Basic Transactions

Dave Clarke
dave.clarke@it.uu.se

Basic Transactions

Today, we will look at transactions from the perspective of a **low-level** programmer (or, equivalently, from the perspective of a compiler that is using transactional memory to implement features like atomicity in a high-level programming language).

This is *not* how you will use transactions in a high-level language—but it will help you understand what's going on behind the scenes.

Transactional Memory: A Simple Interface

Transaction management:

```
void StartTx();
bool CommitTx();
void AbortTx();
```

Data access:

```
T ReadTx(T *addr);
void WriteTx(T *addr, T v);
```

Basic Transactional Memory: Example

```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  do {
    StartTx();
    QNode *leftSentinel = ReadTx(&(q->left));
    QNode *oldLeftNode = ReadTx(&(leftSentinel->right));
    WriteTx(&(qn->left), leftSentinel);
    WriteTx(&(qn->right), oldLeftNode);
    WriteTx(&(leftSentinel->right), qn);
    WriteTx(&(oldLeftNode->left), qn);
  } while (!CommitTx());
}
```

Basic Transactional Memory: Remarks

Typical structure:

```
do {
   StartTx();
   // update from one consistent state to another
   // ...
} while (!CommitTx());
```

It is not necessary to explicitly acquire and release locks—this is managed by the TM implementation.

(Try writing a lock-based implementation of `PushLeft` that doesn't block concurrent `PushRight` operations. It's pretty hard!)

Today

1. Design choices for transactional memory
2. Semantics of transactions
3. Performance and progress

Design choices for transactional memory

Concurrency Control: Pessimistic vs. Optimistic

The TM implementation must detect conflicts, and resolve them by delaying or aborting transactions.

- Pessimistic concurrency control: conflicts are detected **when they occur**, and resolved immediately—in effect, transactions have exclusive ownership of data (cf. locking)
- Optimistic concurrency control: conflicts are detected and resolved some time **before a transaction commits**—allowing conflicting transactions to continue to run in parallel for some time

Concurrency Control: Deadlock vs. Lifelock

Pessimistic concurrency control requires care in the TM implementation to avoid deadlocks.

Optimistic concurrency control requires care in the TM implementation to avoid livelocks.

Concurrency Control: Performance

Pessimistic concurrency control is typically superior when conflicts are frequent (once a transaction holds its locks, it can run to completion).

If conflicts are rare, optimistic concurrency control avoids the cost of locking and may increase concurrency.

Hybrid approaches are commonly used.

Version Management: Eager vs. Lazy

The TM implementation must execute the (tentative) operations performed by a transaction.

- Eager versioning (aka direct update): updates are directly written to memory, maintaining information about overwritten values in an undo log (for when the transaction aborts)
- Lazy versioning (aka deferred update): updates are collected in a private buffer (redo log) and written to memory when the transaction commits

Eager versioning requires pessimistic concurrency control (for write operations) to prevent data races.

Conflict Detection: Granularity

TM implementations often do not perform conflict detection at the level of individual memory locations.

Instead, conflict detection may be performed at a granularity of, e.g., words or cache lines (in hardware TM) or complete objects (in software TM).

A false conflict occurs when the TM implementation treats two transactions as conflicting even though they have accessed distinct memory locations.

False Conflicts: Example

This code might trigger a false conflict in some (hardware or software) TM implementations:

```
// shared data
struct {
  int x;
  int y;
} s;
```

```
// Thread 1
do {
  StartTx();
  WriteTx(&s.x, 1);
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  WriteTx(&s.y, 1);
} while (!CommitTx());
```

Conflict Detection: Eager vs. Lazy

Optimistic concurrency control permits a wide spectrum of conflict detection techniques.

- Eager conflict detection: conflicts are identified while transactions are running, i.e., as soon as they attempt to access data
- Conflicts can be identified by a special validation step, which may occur at any time (or even multiple times) during a transaction's execution
- Lazy conflict detection: conflicts are identified when transactions attempt to commit

Conflict Detection: Tentative vs. Committed

TM implementations may consider different strategies to determine whether a conflict has occurred.

- Tentative conflict detection identifies conflicts between running transactions.
- Committed conflict detection only considers conflicts between active transactions and those that have already committed.

Usually, conflict detection is eager+tentative or lazy+committed.
Hybrid approaches are also commonly used.

Semantics of transactions

Semantics of Transactions: Caveat

It is important that programming abstractions have clean, simple semantics.

Unfortunately, there is no universally agreed semantics of transactions.

Different choices in the implementation of TM systems lead to slightly different semantics.

Two Levels of Concurrency

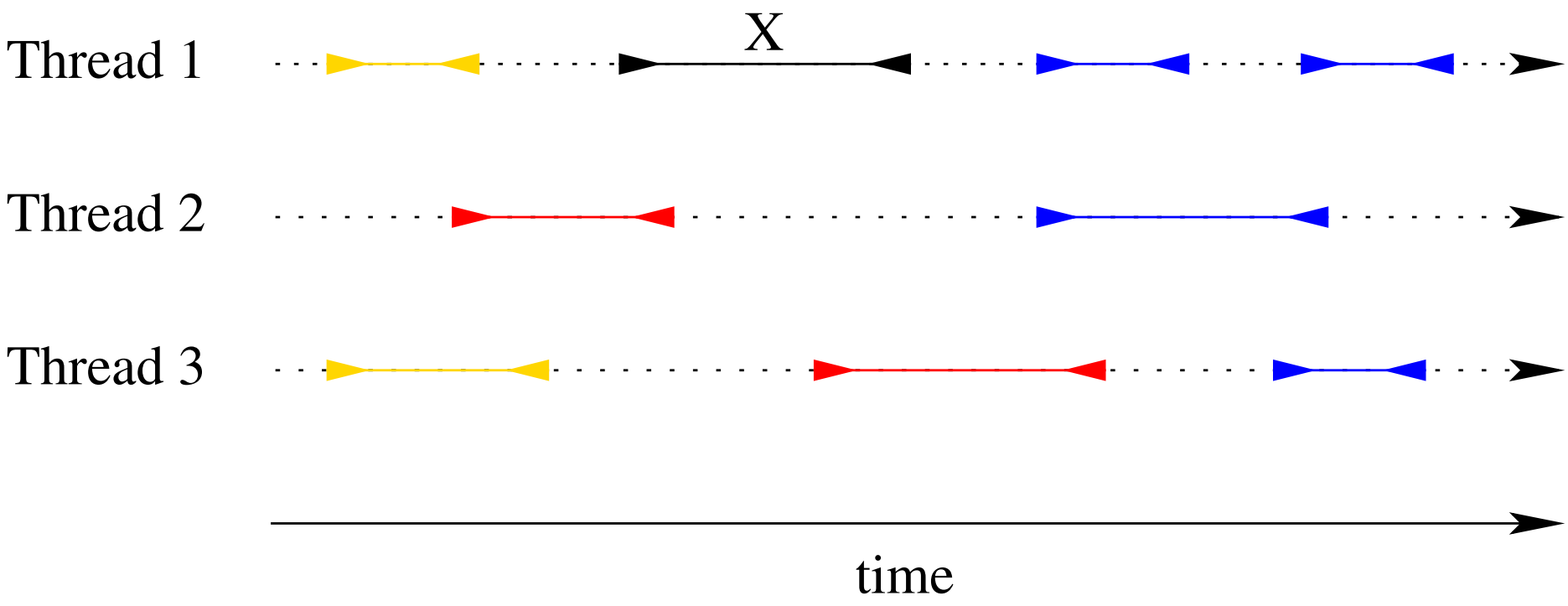
- Concurrency between individual TM operations
- Concurrency between transactions

Individual TM Operations: Linearizability

To abstract over the low-level details of the TM's concurrency control, we require that individual TM operations (e.g., `StartTx`, `CommitTx`, `AbortTx`, `ReadTx`, `WriteTx`) are linearizable.

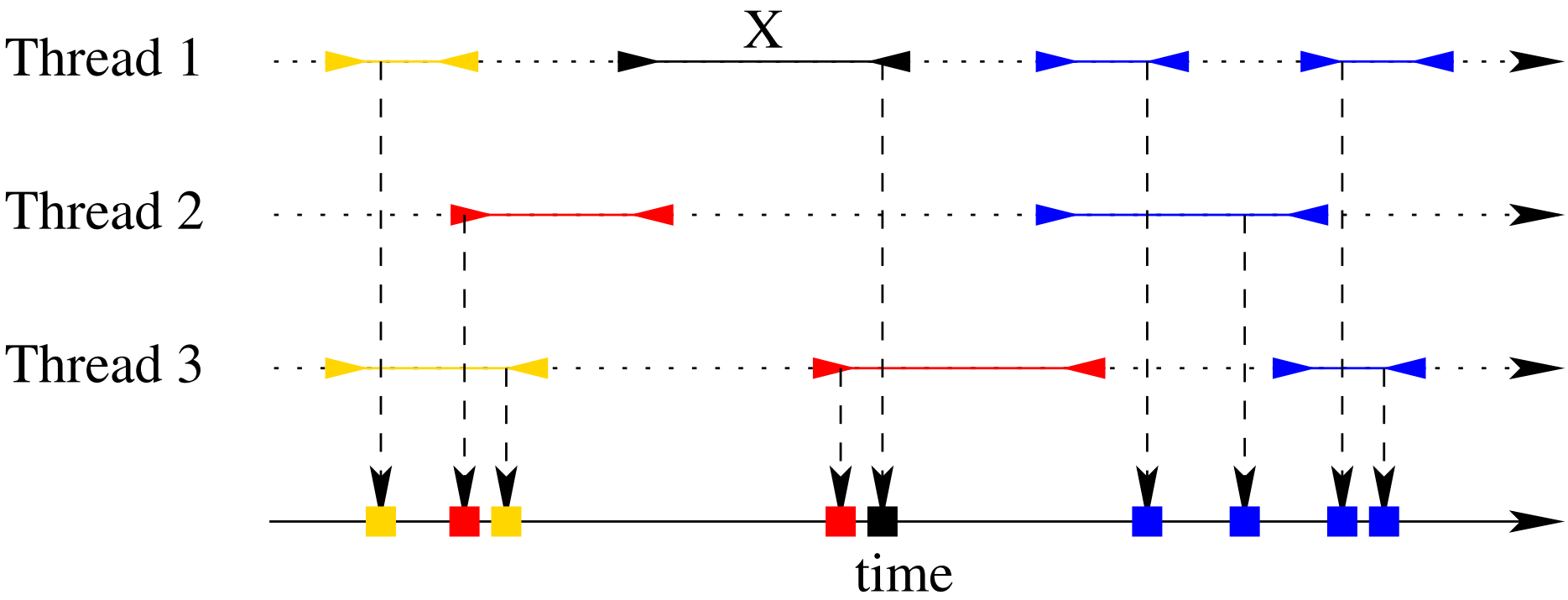
Linearizability requires that each individual operation **appears to take place atomically at some point between its invocation and its return**.

Linearizability: Example



Definitely before X Could happen before or after X Definitely after X

Linearizability: Example



Sequential Semantics of Transactions

We still need to consider what different sequential orderings of TM operations mean (i.e., which behaviors we might observe).

This is known as the sequential semantics of transactions.

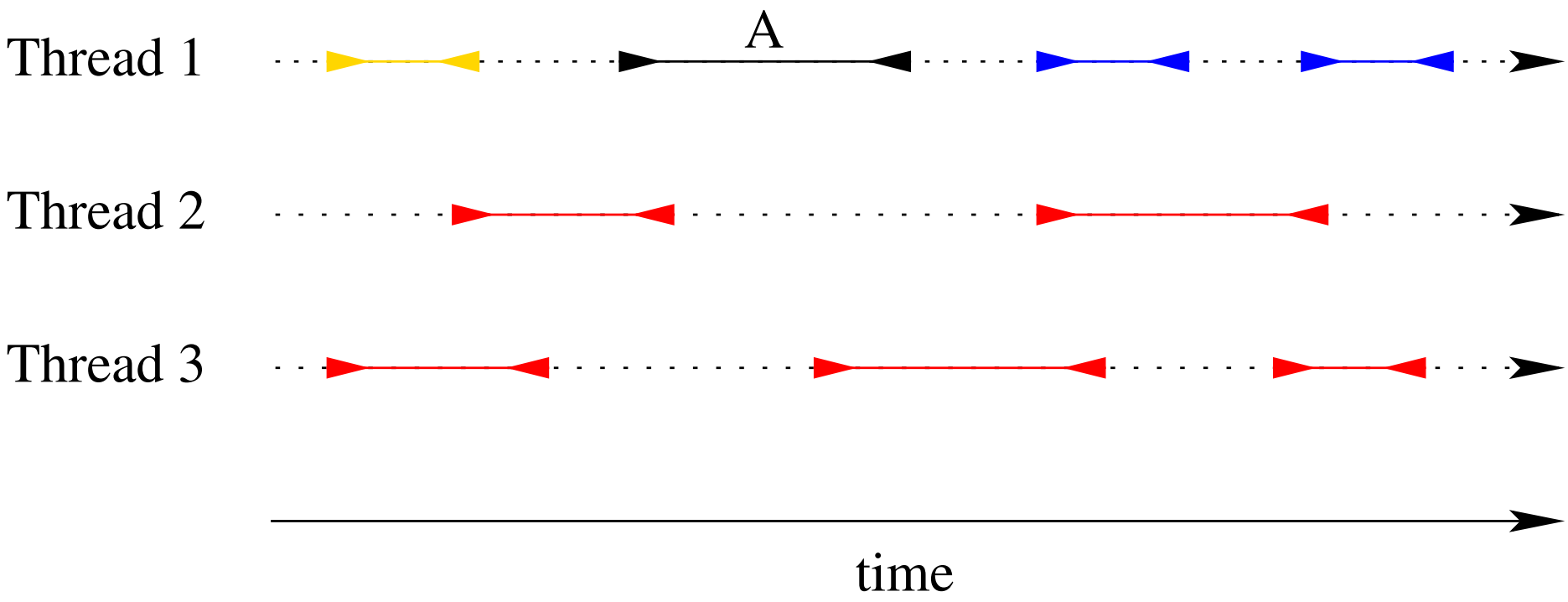
There are many possible choices with different trade-offs between efficiency and programmability (i.e., guarantees provided).

Serializability

Serializability requires that the result of a concurrent execution is identical to the result of **some sequential execution** of the transactions.

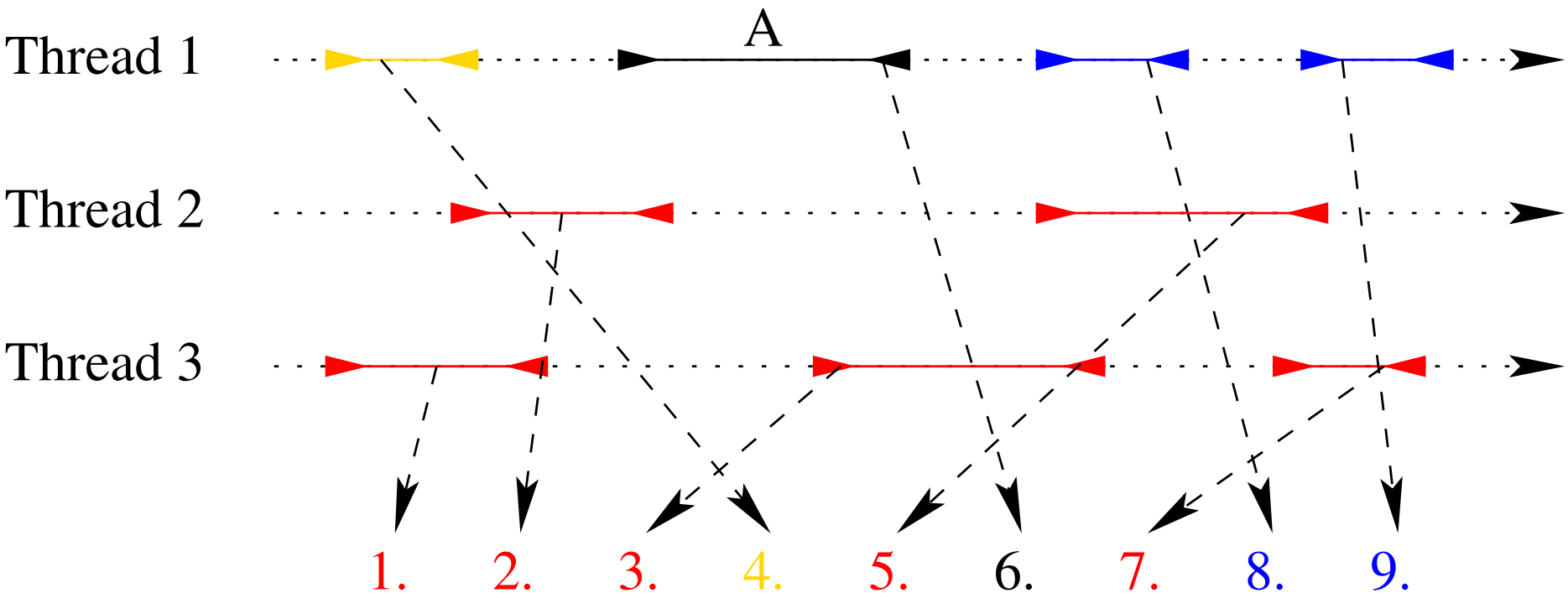
Serializability ensures isolation (cf. ACID). However, it does not impose any real-time constraints on the sequential execution.

Serializability: Example



Definitely before A Could happen before or after A Definitely after A

Serializability: Example

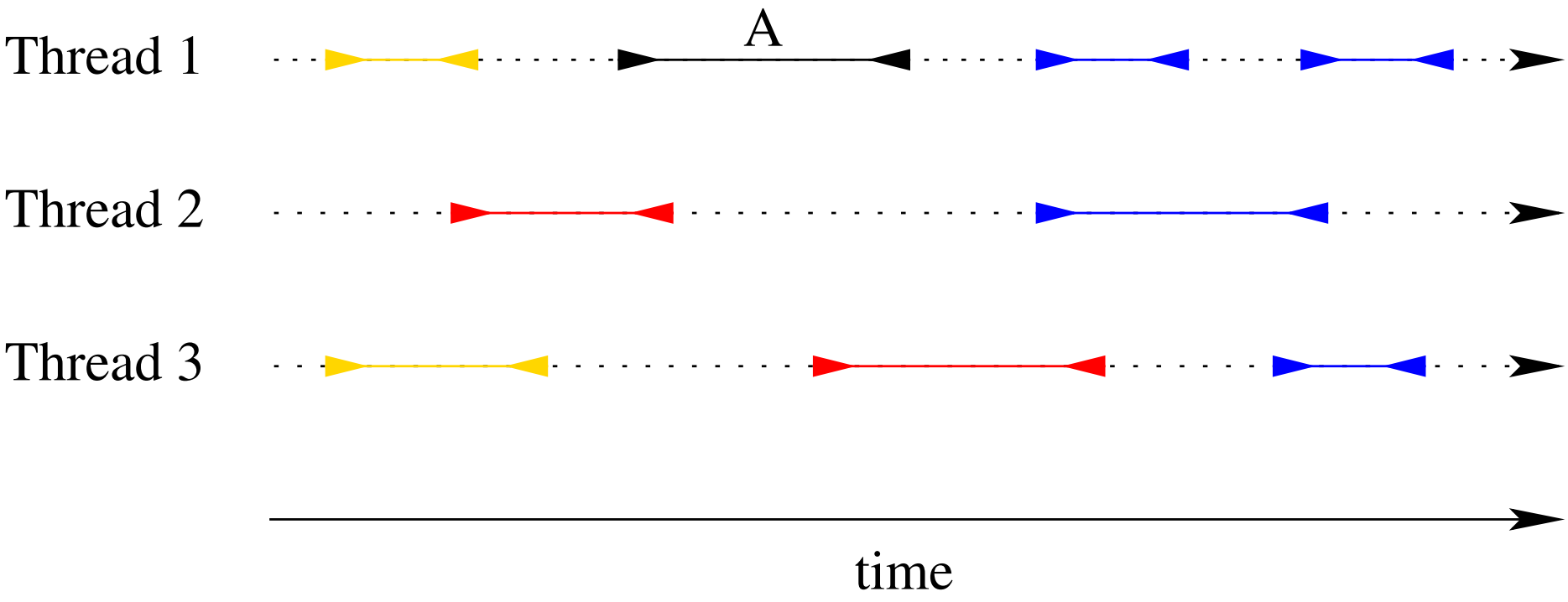


Strict Serializability

Strict serializability additionally requires that if transaction A completes before B starts, then A must occur before B in the sequential execution.

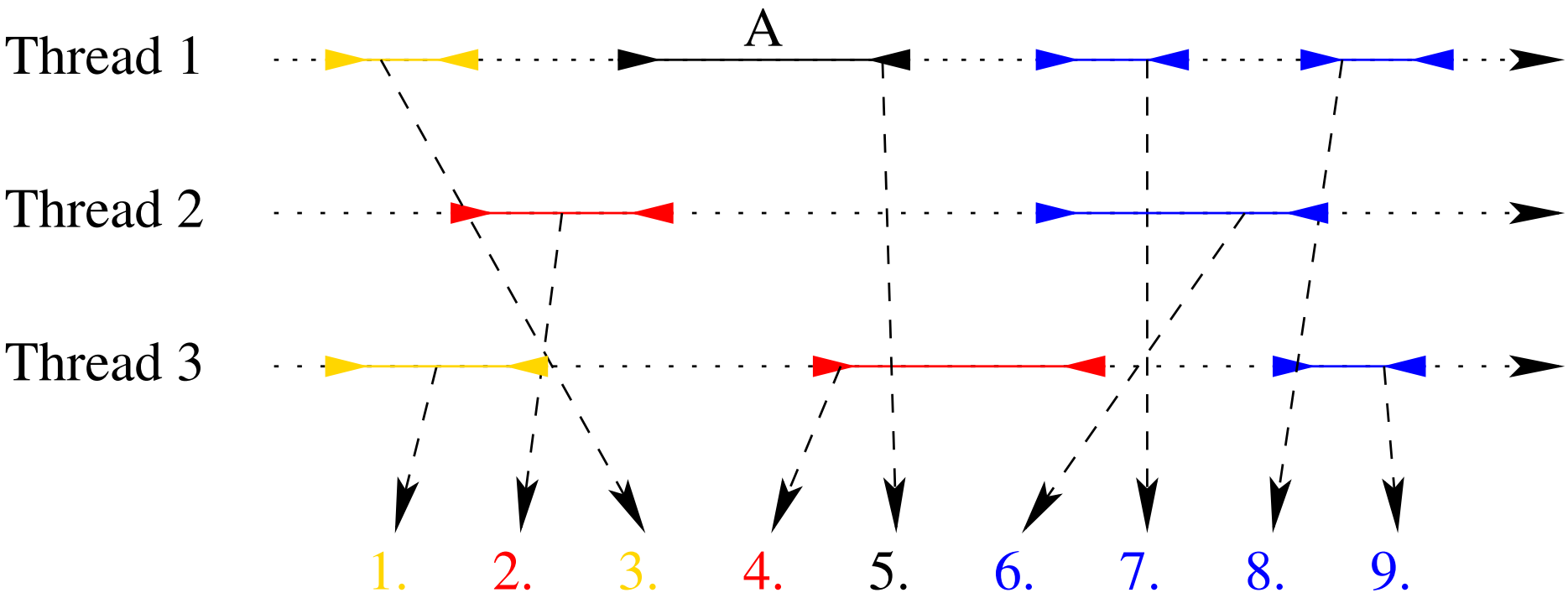
Strict serializability imposes real-time constraints. However, it does not consider non-transactional accesses.

Strict Serializability: Example



Definitely before A Could happen before or after A Definitely after A

Strict Serializability: Example



Linearizability

Linearizability requires that each transaction appears to take place atomically at some point between the beginning of its `StartTx` and the completion of its final `CommitTx` call.

Linearizability covers non-transactional accesses. However, it is not immediately clear how to specify the behavior of transactions that abort.

Consistency During Transactions

Strict serializability and linearizability specify how *committed* transactions behave. But what happens while a transaction runs?

For instance, consider

```
// Thread 1
do {
  StartTx();
  int u = ReadTx(&x);

  int v = ReadTx(&y);
  while (u!=v) {};
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  WriteTx(&x, 42);
  WriteTx(&y, 42);
} while (!CommitTx());
```

Zombie Transactions and Incremental Validation

Transactions that operate on inconsistent data, but where the conflict has not yet been detected by the TM implementation, are known as zombie transactions or doomed transactions.

A "manual" solution is to require that the programmer adds explicit validation operations to transactions where necessary (e.g., after a series of `ReadTx` operations) to ensure that zombie transactions are aborted.

Opacity

Alternatively, the TM implementation can provide stronger guarantees about the consistency of values read by a transaction.

Opacity is an extension of strict serializability where running and aborted transactions must also appear in the serial order (albeit without their effects being exposed to other threads).

With opacity a transaction is always guaranteed to operate on consistent data, by ensuring the transaction's read-set remains consistent.

Mixed-Mode Access

A semantics for TM must consider the interaction between transactional and non-transactional accesses to the same data.

- Weak isolation guarantees transactional semantics only among transactions.
- Strong isolation also guarantees transactional semantics between transactions and non-transactional code.

Strong isolation is much easier to program for, but difficult to provide in software TM implementations.

Problems With Weak Isolation

- Non-repeatable read: a transaction reads the same variable twice and observes an intermediate update by a non-transactional write
- Intermediate lost update: a non-transactional write interposes in a read-modify-write series executed by a transaction and is lost
- Intermediate dirty read: a non-transactional read observes an intermediate (non-committed) value written by a transaction
- Granular lost update: a non-transactional write is lost due to a transactional update to an adjacent memory location

Problems With Weak Isolation and Zombies

```
// Thread 1
do {
  StartTx();
  int u = ReadTx(&x);

  int v = ReadTx(&y);
  if (u != v) {
    WriteTx(&z, 10);
  }
} while (!CommitTx());
```

```
// Thread 2

do {
  StartTx();
  WriteTx(&x, 42);
  WriteTx(&y, 42);
} while (!CommitTx());
```

A concurrent non-transactional read from z might observe the value 10.

Problems With Weak Isolation (cont.)

```
// Thread 1

do {
  StartTx();
  WriteTx(&x_priv, 1);
} while (!CommitTx());
x = 10;
```

```
// Thread 2
do {
  StartTx();
  if (ReadTx(&x_priv) == 0)
    WriteTx(&x, 20);

} while (!CommitTx());
```

The non-transactional update of `x` to 10 may be lost when the transaction in Thread 2 is rolled back.

Problems With Weak Isolation (cont.)

```
// Thread 1

x = 42;
do {
  StartTx();
  WriteTx(&x_pub, 1);
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  int tmp = ReadTx(&x);

  if (ReadTx(&x_pub) == 1) {
    // use tmp
  }
} while (!CommitTx());
```

Thread 2 may miss the non-transactional update of x to 42.

TM Semantics and Memory Models

```
// Thread 1
do {
  StartTx();
  WriteTx(&data, 42);
  WriteTx(&ready, 1);
} while (!CommitTx());
```

```
// Thread 2
int r = ready;
int d = data;
if (r == 1) {
  // use d
}
```

Even with strong isolation, Thread 2 may observe `r==1` but `d!=42`. This depends on the underlying memory model of the programming language.

A Lock-based Semantics for TM

A simple model for defining the semantics of transactions is single-lock atomicity (SLA). Under this model, programs execute **as if** all transactions acquire a single, program-wide mutual exclusion lock.

Unfortunately, SLA is difficult to extend beyond basic transactions.

Moreover, SLA may not agree with our intuition about transactions:

```
// Thread 1
StartTx();
while (true) {};
CommitTx();
```

```
// Thread 2
StartTx();
int tmp = ReadTx(&x);
CommitTx();
```

Under SLA, Thread 2 might block forever while Thread 1 holds the lock.

Transactional Sequential Consistency

Transactional sequential consistency (TSC) is an extension of sequential consistency to TM.

It requires that *all* operations appear to happen in some serial order, and moreover that the operations of a transaction being attempted by one thread are not interleaved with *any* operations from other threads.

Transactional Data-Race Freedom

TSC is a strong consistency model. Practical TM implementations provide TSC only for race-free programs.

A program is transactional data-race free (TDRF) if it has no (ordinary or transactional) data races (assuming TSC).

Nesting

If we allow transactions to be nested, various semantics are possible.

- Flattened nesting: aborting the inner transaction causes the outer transaction to abort; committing has no effect until the outer transaction commits
- Closed nesting: aborting the inner transaction transfers control to the outer transaction; committing has no effect until the outer transaction commits
- Open nesting: aborting the inner transaction transfers control to the outer transaction; committing makes changes immediately visible (even if the outer transaction later aborts)

Performance and progress

Performance: Different Areas of Interest

Inherent concurrency: what is the optimal way in which a TM system could schedule the work (while providing specified semantics, e.g., TSC)?

Actual concurrency: how does the concurrency that is actually achieved compare to inherent concurrency?

Progress guarantees: are transactions guaranteed to make progress (e.g., is there some kind of fairness guarantee)?

Sequential overhead: in the absence of contention, how fast is a transaction when compared to non-transactional code?

Progress Guarantees

Is this program guaranteed to make progress?

```
// Thread 1
do {
  StartTx();
  WriteTx(&x, 1);
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  int tmp = ReadTx(&x);
  while (tmp == 0) {}
} while (!CommitTx());
```

The answer depends on the TM implementation.

Wasted Work vs. Lost Concurrency

```
// Thread 1
do {
  StartTx();
  WriteTx(&x, 1);
  // Long computation
  ...
  if (Prob(p)) {
    AbortTx();
  }
} while (!CommitTx());
```

```
// Thread 2
do {
  StartTx();
  WriteTx(&x, 1);
  // Long computation
  ...
  if (Prob(p)) {
    AbortTx();
  }
} while (!CommitTx());
```

If p is high, it is worthwhile to execute both transactions concurrently. Otherwise, it is best to run only one of the transactions at a time.

Contention Management

To mitigate poor performance caused by conflicts, TM implementations employ contention resolution policies.

Such a policy decides which one of the conflicting transactions should be aborted, and whether any of the transactions involved should be delayed.

There is no single best policy.

Conclusion

TM implementations face a trade-off between performance and semantic guarantees.

TM implementations face a trade-off between sequential performance and concurrency.

Different TM implementations make different choices. Many of the details remain open research questions.