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();
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:

1. `a.bar()` (acquires lock on `a`)
2. `b.bar()` (acquires lock on `b`)
3. `a.bar()` (tried to acquire lock on `b`, which is held by `b.bar()`) - Deadlock

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();
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:

1. `a.bar()` (acquires lock on `a`)
2. `b.bar()` (acquires lock on `b`)
3. `a.bar()` (tried to acquire lock on `b`, which is held by `b.bar()`) - Deadlock

Sequence leading to a deadlock:

1. `a.bar()` (acquires lock on `a`)
2. `b.bar()` (acquires lock on `b`)
3. `a.bar()` (tried to acquire lock on `b`, which is held by `b.bar()`) - Deadlock
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 != null) {
            other.bar();
        }
    }
}
```

and two instances:

```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
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()`
- A blocks on the monitor of b

---

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 != null) {
            other.bar();
        }
    }
}
```

and two instances:

```java
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
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()`
- A blocks on the monitor of b

---
Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [Coffman et al.(1971)Coffman, Elphick, and Shoshani]:

- **mutual exclusion**: processes require exclusive access
- **wait for**: a process holds resources while waiting for more
- **no preemption**: resources cannot be taken away from processes
- **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

---

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [Coffman et al.(1971)Coffman, Elphick, and Shoshani]:

- **mutual exclusion**: processes require exclusive access
- **wait for**: a process holds resources while waiting for more
- **no preemption**: resources cannot be taken away from processes
- **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
- **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock
**Treatment of Deadlocks**

Deadlocks occur if the following four conditions hold [Coffman et al. (1971); Coffman, Elphick, and Shoshani]:

- **mutual exclusion**: processes require exclusive access
- **wait for**: a process holds resources while waiting for more
- **no preemption**: resources cannot be taken away form processes
- **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
- **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

**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 \( p \).
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
\[
\sigma^0 = \sigma \quad \text{and} \quad \sigma^{i+1} = \{ (x_1,x_3) \mid \exists x_2 \in X. (x_1,x_2) \in \sigma^i \land (x_2,x_3) \in \sigma^i \}.
\]

Each time a lock is acquired, we track the lock set at \( p \):

\[ \sigma < \ell \]

**Definition (lock order)**
Define \( \prec \subseteq L \times L \) such that \( l \prec l' \) iff \( l \in \lambda(p) \) and the statement \( v_0 \) is of the form *wait(l')* or *monitor.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.

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 \( b \in L_S \), \( a \prec a \) and \( b \in L_M \), 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 $\exists b \in L_S . a \prec a$ and $\exists a \in L_M, b \in L . a \neq b \wedge a \prec b \wedge b \prec a$ then the program is free of deadlocks.

Note: the set $L$ contains instances of a lock.
- the set of lock instances can vary at runtime
- if we statically want to ensure that deadlocks cannot occur:
  - summarize every monitor that may have several instances into one
  - a summary lock $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - require that $\bar{a} \neq \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Example: Deadlock freedom
Is the example deadlock free? Consider its skeleton:

**double-ended queue: removal**

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

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 $L_M^* \subseteq L_M$

Example: Deadlock freedom
Is the example deadlock free? Consider its skeleton:

**double-ended queue: removal**

```c
void PopRight() {
    ...
    wait(q->t);
    ...
    if (*) { 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 $\{l\}$
- $l < s$ and transitive closure is $t < s$
- $\sim$ the program cannot deadlock
Atomic Execution and Locks

Consider replacing the specific locks with \texttt{atomic} annotations:

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

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

Outlook

Writing \texttt{atomic} annotations around sequences of statements is a convenient way of programming.

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

→ creating locks automatically is non-trivial and, thus, not standard in programming languages

---

**Concurrency across Languages**

In most systems programming languages (C, C++) we have
- the ability to use *atomic* operations
- → we can implement *wait-free* algorithms

In Java, C# and other higher-level languages
- provide monitors and possibly other concepts

---

**Concurrent across Languages**

In most systems programming languages (C, C++) we have
- the ability to use *atomic* operations
- → we can implement *wait-free* algorithms

In Java, C# and other higher-level languages
- provide monitors and possibly other concepts
- often simplify the programming but incur the same problems

<table>
<thead>
<tr>
<th>language</th>
<th>barriers</th>
<th>wait-/lock-free</th>
<th>semaphore</th>
<th>mutex</th>
<th>monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Java, C#</td>
<td>-</td>
<td>-</td>
<td>(b)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(a) some pthread implementations allow a *reentrant* attribute
(b) simulate semaphores using an object with two *synchronized* methods
Summary

Classification of concurrency algorithms:
- wait-free, lock-free, locked
- next on the agenda: transactional

Wait-free algorithms:
- never block, always succeed, never deadlock, no starvation
- very limited in what they can do

Lock-free algorithms:
- never block, may fail, never deadlock, may starve
- invariant may only span a few bytes (8 on Intel)

Locking algorithms:
- can guard arbitrary code
- can use several locks to enable more fine grained concurrency
- may deadlock
- semaphores are not re-entrant, monitors are
- use algorithm that is best fit

References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ISSN 0360-0300.

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

\[
\begin{align*}
    & s.t. (q) \\
    & \text{insert}(s) \cup \text{compute}(q) \cup \cdots \\
    & \text{if } s.t. (q) \text{ then } \text{PushLeft}(q, s) \\
    & s.t. (q) \cup \text{compute}(q) \cup \cdots \\
    & \text{else } \text{PushLeft}(q, s)
\end{align*}
\]

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.

\[
\begin{align*}
    & s.t. (q) \\
    & \text{insert}(s) \cup \text{compute}(q) \cup \cdots \\
    & \text{if } s.t. (q) \text{ then } \text{PushLeft}(q, s) \\
    & s.t. (q) \cup \text{compute}(q) \cup \cdots \\
    & \text{else } \text{PushLeft}(q, s)
\end{align*}
\]
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 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.
- ~~ wrap the two calls in Insert in a mutex
- but other list operations can still be called ~~ use the same mutex

~~ unlike sequential algorithms, thread-safe algorithms cannot always be composed to give new thread-safe algorithms

---

Transactional Memory [2]

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

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

Execute code as *transaction*:
Transactional Memory [2]

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

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

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single 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.

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

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
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
- nested atomic blocks act like a single 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

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
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

Concurrent Transactions | Transaction Semantics
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

- **Concurrency: Transactions**
- **Transaction Semantics**
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:
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)
  - *tentative* detect conflicts before transactions commit, e.g. aborting when transaction TA reads while TB may write the same location
  - *committed* detect conflicts only against transactions that have committed

**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*
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.
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

```c
atomic {
    int tmp1 = x;
    int tmp2 = y;
    assert(tmp1-tmp2==0);
}
```

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 x = 10;
    y = 10;
}
```
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 \( \rightsquigarrow \) 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;
    atomic {
        x = 10;
        y = 10;
        assert(tmp1-tmp2==0);
    }
```

- critical for C/C++ if, for instance, variables are pointers

---

**Definition (opacity)**

A TM system provides *opacity* if failing transactions are serializable w.r.t. committing transactions.

\( \rightsquigarrow \) failing transactions still sees a consistent view of memory

---

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

---

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?
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

```c
// Thread 1
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.
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;
  }
  ```
- 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.

---

Properties of Single-Lock Atomicity

**Observation:**

- SLA enforces order between TM and non-TM accesses ✓

---

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
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
  - SLA makes it possible to use atomic block for synchronization

Disadvantages of the SLA model

The SLA model is simple but often too strong:
- SLA has a weaker progress guarantee than a transaction should have
  // Thread 1
  atomic {
    while (true) {};
  }
  // Thread 2
  atomic {
    int tmp = x; // x in TM
  }
- SLA correctness is too strong in practice
  // Thread 2
  atomic {
    int tmp = data;
  // Thread 1 not in atomic
  if (ready) {
    // use tmp
  }

Transaction Semantics

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 re-ordered △
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 re-ordered

---

<table>
<thead>
<tr>
<th>Concurrency: Transactions</th>
<th>Transaction Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

race free re-orderings