Transactional Memory

Michał Kapałka

Concurrent Algorithms 2009

Outline

1. Why?
2. What?
3. How?

Why?

Problem

Hypothesis: implementing wait-free (obstruction-free) atomic objects efficiently is difficult.

Note: universal construction is sometimes too expensive.

Example: see previous lectures…

Problem

Hypothesis 2: implementing scalable data structures using locks is also difficult.

Example: …

Problems with Locks

- implicit object-lock mapping

Problems with Locks

From the Linux kernel:

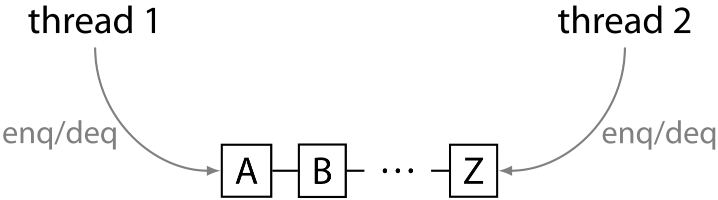
```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */
```

Problems with Locks

- implicit mapping
- lock contention
- deadlock
- lost wakeups

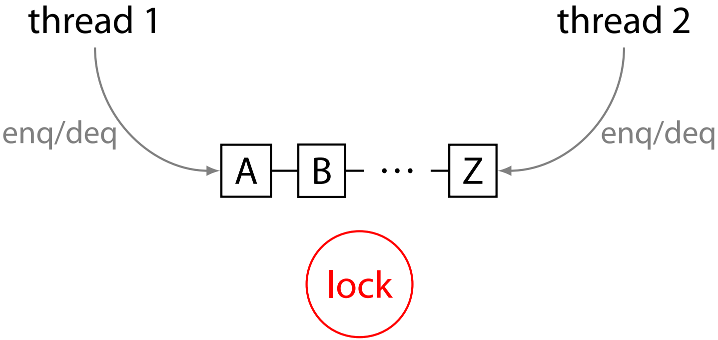
Sadistic Homework (of M. Herlihy)

Implement a double-ended queue:



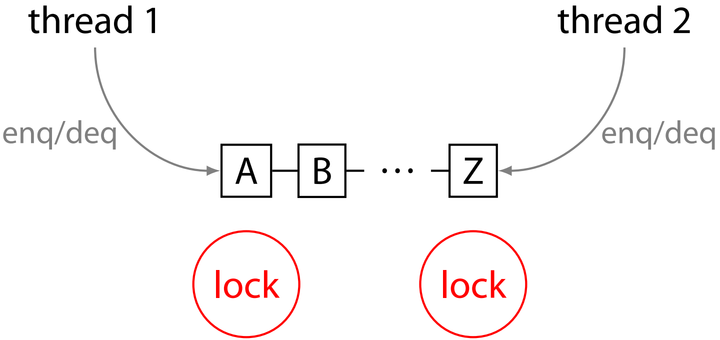
Sadistic Homework (of M. Herlihy)

Implement a double-ended queue:



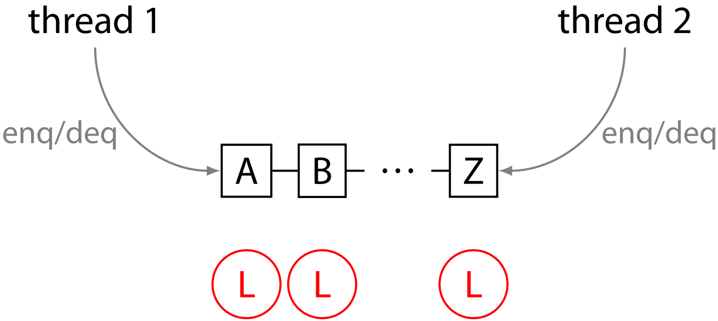
Sadistic Homework (of M. Herlihy)

Implement a double-ended queue:



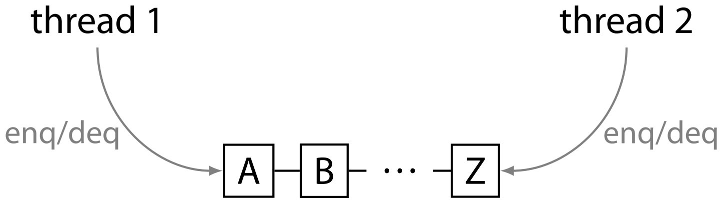
Sadistic Homework (of M. Herlihy)

Implement a double-ended queue:



Sadistic Homework (of M. Herlihy)

Implement a double-ended queue:



Solution: see [Michael & Scott, PODC'96]
Obstruction-free solution: see [Herlihy et al., ICDCS'03]

Problems with Locks

- implicit mapping
- lock contention
- deadlock
- lost wakeups
- no composability

Problems with Locks

```
synchronized(???) {
  val = obj.remove(key);
  obj.put(key, f(val));
}

synchronized(???) {
  val = obj1.remove(key);
  obj2.put(key, val);
}
```

Problems with Locks

- implicit mapping
- lock contention
- deadlock
- lost wakeups
- no composability
- priority inversion
- no robustness
- …

What?

```
atomic {
  val = obj1.remove(key);
  obj2.put(key, val);
}
```

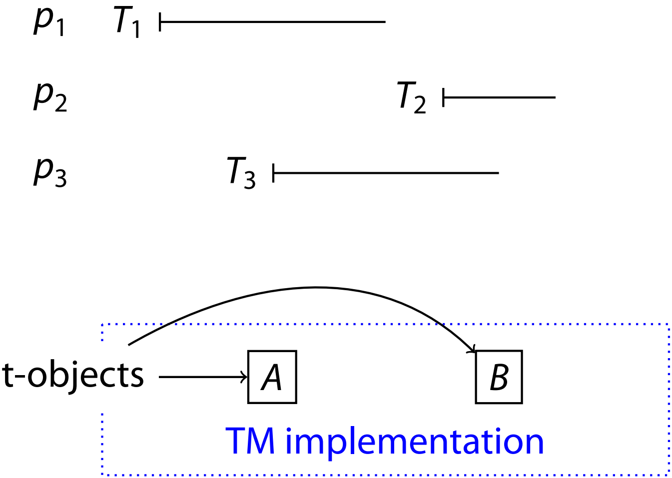
Make simple things easy

```
void enqueue(element) {
  atomic {
    Node newNode = new Node(element);
    newNode.next = head;
    head.prev = newNode;
    head = newNode;
  }
}
```

Make simple things easy

atomic blocks = transactions

Transactional Memory



p_1 T_1

p_2 T_2

p_3 T_3

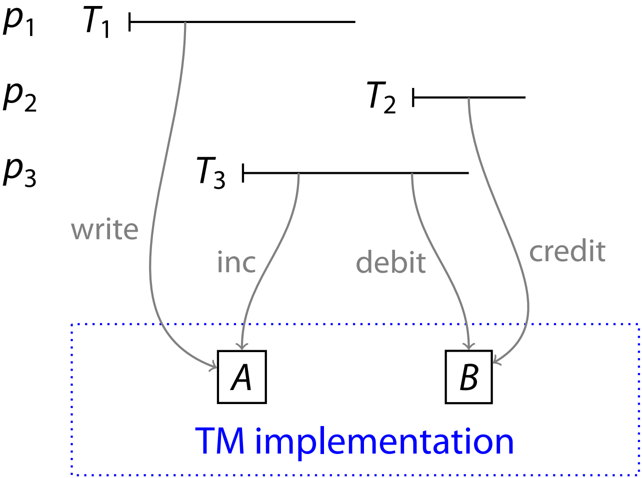
t-objects

A

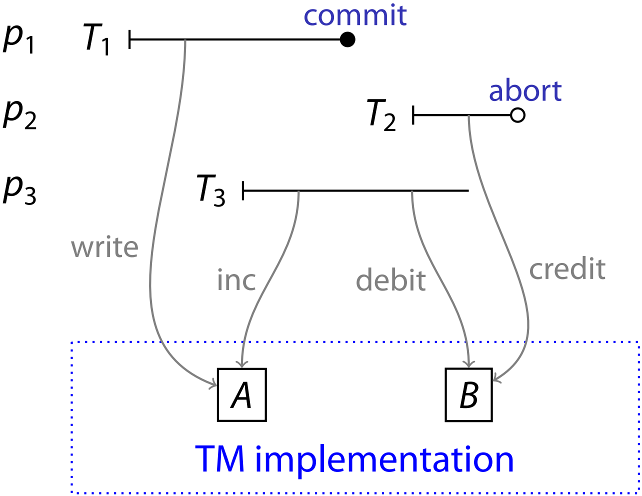
B

TM implementation

Transactional Memory



Transactional Memory



TM Implementations

C/C++ and Java compilers (Intel, IBM, Tanger, DeuceSTM)

Libraries (SwissTM, TinySTM, TL2, …)

Hardware (prototypes)

Model

TM = shared object with operations:

- *texec*($x.op_k$) – execute operation *op* on t-object x within transaction T_k; returns the value returned by *op*, or a special value A_k when T_k is aborted;
- *tryC*(T_k) – try to commit T_k; returns C_k (commit successful) or A_k (commit failed $\Rightarrow$ T_k aborted);
- *tryA*(T_k) – abort T_k; always returns A_k.

A TM object is wait-free,
but not atomic (no sequential spec)

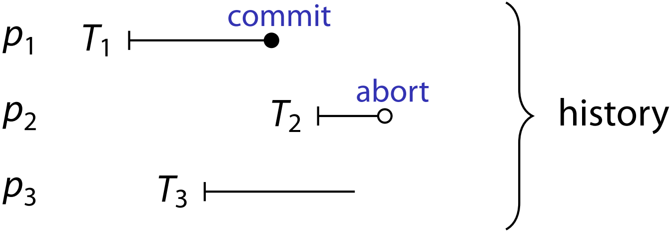
Model

- T-objects are inside the TM object;
  ⇒ can only be accessed via operation *texec*.
- When a process p_i executes an operation
  $texec(x.op_k)$, $tryC(T_k)$, or $tryA(T_k)$,
  we say that transaction T_k executes, respectively,
  $x.op_k$, *tryC*, and *tryA*.
- For simplicity of the lecture: only *read* and *write* operations (like registers).

Terminology

- T_k **starts** when it invokes its first operation.
- T_k **commits** when it receives C_k from *tryC*.
- T_k **aborts** when it receives A_k from any TM operation.
- T_k is **forceably aborted** when it receives A_k from operation *texec* or *tryC*.

Real-Time Order

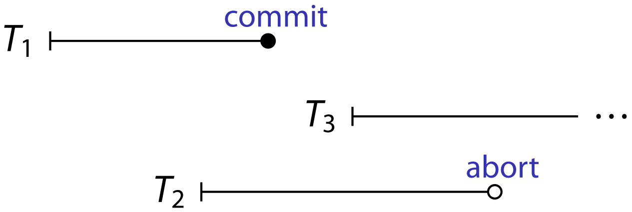


T_1 and T_3 are **concurrent**

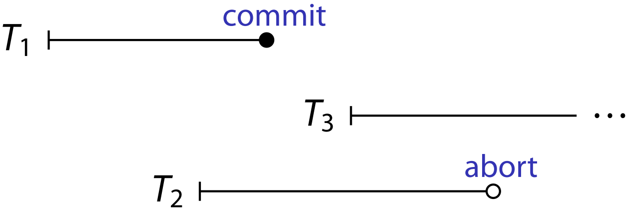
(T_2 and T_3 as well)

T_1 **precedes** T_2

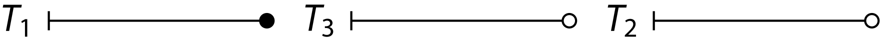
Safety of a TM



Safety of a TM



"looks like"



Opacity

Correctness (safety) of a TM = **opacity**; intuitively:

1. Every transaction appears as if it was executed instantaneously at some point during its lifespan (similar to atomicity / linearizability)
2. No transaction ever observes an inconsistent state of the system

Opacity is like strict serializability, but applied to all transactions, not only the committed ones.

How?

Bogus TM

```
upon $texec(x.op_k)$
    return $A_k$
end

upon $tryC(T_k)$
    return $A_k$
end

upon $tryA(T_k)$
    return $A_k$
end
```

correct (wait-free, ensures opacity), but useless…

⇒ need to specify **progress** properties

Progress property: when a transaction can be forceably aborted?

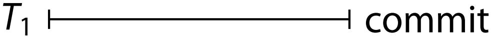
Examples Progress Properties

1. Perfect progressiveness – no transaction is ever forceably aborted.

Examples Progress Properties

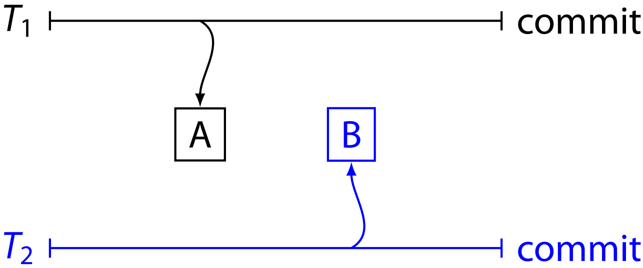
1. Perfect progressiveness – no transaction is ever forceably aborted.
2. Strong progressiveness – if a group of concurrent transactions conflicts on **at most** one t-object, then at least one of those transactions is not forceably aborted.

Strong Progressiveness – Example 1

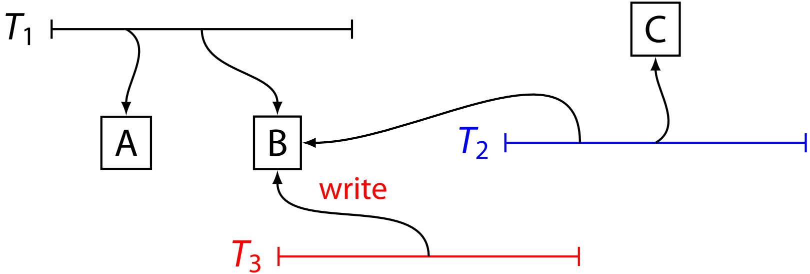


T_1 commit

Strong Progressiveness – Example 2



Strong Progressiveness – Example 3

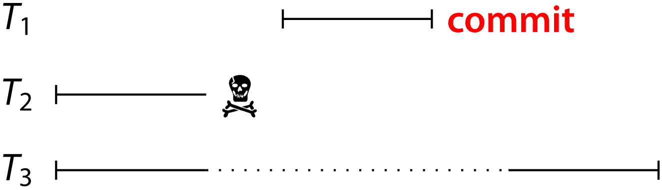


commit or commit or commit

Examples Progress Properties

1. Perfect progressiveness – no transaction is ever forceably aborted.
2. Strong progressiveness – if a group of concurrent transactions conflicts on **at most** one t-object, then at least one of those transactions is not forceably aborted.
3. TM obstruction-freedom – if a transaction T_k executes **alone** (i.e., with all other transactions suspended or crashed during the execution of T_k), then T_k is not forceably aborted.

TM Obstruction-Freedom

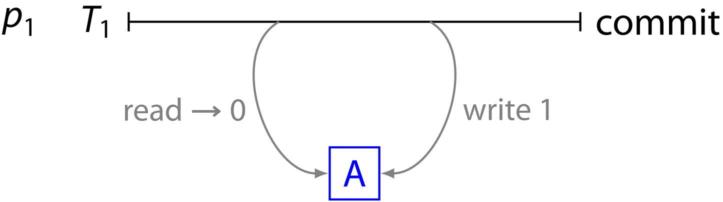


Impossibility

Theorem: There is no TM implementation that ensures perfect progressiveness in an asynchronous system in which processes can crash.

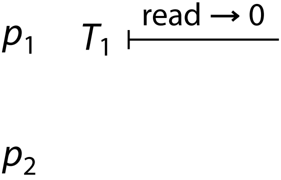
Proof sketch: …

Proof (Intuition)



```
atomic {
  v := A.read();
  A.write(v + 1);
}
```

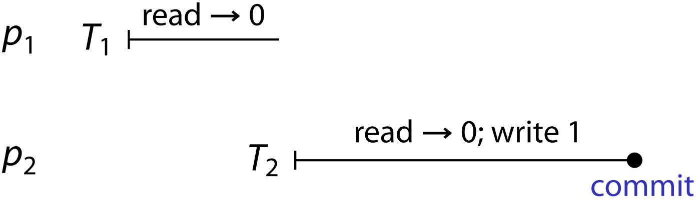
Proof (Intuition)



p_1 T_1 read → 0

p_2

Proof (Intuition)

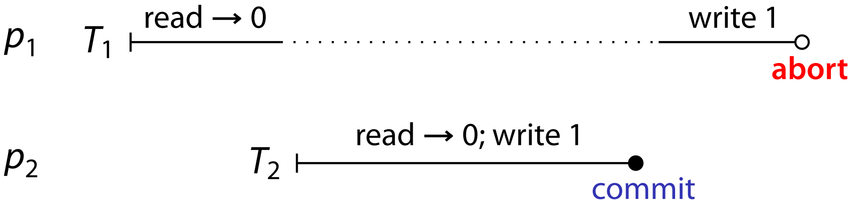


p_1 T_1 read → 0

p_2 T_2 read → 0; write 1

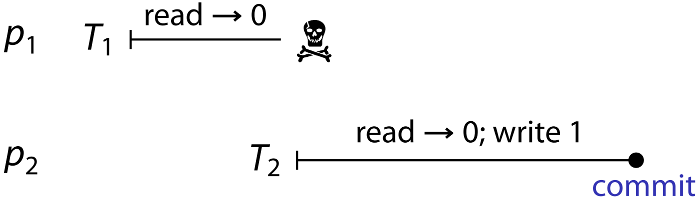
commit

Proof (Intuition)



p_2 cannot distinguish this execution from a one in which p_1 crashes just after T_1 reads 0 ⇒ T_2 cannot wait for T_1 and must eventually commit

Proof (Intuition)



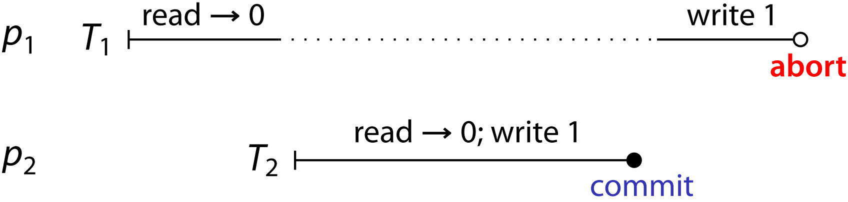
p_1 T_1 read → 0

p_2 T_2 read → 0; write 1

commit

p_2 cannot distinguish this execution from a one in which p_1 crashes just after T_1 reads 0 ⇒ T_2 cannot wait for T_1 and must eventually commit

Proof (Intuition)


p_1 T_1 read → 0 write 1 abort

p_2 T_2 read → 0; write 1 commit

If T_1 and T_2 both read 0, write 1 and commit, then opacity is violated

(one of them should read 1 and write 2,

since each increments the value of A)

Lock-based TM

Lock-Based TM – Simple Algorithm

Idea: use (strict) 2-phase locking (see databases)

Implement: t-objects $x_1, x_2, \ldots$

Every t-object x_m protected with a lock
(a C&S object $C[m]$)
State of x_m stored in register $S[m]$
(variables *wset* and *wlog* are process-local)

Initially: $C[1, \ldots]$ = unlocked, $wset = \emptyset$ at every process

```
upon $x_m.read_k$ or $x_m.write(v)_k$
    if $x_m \notin wset$ then
        if $C[m].C\&S(\text{unlocked}, \text{locked}) = \text{locked}$ then
            rollback
            return $A_k$
        end
        $wset := wset \cup \{x_m\}$
        $wlog[m] := S[m].read$
    end
    if $op = read$ then return $S[m].read$
    $S[m].write(v)$
    return ok
end
```

```
upon $tryC(T_k)$
    cleanup
    return $C_k$
end

upon $tryA(T_k)$
    rollback
    return $A_k$
end
```

```
procedure rollback
    for $x_m \in wset$ do $S[m].write(wlog[m])$
    cleanup
end

procedure cleanup
    for $x_m \in wset$ do $C[m].C\&S$(locked, unlocked)
    $wset := \varnothing$
end
```

Possible improvement: use read-write locks ⇒ single writer, multiple readers semantics

Even then a (big) problem: readers must **write** to memory ⇒ cache contention

Solution: **invisible reads**

Lock-Based TM with Invisible Reads

Uses: $C[1,...]$ – readable C&S objects,
$S[1,...]$ – registers
(other variables are process-local)

Initially: $C[1,...]$ = unlocked, $S[1,...] = (0,0)$,
$wset = \emptyset$, $rset[1,...] = \perp$

```
upon $x_m.write(v)_k$
    if $x_m \notin wset$ then
        if $C[m].C\&S(\text{unlocked}, \text{locked}) = \text{locked}$ then
            rollback
            return $A_k$
        end
        $wset := wset \cup \{x_m\}$
        $wlog[m] := S[m].read$
    end
    $(v', ts) := wlog[m]$
    $S[m].write(v, ts)$
    return ok
end
```

```
upon $x_m.read_k$
    $(v, ts) := S[m].read$
    if $x_m \in wset$ then return $v$
    if $C[m].read = \text{locked}$ or not $validate$ then
        $rollback$
        return $A_k$
    end
    if $rset[m] = \bot$ then $rset[m] = ts$
    return $v$
end
```

```
procedure validate
    for $m : rset[m] \neq \bot$ do
        $(v, ts) := S[m].read$
        if $ts \neq rset[m]$ or ($x_m \notin wset$ and
        $C[m].read = \text{locked}$) then return false
    end
    return true
end
```

```
upon $tryC(T_k)$
    if not $validate$ then
        $rollback$
        return $A_k$
    end
    for $x_m \in wset$ do
        $(v, ts) := S[m].read$
        $S[m].write(v, ts + 1)$
    end
    $cleanup$
    return $C_k$
end
```

```
upon $tryA(T_k)$
    rollback
    return $A_k$
end

procedure rollback
    for $x_m \in wset$ do $S[m].write(wlog[m])$
    cleanup
end

procedure cleanup
    for $x_m \in wset$ do $C[m].C\&S(\text{locked}, \text{unlocked})$
    $wset := \varnothing$
    for $m = 1, 2, \ldots$ do $rset[m] := \perp$
end
```

Obstruction-free TM

Obstruction-Free TM – Simple Algorithm

Idea: use a global **revocable** lock

Implements: t-objects $x_1, x_2, \ldots$

Uses: C – C&S object, F – fetch&increment object, $S[1, \ldots]$ – unbounded registers
(other variables are process-local)

Initially: $C = 1, F = 2, S[1] = (0, 0, \ldots), slot = \perp$

```
upon $x_m.read_k$ or $x_m.write(v)$
    if $slot = \bot$ then
        $slot := F.fetch\&increment$
        $current := C.read$
        $values = S[current].read$
        $S[slot].write(values)$
    end
    $values = S[slot].read$
    if $op = read$ then return $values[m]$
    $values[m] := v$
    $S[slot].write(values)$
    return ok
end
```

```
upon tryC($T_k$)
    s := C.C&S(current, slot)
    slot := $\bot$
    if s = current then return $C_k$
    else return $A_k$
end

upon tryA($T_k$)
    slot := $\bot$
    return $A_k$
end
```

Possible improvements:

- one C&S and one register per t-object (finer grained)
- use memory management (+ garbage collector) instead of infinite arrays

Practical examples: DSTM, NZTM

Transactions @ EFPL:

lpd.epfl.ch