Script generated by TTT

Title: Simon: Programmiersprachen (22.11.2013)

Date: Fri Nov 22 14:16:26 CET 2013

Duration: 90:03 min

Pages: 96

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.)

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:

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

and two instances:

```
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:

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:

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

and two instances:

```
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()`

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:

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

and two instances:

```
  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()`

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:

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

and two instances:

```
  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()`
- 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:

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

and two instances:

```
  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()`
- A blocks on the monitor of b
- B happens to execute `other.bar()`
- ⇝ both *block* indefinitely

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:

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

and two instances:

```
  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()`
- A blocks on the monitor of b
- 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 [Coffman et al.(1971)Coffman, Elphick, and Shoshani]:

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

---

Treatment of Deadlocks

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

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 [Coffman et al.(1971)Coffman, Elphick, and Shoshani]:

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*
3. *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]:

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*
3. *prevention*: design programs to be deadlock-free
4. *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]:

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*
3. *prevention*: design programs to be deadlock-free
4. *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.

$a \rightarrow b$

lock(a)
// b is taken

unlock(a)

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 σ^+ of a relation σ:

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

$$\begin{array}{rcl} \sigma^0 & = & \sigma \\ \sigma^{i+1} & = & \{\langle x_1, x_3\rangle \mid \exists x_2 \in X \,.\, \langle x_1, x_2\rangle \in \sigma^i \wedge \langle x_2, x_3\rangle \in \sigma^i\} \end{array}$$

$\sigma^0 = \sigma =$

$\sigma^1 =$

$\sigma^+ = \bigcup \sigma^i =$

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 σ^+ of a relation σ:

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

$$\begin{array}{rcl} \sigma^0 & = & \sigma \\ \sigma^{i+1} & = & \{\langle x_1, x_3\rangle \mid \exists x_2 \in X \,.\, \langle x_1, x_2\rangle \in \sigma^i \wedge \langle x_2, x_3\rangle \in \sigma^i\} \end{array}$$

l wait(l')

$l \lhd l'$ l l' wait(l'')

Each time a lock is acquired, we track the lock set at p: $l \lhd l'$

Definition (lock order)

Define $\lhd \subseteq L \times L$ such that $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the strict lock order $\prec = \lhd^+$.

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

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 $\nexists a \in L_S \,.\, a \prec a$ and $\nexists a \in L_M, b \in L \,.\, a \neq b \wedge a \prec b \wedge b \prec a$ 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 $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

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 $L^s_M \subseteq L_M$

{t} ———

{t}

{s,t}

$t \lhd s$ $t \prec s$

(s)

[illegible]

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

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

$t \in \emptyset$

{t}

$s \notin \{t\}$ ✓

{s,t}

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

```
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 $\{t\}$
- $t \lhd s$ and transitive closure is $t \prec s$
- ⇝ the program cannot deadlock

$a \prec a$

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

double-ended queue: removal

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

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

double-ended queue: removal

```
void PopRight() {
  ...
  wait(q->t);
  ...
  if (*) { 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.

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 use 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 $\lambda(l)$ when acquiring l

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 use 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 $\lambda(l)$ when acquiring l

⇝ 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

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

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
- often simplify the programming but incur the same problems

language	barriers	wait-/lock-free	semaphore	mutex	monitor
C,C++	✓	✓	✓	✓	(a)
Java,C#	-	-	(b)	✓	✓

(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:

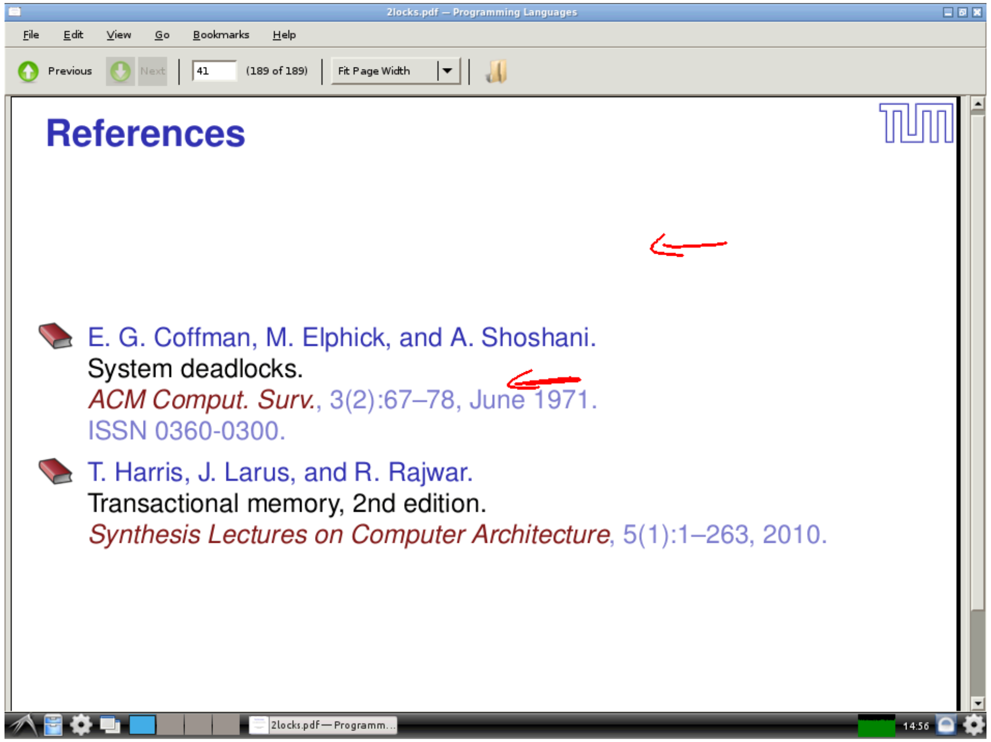
- can guard arbitrary code
- can use several locks to enable more fine grained concurrency
- may deadlock
- semaphores are not re-entrant, monitors are

⇝ use algorithm that is best fit

References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ACM Comput. Surv., 3(2):67–78, June 1971.
ISSN 0360-0300.

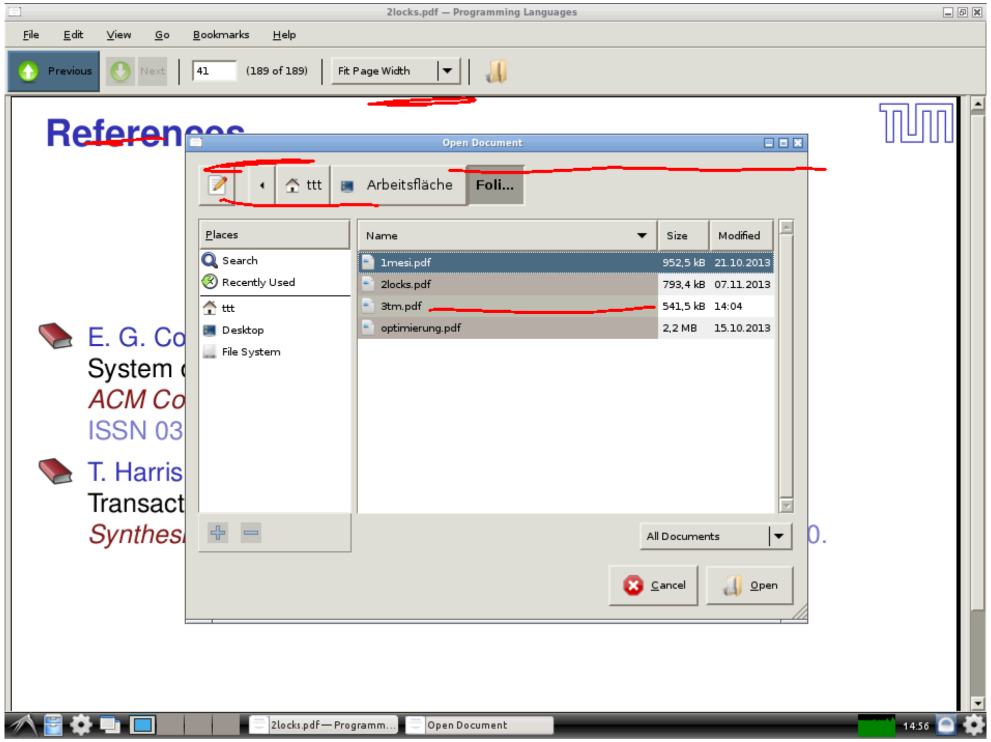
T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.



References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ACM Comput. Surv., 3(2):67–78, June 1971.
ISSN 0360-0300.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.



Name	Size	Modified
1mesi.pdf	952,5 kB	21.10.2013
2locks.pdf	793,4 kB	07.11.2013
3tm.pdf	541,5 kB	14:04
optimierung.pdf	2,2 MB	15.10.2013

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.

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.

```
Set { q
  Insert(s) {
     empty=forAll(q...)   // s ∈ q
     if empty then   PushLeft(q, s)
} }
```

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

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

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.

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

Execute code as *transaction*:

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.

```
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.

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

Execute code as *transaction*:

- execute the code of an atomic block

Transactional Memory [2]

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

```
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

Transactional Memory [2]

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

```
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.

```
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.

```
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

Transactional Memory [2]

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

```
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
  - 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*:

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

---

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

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

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

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

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

---

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

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:

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

---

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

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

---

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

---

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

---

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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
3. reference of conflict (for non-*eager conflict* detection)

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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
3. reference of conflict (for non-*eager conflict* detection)
   - *tentative* detect conflicts before transactions commit, e.g. aborting when transaction TA reads while TB may writes 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:

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

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

---

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

---

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 ✓

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

---

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

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

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             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 ⇝ zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             y = 10;
}                                 }
```

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

---

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

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             y = 10;
}                                 }
```

- 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.

⇝ 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?

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?

- no conflict detection for non-transactional accesses
- standard *race* problems as in unlocked shared accesses

```
// Thread 1
atomic {                        // Thread 2
  x = 42;                       int tmp = x;
}
```

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 {                        // Thread 2
  x = 42;                       int tmp = x;
}
```

- ⇝ give programs with races the same semantics as if using a single global lock for all `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 {                        // Thread 2
  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

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 {                        // Thread 2
  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 {                          // Thread 2
  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

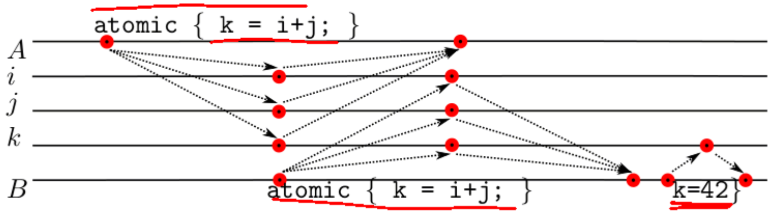
Definition (SLA)

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

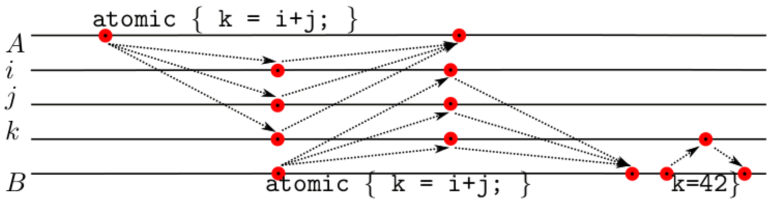
---

Properties of Single-Lock Atomicity



Observation:

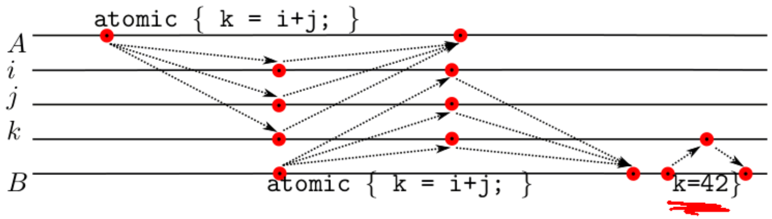
Properties of Single-Lock Atomicity



Observation:

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

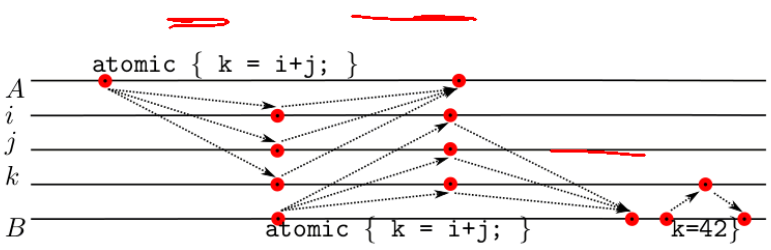
Properties of Single-Lock Atomicity



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:

1. 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
}
```

2. SLA correctness is too strong in practice

```
// Thread 1
data = 1;
atomic {
}
ready = 1;
```

```
// Thread 2
atomic {
  int tmp = data;
  // Thread 1 not in atomic
  if (ready) {
    // use tmp
  }
}
```

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

```
// Thread 1
atomic {
  while (true) {};
}
```

```
// Thread 2
atomic {
  int tmp = x; // x in TM
}
```

2. SLA correctness is too strong in practice

```
// Thread 1
data = 1;
atomic {
}
ready = 1;
```

```
// Thread 2
atomic {
  int tmp = data;
  // Thread 1 not in atomic
  if (ready) {
    // use tmp
  }
}
```

- under the SLA model, `atomic {}` acts as barrier

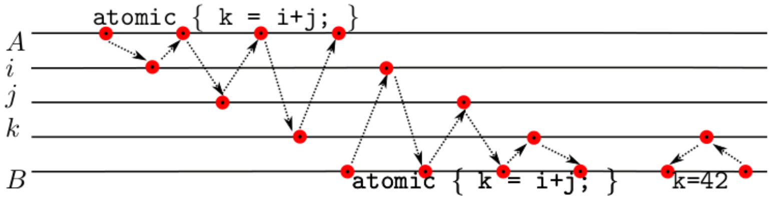
Transactional Sequential Consistency

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 ⚠

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

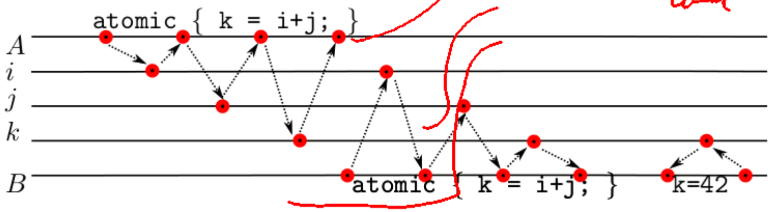
Definition (TSC)

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

lazy

permissive

weak



eager

- TSC is weaker: gives *strong isolation*, but allows parallel execution ✓
- TSC is stronger: accesses within a transaction may *not* be re-ordered ⚠

⇝ actual implementations use TSC with some *race free* re-orderings