Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        ... if (*) other.bar(); ...
    }
}
```

and two instances:

```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:

- threads A and B execute a.bar() and b.bar()
- a.bar() acquires the monitor of a
- b.bar() acquires the monitor of b
- A happens to execute other.bar()
- B happens to execute other.bar()
- → both block indefinitely

How can this situation be avoided?

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. **mutual exclusion**: processes require exclusive access
2. **wait for**: a process holds resources while waiting for more
3. **no preemption**: resources cannot be taken away form processes
4. **circular wait**: waiting processes form a cycle

The occurrence of deadlocks can be:

1. **ignored**: for the lack of better approaches, can be reasonable if deadlocks are rare
2. **detection**: check within OS for a cycle, requires ability to preempt
Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:
1. **mutual exclusion**: processes require exclusive access
2. **wait for**: a process holds resources while waiting for more
3. **no preemption**: resources cannot be taken away from processes
4. **circular wait**: waiting processes form a cycle

The occurrence of deadlocks can be:
- **ignored**: for the lack of better approaches, can be reasonable if deadlocks are rare
- **detection**: check within OS for a cycle, requires ability to preempt
- **prevention**: design programs to be deadlock-free

---

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be partially ordered.

**Definition (lock sets)**

Let $L_p$ denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at $p$, that is, the set of locks that may be in the “acquired” state at program point $p$.

---

~prevention~ is the only safe approach on standard operating systems
- can be achieve using lock-free algorithms
- but what about algorithms that require locking?
Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be partially ordered.

Definition (lock sets)

Let $L$ denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at $p$, that is, the set of locks that may be in the “acquired” state at program point $p$.

We require the transitive closure $\sigma^+$ of a relation $\sigma$:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \sigma^i \cup \{(x_1, x_3) \mid \exists x_2 \in X, (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i\}
\]

Each time a lock is acquired, we track the lock set at $p$:

Definition (lock order)

Define $\prec \subseteq L \times L$ such that $l \prec l'$ iff $l \in \lambda(p)$ and the statement at $p$ is of the form $\text{wait}(l')$ or $\text{monitor}_{\text{enter}}(l')$. Define the strict lock order $\prec^+$ as:

\[
\text{Definition (lock order)}
\]

\[
\text{Define } \prec \subseteq L \times L \text{ such that } l \prec l' \text{ iff } l \in \lambda(p) \text{ and the statement at } p \text{ is of the form } \text{wait}(l') \text{ or } \text{monitor}_{\text{enter}}(l'). \text{ Define the strict lock order } \prec^+.
\]

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be partially ordered.

Definition (lock sets)

Let $L$ denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at $p$, that is, the set of locks that may be in the “acquired” state at program point $p$.

We require the transitive closure $\sigma^+$ of a relation $\sigma$:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \sigma^i \cup \{(x_1, x_3) \mid \exists x_2 \in X, (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i\}
\]

Each time a lock is acquired, we track the lock set at $p$:

Definition (lock order)

Define $\prec \subseteq L \times L$ such that $l \prec l'$ iff $l \in \lambda(p)$ and the statement at $p$ is of the form $\text{wait}(l')$ or $\text{monitor}_{\text{enter}}(l')$. Define the strict lock order $\prec^+$.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.
Freedom of Deadlock

The following holds for a program with mutexes and monitors:

**Theorem (freedom of deadlock)**

If there exists no \( a \in L \) with \( a \prec a \) then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) at \( L_S \) and on monitors at \( L_M \) such that \( L = L_S \cup L_M \).

**Theorem (freedom of deadlock for monitors)**

If \( \beta a \in L_S, a \prec a \) and \( \beta b \in L_M, b \in L, a \not\prec b \land a \prec b \land b \prec a \) then the program is free of deadlocks.

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks \( L_S \) and summarized monitor locks \( L_M^S \subseteq L_M \)

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks \( L_S \) and summarized monitor locks \( L_M^S \subseteq L_M \)
- identify non-summary monitor locks \( L_M^N = L_M \setminus L_M^S \)
- sort locks into ascending order according to lock sets
- modify code that locks are only acquired in strictly ascending order
Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?
- identify mutex locks $L_s$ and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^w = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets
- modify code that locks are only acquired in strictly ascending order

⚠️ Ordering might be hard or impossible to find:
- determining which locks may be acquired at each program point is
  undecidable \( \rightarrow \) approximate lock set
- an array of locks: lock in increasing array index sequence
- if \( (x,P) \) exists where \( i' < i \) should be locked: release \( l \)
  acquire \( l' \), then acquire \( l \) again \( \rightarrow \) inefficient
- if a lock set contains a summarized lock $\bar{n}$ and $\bar{n}$ is to be acquired, we're stuck

---

Refining the Queue: Concurrent Access

Add a second lock \( s \rightarrow t \) to allow concurrent removal:

```c
int PopRight (DQueue* q, int val) {
    QNode* oldRightNode;
    wait(q->t); // wait to enter the critical section
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode = leftSentinel) { signal(q->t); return -1; }
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    if (newRightNode = leftSentinel) signal(q->s);
    signal(q->t); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

---

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

```c
void PopRight() {
    ... wait(q->t);
    ... if (*) { signal(q->t); return } ...
    ... if (c) wait(q->s);
    ... if (c) signal(q->s);
    signal(q->t);
}
```
Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

```c
void PopRight()
{
    ...
    wait(q->t);
    ...
    if (s) { signal(q->t); return; }
    ...
    if (c) wait(q->s);
    if (c) signal(q->s);
    signal(q->t);
}
```

- in PushLeft, the lock set for s is empty
- here, the lock set of s is {t}
- \( I \sim s \) and transitive closure is \( I \sim t \)
- ~ the program cannot deadlock

Atomic Execution and Locks

Consider replacing the specific locks with atomic annotations:

double-ended queue: removal

```c
void PopRight()
{
    ...
    wait(q->t);
    ...
    if (s) { signal(q->t); return; }
    ...
    if (c) wait(q->s);
    if (c) signal(q->s);
    signal(q->t);
}
```

- nested atomic blocks still describe one atomic execution
- locks convey additional information over atomic
- locks cannot easily be recovered from atomic declarations

Outlook

Writing atomic annotations around sequences of statements is a convenient way of programming.

Idea: Replace atomic sections with locks:
- a single lock could be used to protect all atomic blocks
- more concurrency is possible by using several locks
  - compare the PushLeft, PopRight example
- some statements might modify variables that are never read by other threads -- no lock required
- statements in one atomic block might access variables in a different order to another atomic block -- deadlock prevention when creating locks
- creating too many lock can decrease the performance, especially when required to release locks in \( \lambda(t) \) when acquiring /
References


Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals.
- **composition**: several objects can be combined to a new object without interference.

Both, abstraction and composition are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`.
- a set object may internally use the list object and expose a set of operations, including `PushLeft`.

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals.
- **composition**: several objects can be combined to a new object without interference.

Both, abstraction and composition are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`.
- a set object may internally use the list object and expose a set of operations, including `PushLeft`.

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the **set** thread-safe.
Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals
- **composition**: several objects can be combined to a new object without interference

Both, abstraction and composition, are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`
- a set object may internally use the list object and expose a set of operations, including `PushLeft`

The `Insert` operations use the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the `set` thread-safe.

- wrap the two calls in `insert` in a mutex
- but other list operations can still be called — use the `same` mutex

Transactional Memory [2]

Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

```java
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```

Execute code as `transaction:

```java
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```

Execute code as `transaction`:

- execute the code of an atomic block
- check that it runs without `conflicts` due to accesses from another thread
**Transactional Memory [2]**

**Idea:** automatically convert atomic blocks into code that ensures atomic execution of the statements.

```java
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```

Execute code as transaction:
- execute the code of an atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
  - undo the computation done so far
  - re-start the transaction

**Managing Conflicts**

**Definition (Conflicts)**

A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
Managing Conflicts

Definition (Conflicts)
A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- optimistic vs. pessimistic concurrency control:
  - pessimistic: conflict occurrence, detection, resolution occur at once
  - resolution here is usually delaying one transaction
  - can be implemented using locks: deadlock problem

- optimistic: detection and resolution can happen after a conflict occurs
Managing Conflicts

Definition (Conflicts)
A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control**:
  - pessimistic: conflict occurrence, detection, resolution occur at once
    - resolution here is usually delaying one transaction
    - can be implemented using locks: deadlock problem
  - optimistic: detection and resolution can happen after a conflict occurs
    - resolution here must be aborting one transaction

- **eager vs. lazy version management**: how read and written data are managed during the transaction
  - eager: writes modify the memory and an undo-log is necessary if the transaction aborts
Managing Conflicts

Definition (Conflicts)

A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:

- **optimistic vs. pessimistic concurrency control**:
  - pessimistic: conflict occurrence, detection, resolution occur at once
  - resolution here is usually delaying one transaction
  - can be implemented using locks: deadlock problem
  - optimistic: detection and resolution can happen after a conflict occurs
  - resolution here must be aborting one transaction
  - need to repeated aborted transaction: livelock problem

- **eager vs. lazy version management**: how read and written data are managed during the transaction
  - eager: writes modify the memory and an undo-log is necessary if the transaction aborts
  - lazy: writes are stored in a redo-log and modifications are done on committing

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible

- **conflict detection**:
  - eager: conflicts are detected when memory locations are first accessed
Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- conflict detection:
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
  - **lazy**: conflicts are detected when committing a transaction
- reference of conflict (for non-**eager conflict** detection)
The goal is to use transactions to specify *atomic executions*.
Transactions are rooted in databases where they have the ACID properties:

- **Atomicity**: A transaction completes or seems not to have run.
  - We call this *failure atomicity* to distinguish it from *atomic executions*.
- **Consistency**: Each transaction transforms a consistent state to another consistent state.

Transactions are rooted in databases where they have the ACID properties:

- **Atomicity**: A transaction completes or seems not to have run.
**Semantics of Transactions**

The goal is to use transactions to specify *atomic executions*. Transactions are rooted in databases where they have the ACID properties:

- **atomicity**: a transaction completes or seems not to have run
  - we call this *failure atomicity* to distinguish it from *atomic executions*

- **consistency**: each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain *invariants* hold

- **isolation**: transactions do not influence each other
  - not so evident with respect to non-transactional memory

- **durability**: the effects are permanent ✓

Transactions themselves must be *serializable*.
Semantics of Transactions
The goal is to use transactions to specify atomic executions. Transactions are rooted in databases where they have the ACID properties:
- atomicity: a transaction completes or seems not to have run
  - we call this failure atomicity to distinguish it from atomic executions
- consistency: each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain invariants hold
  - invariants depend on the application (e.g., queue data structure)
- isolation: transactions do not influence each other
  - not so evident with respect to non-transactional memory
- durability: the effects are permanent
Transactions themselves must be serializable:
- the result of running current transactions must be identical to one execution of them in sequence
- serializability for transactions is insufficient to perform synchronization between threads

Consistency During Transactions
Consistency during a transaction.
ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction that is run on an inconsistent state may generate an inconsistent state → zombie transaction
  - this is usually ok since it will be aborted eventually
  - but transactions may cause havoc when run on inconsistent states
```c
atomic {
  int tmp1 = x;
  int tmp2 = y;
  assert(tmp1 - tmp2 == 0);
}
```
  // preserved invariant: x==y

Consistency During Transactions
Definition (opacity)
A TM system provides opacity if failing transactions are serializable w.r.t. committing transactions.
→ failing transactions still sees a consistent view of memory
Weak- and Strong Isolation

If guarantees are only given about memory accessed inside \textit{atomic}, a TM implementation provides \textit{weak isolation.}
Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard \textit{race} problems as in unlocked shared accesses

// Thread 1
atomic {
    \textcolor{red}{x = 0};
}

// Thread 2
int \textcolor{red}{tmp = x};

Weak- and Strong Isolation

If guarantees are only given about memory accessed inside \textit{atomic}, a TM implementation provides \textit{weak isolation.}
Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard \textit{race} problems as in unlocked shared accesses

// Thread 1
atomic {
    \textcolor{red}{x = 42;}
}

// Thread 2
int \textcolor{red}{tmp = x};

- give programs with races the same semantics as if using a single global lock for all \textit{atomic} blocks
Weak- and Strong Isolation

If guarantees are only given about memory accessed inside `atomic`, a TM implementation provides weak isolation. Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard race problems as in unlocked shared accesses
  ```
  // Thread 1
  atomic {
    x = 42;
    int tmp = x;
  }

  // Thread 2
  atomic {
    x = 42;
    int tmp = x;
  }
```
- give programs with races the same semantics as if using a single global lock for all `atomic` blocks
- **strong isolation**: retain order between accesses to TM and non-TM

Definition (SLA)
The single-lock atomicity is a model in which the program executes as if all transactions acquire a single, program-wide mutual exclusion lock.

~like sequential consistency, SLA is a statement about program equivalence

Observation:
Properties of Single-Lock Atomicity

Observation:
- SLA enforces order between TM and non-TM accesses ✓
  - this guarantees strong isolation between TM and non-TM accesses
  - within one transactions, accesses may be re-ordered ✓
- the content of non-TM memory conveys information which atomic block has executed, even if the TM regions do not access the same memory

Observation:
- SLA enforces order between TM and non-TM accesses ✓
  - this guarantees strong isolation between TM and non-TM accesses
  - within one transactions, accesses may be re-ordered ✓
- the content of non-TM memory conveys information which atomic block has executed, even if the TM regions do not access the same memory
  - SLA makes it possible to use atomic block for synchronization
Disadvantages of the SLA model

The SLA model is simple but often too strong:

1. SLA has a weaker progress guarantee than a transaction should have
   ```c
   // Thread 1
   atomic {
       while (true) {
   // Thread 2
   atomic {
       int tmp = x; // x in TM
   }
   ```

2. SLA correctness is too strong in practice
   ```c
   // Thread 1
   atomic {
       data = 1;
       while (true) {
       // Thread 2 not in atomic
       atomic {
           int tmp = data;
           if (ready) {
               // use tmp
               ready = 1;
           }
       }
   }
   ```

   - under the SLA model, atomic {} acts as barrier
   - intuitively, the two transactions should be independent rather than synchronize

Disadvantages of the SLA model

The SLA model is simple but often too strong:

1. SLA has a weaker progress guarantee than a transaction should have
   ```c
   // Thread 1
   atomic {
       while (true) {
   // Thread 2
   atomic {
       int tmp = x; // x in TM
   }
   ```

2. SLA correctness is too strong in practice
   ```c
   // Thread 2
   atomic {
       int tmp = data;
       // Thread 1 not in atomic
       atomic {
           if (ready) {
               // use tmp
               ready = 1;
           }
       }
   ```

Transactionally Sequential Consistency

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks
- the programmer cannot rely on synchronization

Definition

TSC The transactional sequential consistency is a model in which the accesses within each transaction are sequentially consistent.
**Transactional Sequential Consistency**

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks
- ~ the programmer cannot rely on synchronization

**Definition**

TSC The transactional sequential consistency is a model in which the accesses within each transaction are sequentially consistent.

- TSC is weaker: gives strong isolation, but allows parallel execution ✓
- TSC is stronger: accesses within a transaction may not be reordered !

---

**Quick Quiz**

Associate one item on the left with one or two on the right.

- a transaction waits rather than creating a conflict
- in case of a conflict, a kind of log is needed
- a zombie transaction sees an inconsistent state
- no guarantee if a transaction accesses a non-TM
- a write in a transaction is immediately globally visible
- redo and undo
- conflict detection
- concurrency control
- isolation
- version management
- eager, lazy
- optimistic, pessimistic
- strong, weak

---

**Translation of atomic-Blocks**

A TM system must track which shared memory locations are accessed:

- convert every read access \( x \) from a shared variable to \( \text{ReadFix}(x) \)
Translation of atomic-Blocks

A TM system must track which shared memory locations are accessed:
- convert every read access x from a shared variable to \texttt{ReadTx}(\$x)
- convert every write access \(x=e\) to a shared variable to \texttt{WriteTx}(\$x,e)

Convert atomic blocks as follows:

\[
\begin{align*}
\text{atomic} & \quad \rightarrow \quad \text{do} \quad \text{// code} \quad \rightarrow \quad \text{// code with } \texttt{ReadTx} \text{ and } \texttt{WriteTx} \quad \text{// code with } \texttt{CommitTx} \text{ and } \texttt{AbortTx} \\
\text{while} & \quad \text{// CommitTx()};
\end{align*}
\]

- translation can be done using a pre-processor
  - determining a minimal set of memory accesses that need to be transactional
    requires a good static analysis
  - idea: translate all accesses to global variables and the heap as TM
  - more fine-grained control using manual translation
- an actual implementation might provide a \texttt{retry} keyword
  - when executing \texttt{retry}, the transaction aborts and re-starts
  - the transaction will again wind up at \texttt{retry} unless its \texttt{read-set} changes
  - similar to condition variables in monitors

Transactionality for the Queue

If a preprocessor is used, \texttt{PopRight} can be implemented as follows:

```c
double-ended queue: removal

int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    atomic {
        QNode* rightSentinel = q->right;
        oldRightNode = rightSentinel->left;
        if (oldRightNode==leftSentinel) retry;
        QNode* newRightNode = oldRightNode->left;
        newRightNode->right = rightSentinel;
        rightSentinel->left = newRightNode;
    }
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- the transaction will abort if other threads call \texttt{PopRight}

Transactionality for the Queue

If a preprocessor is used, \texttt{PopRight} can be implemented as follows:

```c
double-ended queue: removal

int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    atomic {
        QNode* rightSentinel = q->right;
        oldRightNode = rightSentinel->left;
        if (oldRightNode==leftSentinel) retry;
        QNode* newRightNode = oldRightNode->left;
        newRightNode->right = rightSentinel;
        rightSentinel->left = newRightNode;
    }
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- the transaction will abort if other threads call \texttt{PopRight}
- if the queue is empty, it may abort if \texttt{PushLeft} is executed (lines 8, 9)
A Software TM Implementation

A software TM implementation allocates a transaction descriptor to store data specific to each atomic block, for instance:

- undo-log of writes if writes have to be undone if a commit fails
- redo-log of writes if writes are postponed until a commit
- read- and write-set: locations accessed so far
- read- and write-version: time stamp when value was accessed

Consider the TM2 STM (software transactional memory) algorithm [1]:

Principles of TL2

The idea is to obtain a version tx.RV from the global clock when starting the transaction, the read-version, and set the versions of all written cells to a new version on commit.

A read from a field at offset of object obj is implemented as follows:

```
transactional read
int ReadTx(TMDesc tx, object obj, int offset) {
    if ((obj[offset]) in tx.redoLog) {
        return tx.redoLog[obj[offset]];
    } else {
        atomic { v1 = obj.timestamp; locked = obj.sem<1; }
        result = obj[offset];
        v2 = obj.timestamp;
        if (locked || v1 != v2 || v1 > tx.RV) AbortTx(tx);
        tx.readSet = tx.readSet.add(obj);
        return result;
    }
```

WriteTx is simpler: add or update the location in the redo-log.
**Committing a Transaction**

A transaction can succeed if none of the read locations has changed:

```c
bool CommitTx(TMDesc tx) {
  foreach (e in tx.writeSet) {
    if (!try wait(e.obj.sem)) goto Fail;
    iW = FetchAndAdd(&globalClock);
    foreach (e in tx.readValue) {
      if (e.obj.version > tx.iW) goto Fail;
      foreach (e in tx.redoLog) {
        e.obj[e.offset] = e.value;
        foreach (e in tx.writeSet) {
          e.obj = iW; signal(e.obj.sem);
        }
      }
    }
    return true;
  }
  Fail:
  // signal all acquired semaphores
  return false;
}
```

**Properties of TL2**

Opacity is guaranteed by aborting a read access with an inconsistent value:

```
StartTx  ReadTx  WriteTx  ReadTx  CommitTx
memory state seems to be consistent
```

Other observations:
- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)

**Hardware Transactional Memory**

Transactions of a limited size can also be implemented in hardware:
- additional hardware to track read- and write-sets
Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:
- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - due to limited size, a STM backup must be provided

Two principal implementation of HTM:
- Explicit Transactional HTM: each access is marked as transactional
  - similar to \texttt{StartTx}, \texttt{ReadTx}, \texttt{WriteTx}, and \texttt{CommitTx}
  - track an extra bit with each cache-line that is set if the transaction became invalid, return this bit after each access
- Implicit Transactional HTM: only the beginning and end of a transaction are marked
Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is **eager** using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

~ due to limited size, a STM backup must be provided

Two principal implementation of HTM:

- Explicit Transactional HTM: each access is marked as transactional
  - similar to **StartTx, ReadTx, WriteTx, and CommitTx**
  - track an extra bit with each cache-line that is set if the transaction became invalid, return this bit after each access

- Implicit Transactional HTM: only the beginning and end of a transaction are marked
  - provide a target to jump when transaction aborts

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): **implicit transactional**

- Hardware Lock Elision

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): **implicit transactional**

- Hardware Lock Elision
  - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

- **Hardware Lock Elision**
  - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
  - requires annotations

- **Restricted Transactional Memory**
  - provides new instructions `XBEGIN, XEND, XABORT, and XTTEST`
  - `XBEGIN` takes an instruction address where execution continues if the transaction aborts
  - `XABORT` aborts the current transaction with an error code
  - `XTTEST` checks if the processor is executing transactionally
  - internal operations similar to lock elision
  - aborts on every use of OS calls, IO, accesses to non-MESI addresses, etc.
  - programmer must provide alternative code path
  - in contrast to lock elision, there is no progress guarantee
  - **Nested locks**

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

- **Hardware Lock Elision**
  - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
  - requires annotations
    - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
    - instruction that increments the semaphore must be prefixed with `XRELEASE`
    - these prefixes are ignored on older platforms
  - after `XACQUIRE instr`, all accesses are stored in read/write-sets
  - the value of `instr` updating to 0 is only read, not written
  - any accessed cache line is tracked in the read/write-sets
  - if any other processor invalidates any of these cache lines, the transaction aborts
  - when `XRELEASE instr` is seen, tracking of read/write-sets stops
  - if `XRELEASE instr` writes a value different to v, the transaction aborts
    - a shadow copy of the processor state at `XACQUIRE`
  - aborting a transaction requires:
    - an invalidation of all cache lines in the read/write sets
    - a re-execution of the code with normal lock semantics
Integrating Non-TM Resources

Allowing access to other resources than memory inside an atomic block poses problems:
- storage management, condition variables, volatile variables, input/output
- semantics should be as if atomic implements SLA or TSC semantics

Usual choice is one of the following:
- Prohibit It. Certain constructs do not make sense. Use compiler to reject these programs.
- Execute It. Library routines may be executable as transactions.
- Irrevocably Execute It. Universal way to deal with operations that cannot be undone: ensure that this transaction is able to terminate before starting it by making all other transactions conflict.
- Integrate It. Re-write code to be transactional: error logging, writing data to a file, ...

--- currently best to use TM only for memory; check if TM supports irrevocable transactions

---

Transactional Memory: Summary

Transactional memory aims to provide atomic blocks for general code:
- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

The devil lies in the details:
- semantics of explicit HTM and STM transactions quite subtle when mixing with non-TM (weak vs. strong isolation)
- single-lock atomicity and transactional sequential consistency semantics
- STM not the right tool to synchronize threads
- STM providing opacity require eager conflict detection
Outlook

Several other principles exist for concurrent programming:

1. **non-blocking message passing (the actor model)**
   - a program consists of actors that send messages
   - each actor has a queue of incoming messages
   - messages can be processed and new messages can be sent
   - special filtering of incoming messages
   - example: Erlang, many add-ons to existing languages

2. **blocking message passing (CSP, π-calculus, join-calculus)**
   - a process sends a message over a channel and blocks until the recipient accepts it
   - channels can be send over channels (π-calculus)
   - examples: Occam, Occam-π, Go

3. **(immediate) priority ceiling**
   - declare processes with priority and resources that each process may acquire
   - each resource has the maximum (ceiling) priority of all processes that may acquire it
   - a process' priority at run-time increases to the maximum of the priorities of held resources
   - the process with the maximum (run-time) priority executes

References

D. Dice, O. Shalev, and N. Shavit.
Transactional Locking II.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.