Atomic Transactions

- Problems with Locks
- Transactions
- Serializability and Atomicity
- How to ensure Serializability
- Optimistic Concurrency Control
- Implementation: How to deal with aborting transactions

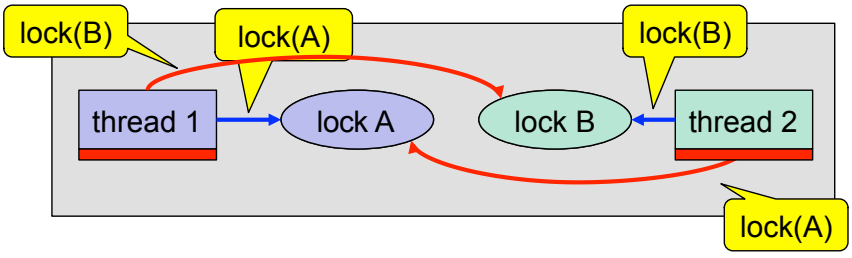
Locking has many Problems

- Locks reduce Concurrency

Locking has many Problems

- Locks reduce Concurrency
- Locks lead to Deadlocks

Locks lead to Deadlocks



Locking has many Problems

- Locks reduce Concurrency
- Locks lead to Deadlocks
- Locks lead to Priority Inversion

Locks lead to Priority Inversion

Mars Pathfinder Mission

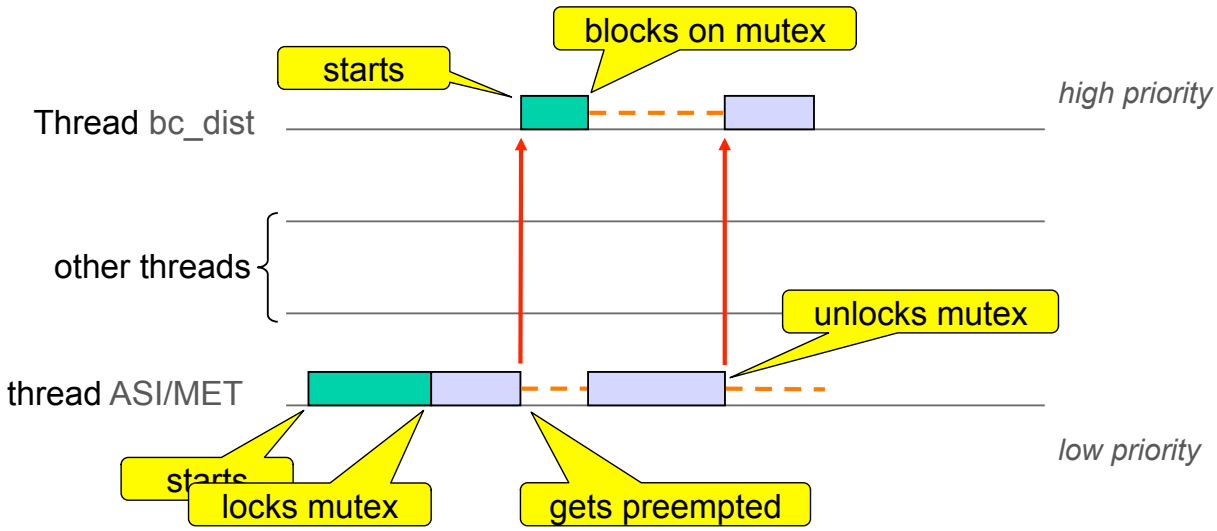
(NASA)

- Landing on July 4, 1997
- "experiences software glitches"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.

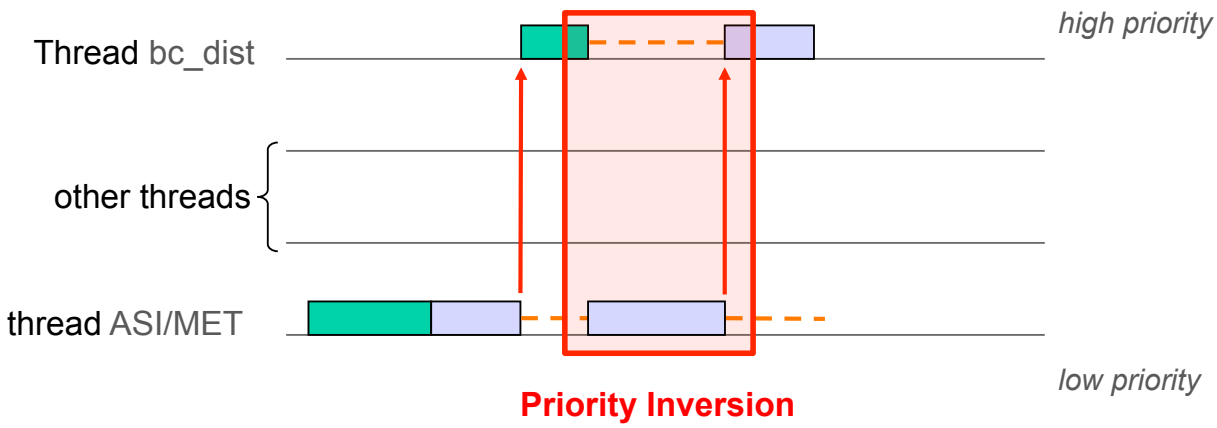
- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.

as reported by Mike Jones
https://www.microsoft.com/en-us/research/people/mbj/

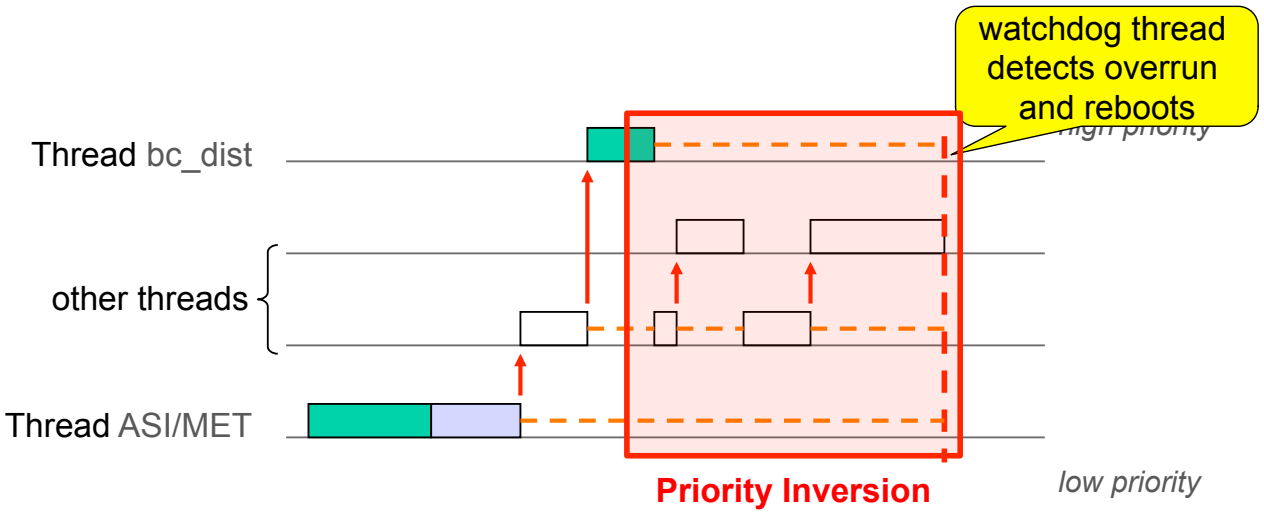
Priority Inversion on Mars Pathfinder



Priority Inversion on Mars Pathfinder



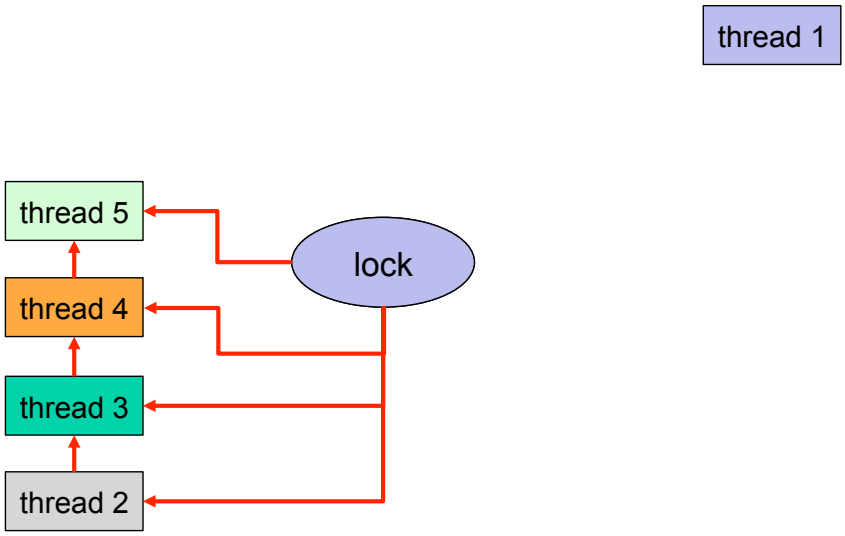
Priority Inversion on Mars Pathfinder



Locking has many Problems

- Locks reduce Concurrency
- Locks lead to Deadlocks
- Locks lead to Priority Inversion
- Locks lead to Convoying

Locks lead to Convoying



Locking has many Problems

- Locks reduce Concurrency
- Locks lead to Deadlocks
- Locks lead to Priority Inversion
- Locks lead to Convoying
- Locks lack "soft" Composability (locking "by Convention")

Locks lack "soft" Composability

C. J. Rossbach et al. :
"TxLinux: Using and Managing Hardware Transactional Memory in an Operating System", SOSP 2007

- In 2001 study of Linux bugs, 346 of 1025 bugs (34%) involved synchronization.
- 2003 study of Linux 2.5 kernel found 4 confirmed and 8 unconfirmed deadlock bugs.
- Locks have to be managed "by convention", which has to be documented in for each case in detail.
- Linux source file `mm/filemap.c` has a 50-line comment on the top of the file describing the lock ordering used in the file. The comment describes locks used at a calling depth of 4 from functions in the file.

Locks lack "soft" Composability

Another example in the Linux source code:

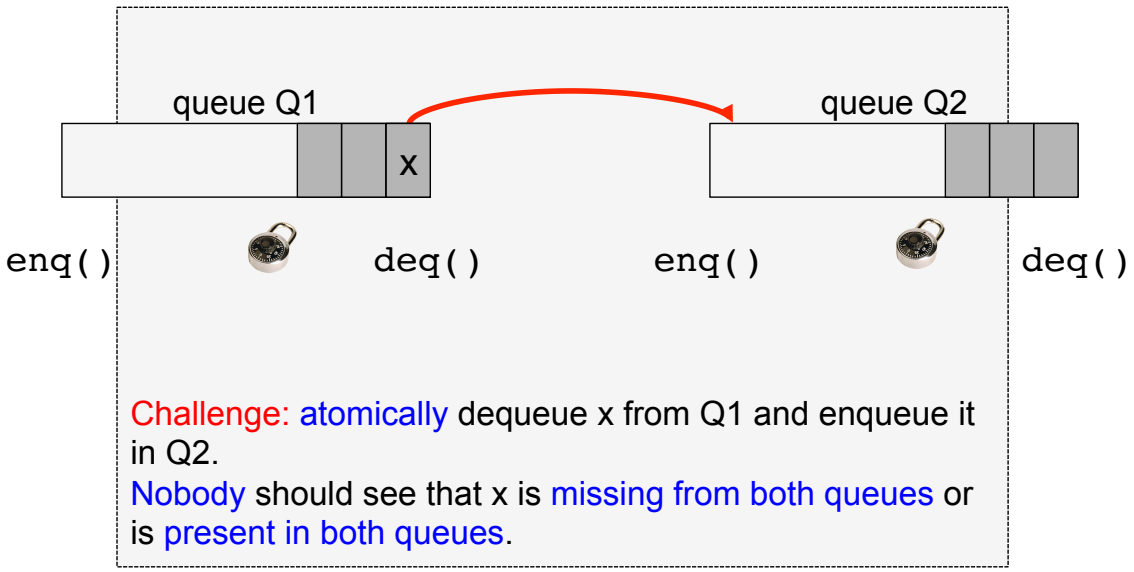
```
/*
 * When a locked buffer is visible to the I/O layer BH_Launder
 * is set. This means before unlocking we must clear BH_Launder,
 * mb() on alpha and then clear BH_Lock, so no reader can see
 * BH_Launder set on an unlocked buffer and then risk to deadlock.
 */
```

Linux kernel 2.4.19 `fs/buffer.c`

Locking has many Problems

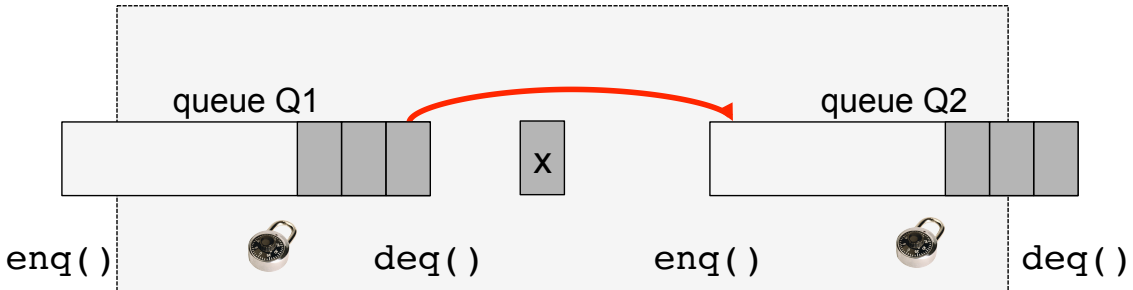
- Locks reduce Concurrency
- Locks lead to Deadlocks
- Locks lead to Priority Inversion
- Locks lead to Convoying
- Locks lack "soft" Composability (locking "by Convention")
- Locks lack "hard" Composability

Locks lack "hard" Composability



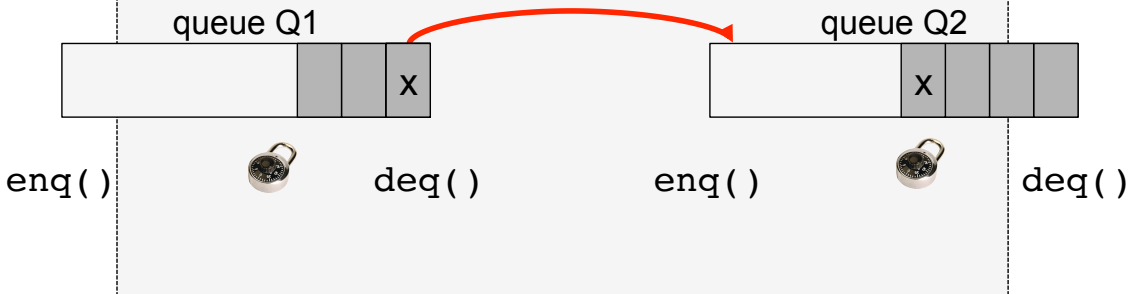
Challenge: atomically dequeue x from Q1 and enqueue it in Q2.
Nobody should see that x is missing from both queues or is present in both queues.

Locks lack "hard" Composability



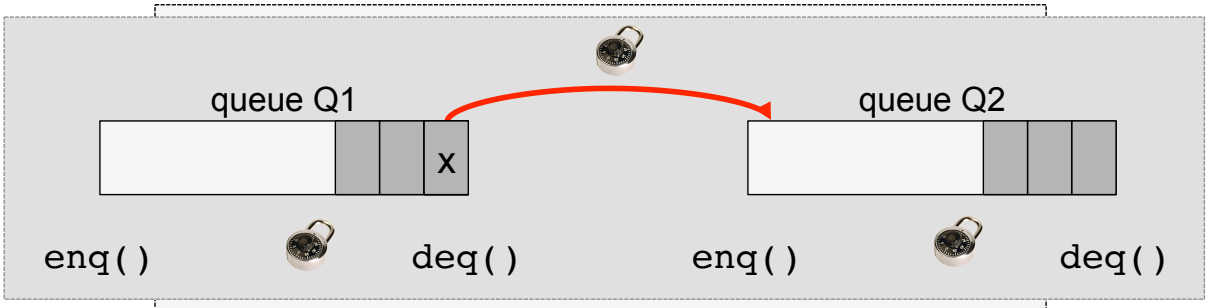
Challenge: atomically dequeue x from Q1 and enqueue it in Q2.
Nobody should see that x is missing from both queues or is present in both queues.

Locks lack "hard" Composability



Challenge: atomically dequeue x from Q1 and enqueue it in Q2.
Nobody should see that x is missing from both queues or is present in both queues.

Locks lack "hard" Composability



Challenge: atomically dequeue x from Q1 and enqueue it in Q2.
Nobody should see that x is missing from both queues or is present in both queues.

Locking has many Problems

- Locks reduce Concurrency
- Locks lead to Deadlocks
- Locks lead to Priority Inversion
- Locks lead to Convoying
- Locks lack "soft" Composability (locking "by Convention")
- Locks lack "hard" Composability

Alternative: Transactions

Transaction: Group of operations executed by a thread

```
transaction.begin()
  <operation>
  <operation>
  throw new TransAbortEx()
  <operation>
transaction.end()
```

Operations:

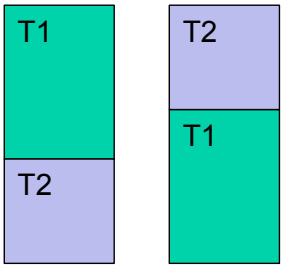
```
object.READ();

object.WRITE();
```

```
Transaction t;
t.begin();
  Object x = Q1.deq();
  Q2.enq(x);
t.end();
```

Atomicity Transaction-Style: Serializability

Observation: Atomicity is about how we execute operations of instructions.

Serial execution: Operations are scheduled one transaction at a time. One transaction executes after another other.

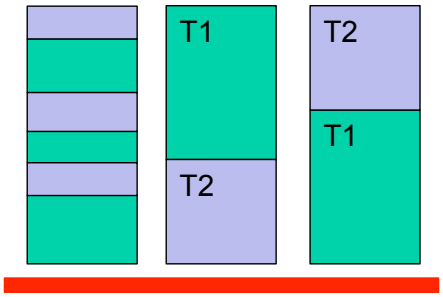


Atomicity Transaction-Style: Serializability

Observation: Atomicity is about how we execute operations of instructions.

Serial execution: Operations are scheduled one transaction at a time. One transaction executes after another other.

Serializable execution: Transactions **appear** to execute serially, one transaction at a time.



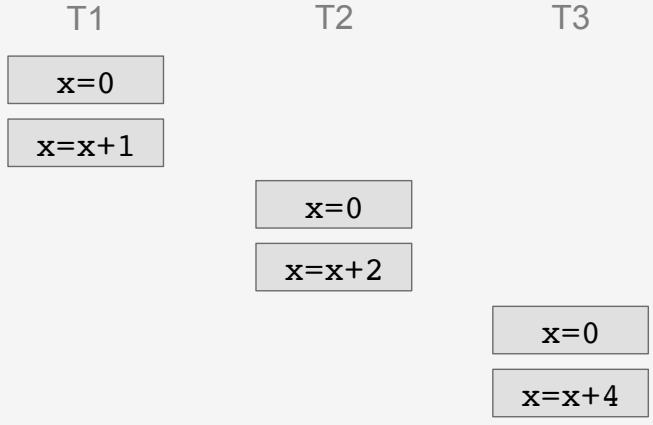
Serializability

T1:
```
begin_transaction;
  x = 0;
  x = x + 1;
end_transaction;
```

T2:
```
begin_transation;
  x = 0;
  x = x + 2;
end_transaction;
```

T3:
```
begin_transaction;
  x = 0;
  x = x + 4;
end_transaction;
```

Execution:



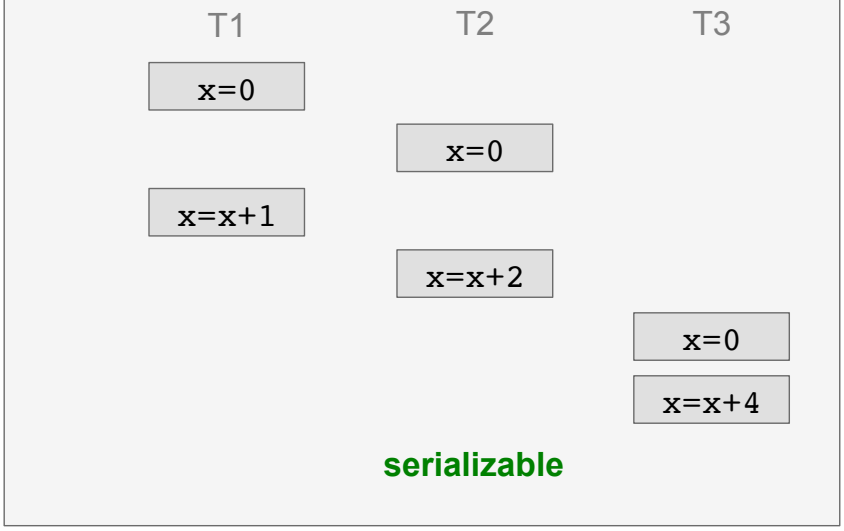
Serializability

```
T1:
begin_transaction;
  x = 0;
  x = x + 1;
end_transaction;
```

```
T2:
begin_transation;
  x = 0;
  x = x + 2;
end_transaction;
```

```
T3:
begin_transaction;
  x = 0;
  x = x + 4;
end_transaction;
```

Execution:



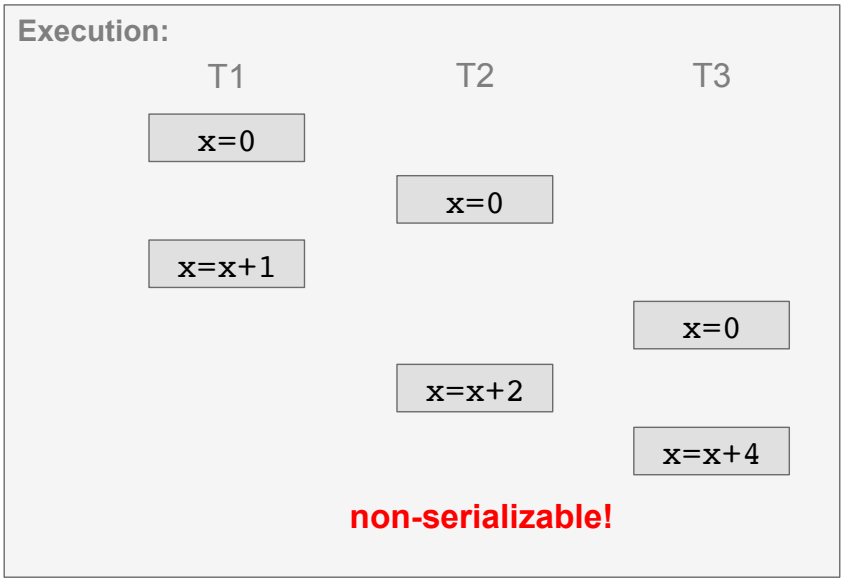
Serializability

```
T1:
begin_transaction;
  x = 0;
  x = x + 1;
end_transaction;
```

```
T2:
begin_transation;
  x = 0;
  x = x + 2;
end_transaction;
```

```
T3:
begin_transaction;
  x = 0;
  x = x + 4;
end_transaction;
```

Execution:



How to ensure Serializability

Basic idea:

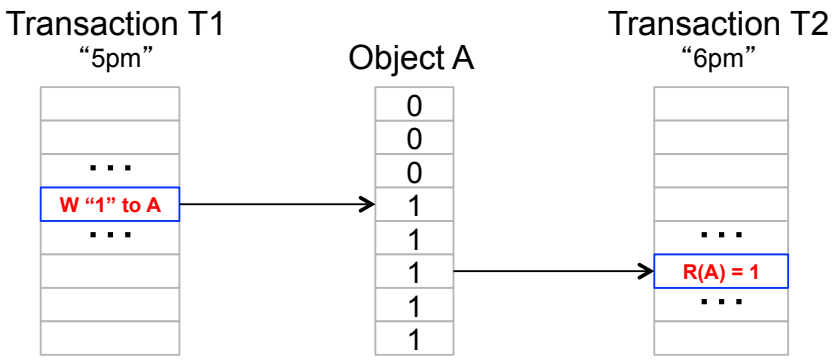
- Execute transaction without attention to serializability.
- Keep track of accessed objects.
- At commit point, check for conflicts with other transactions.
- Abort if conflicts occurred.

Approach:

?!

- Assign timestamp to each transaction.
- Make sure that execution has the same effect of a serial execution in order of assigned timestamps.

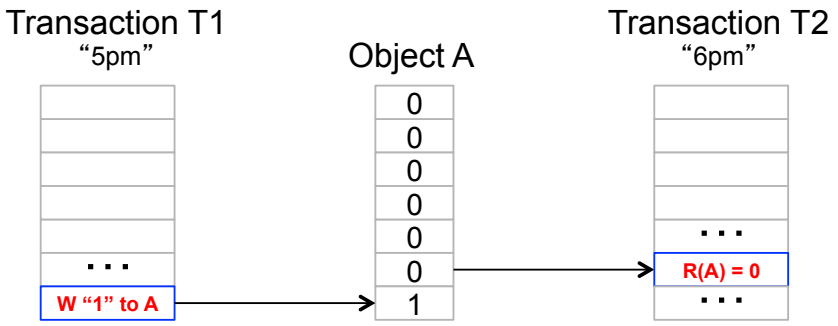
Ensure Serializability: Scenario 1



Q: Is this consistent with an execution where T1 executes a 5pm and T2 at 6pm?

A: YES, as T1 writes a "1" at 5pm and T2 reads the same "1" at 6pm.

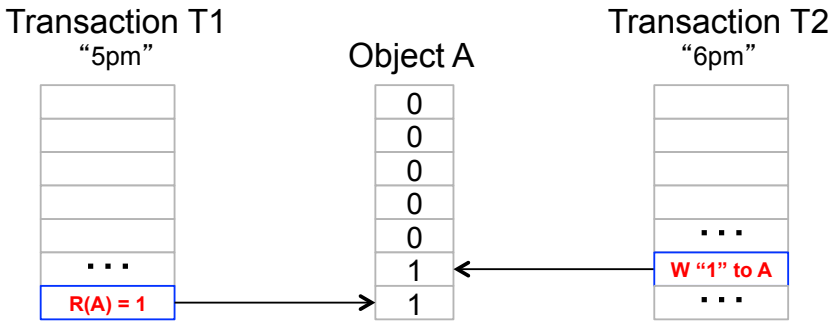
Ensure Serializability: Scenario 2



Q: Is this consistent with an execution where T1 executes a 5pm and T2 at 6pm?

A: NO, as T1 overwrites a value at 5pm that was "read already" by T2 at 6pm.

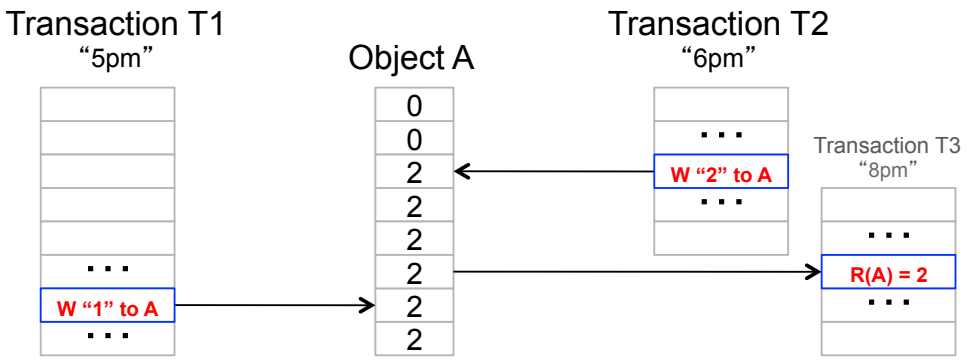
Ensure Serializability: Scenario 3



Q: Is this consistent with an execution where T1 executes a 5pm and T2 at 6pm?

A: NO, as T1 reads a value at 5pm that "would not be written until" 6pm.

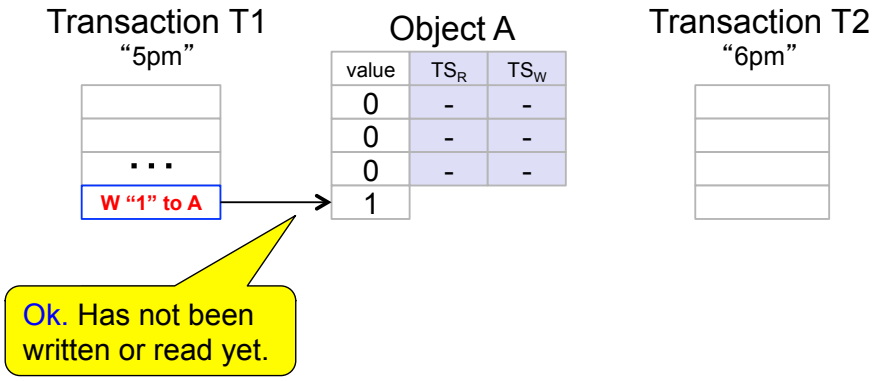
Ensure Serializability: Scenario 4



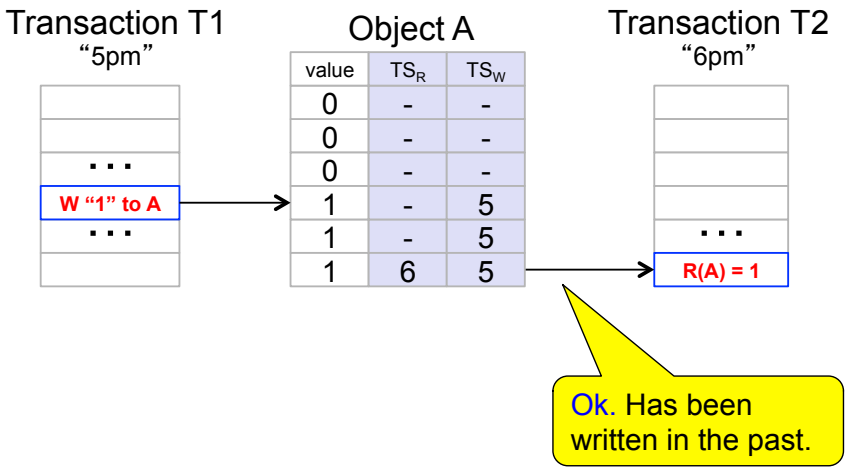
Q: Is this consistent with an execution where T1 executes a 5pm and T2 at 6pm?

A: YES, if we ignore the write "1" by T1.

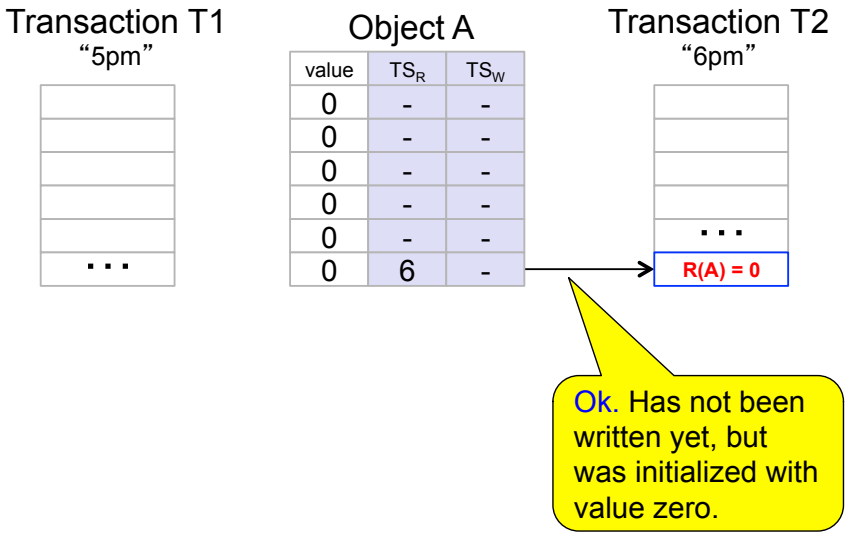
Ensure Serializability: Scenario 1



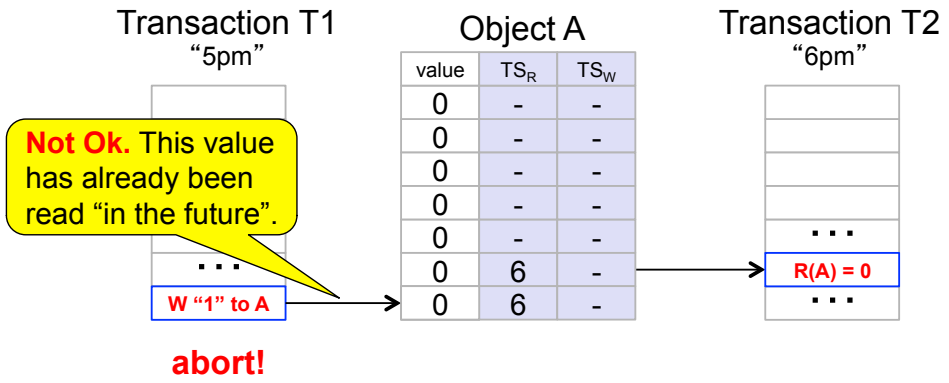
Ensure Serializability: Scenario 1



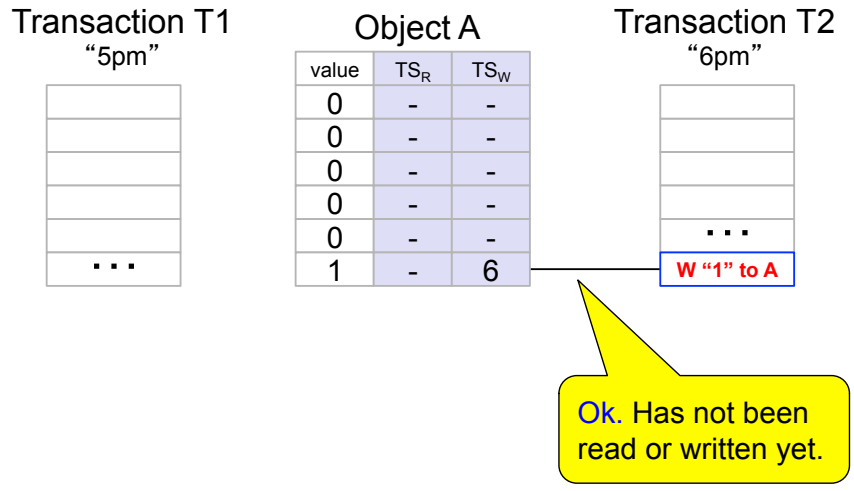
Ensure Serializability: Scenario 2



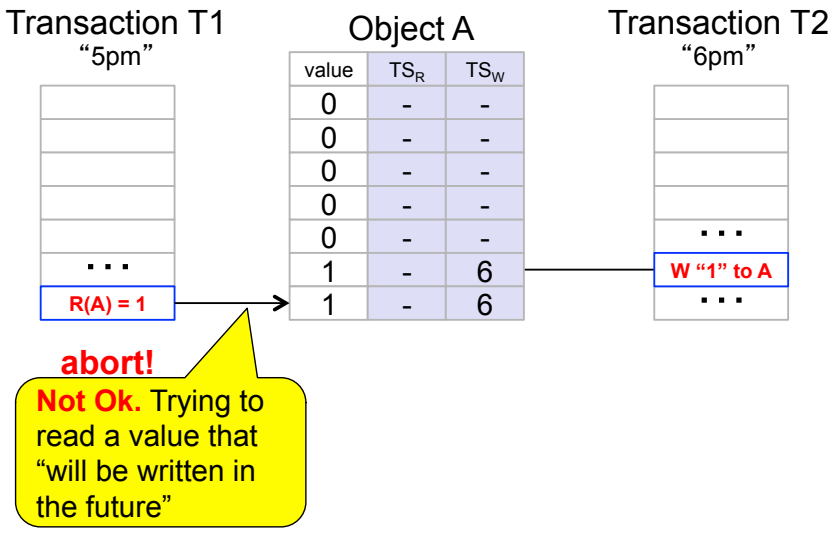
Ensure Serializability: Scenario 2



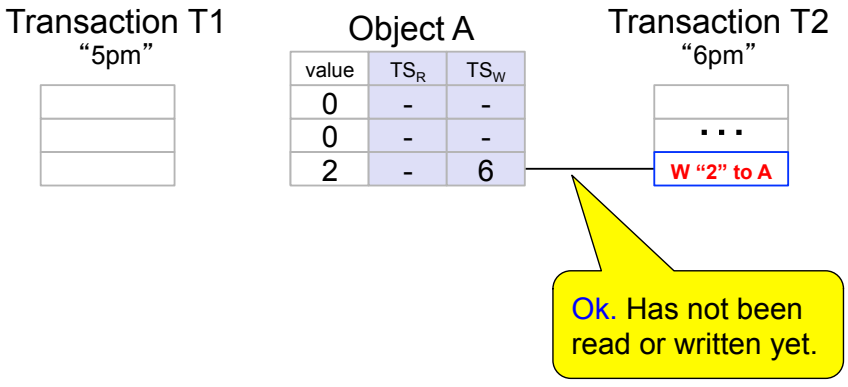
Ensure Serializability: Scenario 3



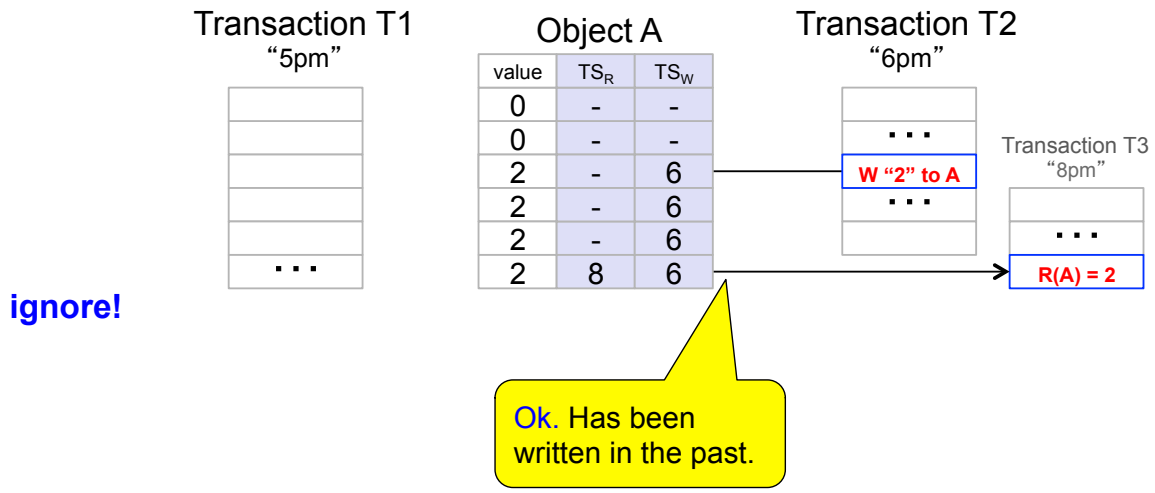
Ensure Serializability: Scenario 3



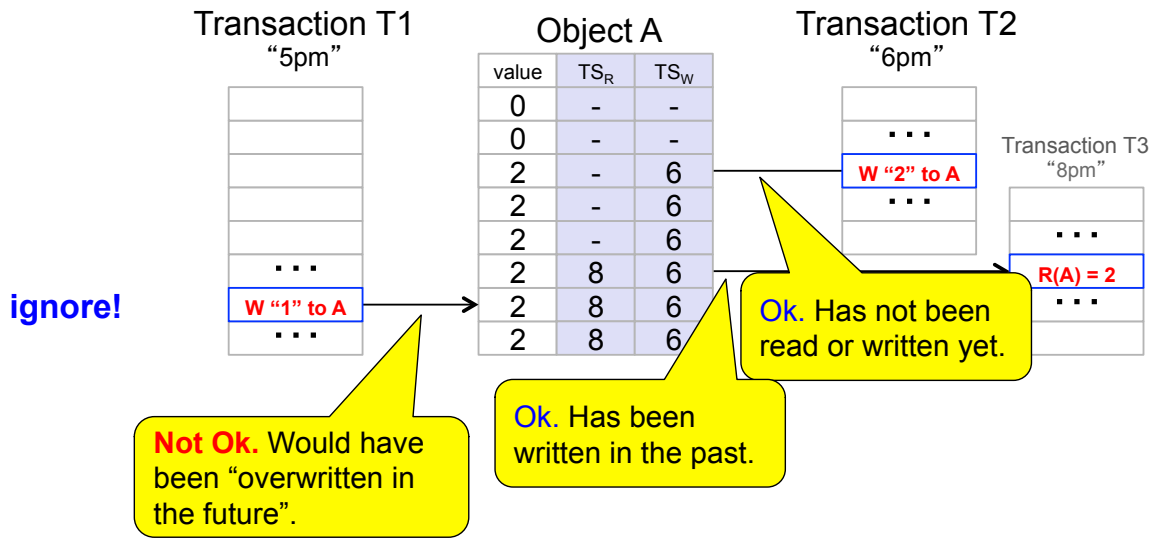
Ensure Serializability: Scenario 4



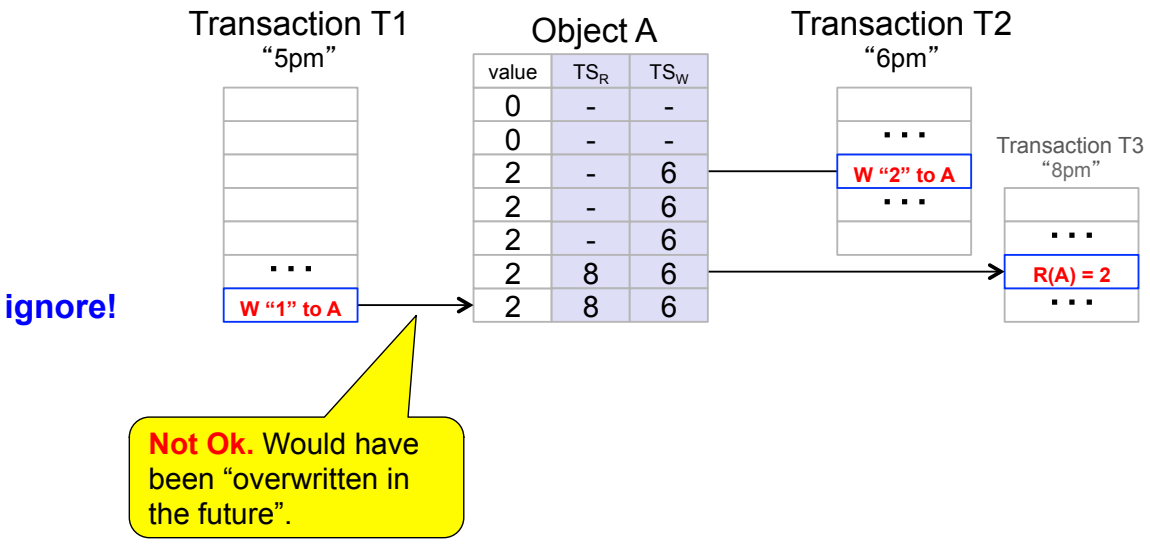
Ensure Serializability: Scenario 4



Ensure Serializability: Scenario 4



Ensure Serializability: Scenario 4



Timestamp-based Optimistic Conc. Control

Objects are tagged with read-time and write-time.

1. Transaction cannot **read** value of object if that value has not been written until after the transaction executed.
   Transaction with T.S. t_1 cannot read item with write-time t_2 if $t_2 > t_1$.
   (abort and try with new timestamp)
2. Transaction cannot **write** object if object has value read at later time.
   Transaction with T.S. t_1 cannot write item with read-time t_2 if $t_2 > t_1$.
   (abort and try with new timestamp)

Other possible conflicts:

- Two transactions can read the same item at different times.
- Ignore write of transaction with T.S. t_1 that wants to write to item with write-time t_2 if $t_2 > t_1$

Timestamp-Based Conc. Control

Rules for preserving serial order using timestamps:

a) Perform the operation X

```
if X    == READ   and t >= t_w
or if X == WRITE  and t >= t_r and t >= t_w

  if X == READ : set t_r = t if t > t_r
  if X == WRITE: set t_w = t if t > t_w
```

b) Do nothing

```
if X == WRITE and t_r <= t < t_w
```

c) Abort transaction

```
if X == READ and t < t_w
or X == WRITE and t < t_r
```

Dealing with Aborting Transactions

How to maintain information for not-yet committed transactions: "Prepare for aborts"

- transactions use a **private workspace**
- "**read set**" of versions of objects that have been read.
- "**write set**" of tentatively written objects.
- threads **commit** object values into memory when transaction is "done"
- if transaction aborts, thread releases read and write sets.

Atomic Transactions

- Problems with Locks
- Transactions
- Serializability and Atomicity
- How to ensure Serializability
- Optimistic Concurrency Control
- Implementation: How to deal with aborting transactions