locks

A transaction that is aborted restarts again
Two-phase locking - better kill than wait -

- To write O, T requires a write-lock on O; T aborts T' if some T' acquired a lock on O

- To read O, T requires a read-lock on O; T waits if some T' acquired a write-lock on O

- Before committing, T releases all its locks
- A transaction that is aborted restarts again
Visible Read
(SXM, RSTM, TLRW)

Write is mega killer: to write an object, a transaction aborts any live one which has read or written the object.

Visible but not so careful read: when a transaction reads an object, it says so.
Visible Read

A visible read invalidates cache lines

For read-dominated workloads, this means a lot of traffic on the bus between processors
  - This reduces the throughput
  - Not a big deal with single-CPU, but with many core machines
Two-phase locking with invisible reads

To **write** O, T requires a **write-lock on** O; T **waits** if some T’ acquired a **write-lock** on O

To **read** O, T checks if **all objects read remain valid** - else T **aborts**

Before committing, T checks if all objects read remain valid and releases all its locks
Every object $O$, with state $s(O)$ (register), is protected by a lock $l(O)$ (c&s).

Every transaction maintains, besides $wSet$ and $wLog$:

A local variable $rset(O)$ for every object.
Invisible reads

Upon \texttt{write(v)} on object \texttt{O}

if \texttt{O} \notin \texttt{wSet} then

\hspace{1em} wait until unlocked= \texttt{l(O).c&s(unlocked,locked)}

\hspace{1em} \texttt{wSet} = \texttt{wSet U O}

\hspace{1em} \texttt{wLog} = \texttt{wLog U S(O).read()}

\hspace{1em} (*,ts) = \texttt{S(O).read()}

\hspace{1em} \texttt{S(O).write(v,ts)}

return \texttt{ok}
Invisible reads

Upon `read()` on object O

\[(v, ts) = S(O).read()\]

if \( O \in wSet \) then return \( v \)

if \( l(O) = \text{locked} \) or not validate() then abort()

if \( rset(O) = 0 \) then \( rset(O) = ts \)

return \( v \)
Invisible reads

Upon `validate()`
for all O s.t rset(O) > 0 do
  (v,ts) = S(O).read()
  if ts ≠ rset(O) or
     (O ∉ wset and l(O) = locked)
then return false
else return true
Invisible reads

Upon \textit{commit()} if not validate() then abort()
for all \( O \in \text{wset} \) do
\[
(v,ts) = S(O).\text{read}()
\]
S(O).\text{write}(v,ts+1)
cleanup()
Invisible reads

Upon \textit{rollback}()
for all $O \in \text{wSet}$ do $S(O).write(wLog(O))$

$wLog = \emptyset$

Upon \textit{cleanup}()
for all $O \in \text{wset}$ do $l(O).write(\text{unlocked})$

$wset = \emptyset$

$rset(O) = 0$ for all $O$
**DSTM (SUN)**

- To **write** O, T requires a **write-lock on** O; T aborts T' if some T' acquired a **write-lock** on O.

- To **read** O, T checks if all objects read remain valid – else T **abort**.

- Before committing, T releases all its locks.
DSTM

*Killer write* (ownership)

*Careful read* (validation)
More efficient algorithm?

Apologizing versus asking permission

- Killer write
- Optimistic read
  - validity check only at commit time
Example

Invariant: $0 < x < y$

Initially: $x := 1; y := 2$
Division by zero

**T1:** \( x := x + 1 \; ; \; y := y + 1 \)

**T2:** \( z := \frac{1}{y - x} \)
Infinite loop

T1: x := 3; y := 6

T2: a := y; b := x;
    repeat b := b + 1 until a = b
Opacity

Serialization

Consistent memory view
Trade-off

The read is either visible or careful
Intuition

T1
- read()
- O1, O2, ..., On

T2
- write()
- I1, I2, ..., Im

commit

T1
- read()
- Ik
**Read invisibility**

The fact that the read is invisible means T1 cannot inform T2, which would in turn abort T1 if it accessed similar objects (SXM, RSTM).

NB. Another way out is the use of multiversions: T2 would not have written “on” T1.
Aborting is a fatality

T1

read()

commit

O1

write()  O2

abort

T2

write()  read()

O1  O2

commit
Conditional progress - obstruction-freedom -

A correct transaction that eventually does not encounter *contention* eventually commits

*Obstruction-freedom* seems reasonable and is indeed possible
DSTM

- To **write** O, T requires a **write-lock on** O (use C&S); T aborts T’ if some T’ acquired a **write-lock** on O (use C&S)

- To **read** O, T checks if all objects read remain valid - else abort (use C&S)

- Before committing, T releases all its locks (use C&S)
If a transaction $T$ wants to write an object $O$ owned by another transaction $T'$, $T$ calls a

*contention manager*

The contention manager can decide to wait, retry or abort $T'$
Contention managers

- **Aggressive**: always aborts the victim

- **Backoff**: wait for some time (exponential backoff) and then abort the victim

- **Karma**: priority = cumulative number of shared objects accessed – work estimate. Abort the victim when number of retries exceeds difference in priorities.

- **Polka**: Karma + backoff waiting
Greedy contention manager

- **State**
  - Priority (based on start time)
  - Waiting flag (set while waiting)

- **Wait** if other has
  - Higher priority AND not waiting

- **Abort** other if
  - Lower priority OR waiting
Aborting is a fatality

T1

\[
\begin{array}{c}
\text{read()} \\
\text{O1}
\end{array}
\]

\[\text{write()} \]

\[\text{O2}\]

T2

\[\begin{array}{c}
\text{write()} \\
\text{O1}
\end{array}\]

\[\begin{array}{c}
\text{read()} \\
\text{O2}
\end{array}\]

commit

abort
Concluding remarks

TM does not always replace locks: it hides them

Memory transactions look like db transactions but are different
The garbage-collection analogy

In the early times, the programmers had to take care of allocating and de-allocating memory.

Garbage collectors do it for you: they are now incorporated in Java and other languages.

Hardware support was initially expected, but now software solutions are very effective.