# Concurrency Control 1

R&G Chapter 17

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

# Reminders

- Midterm on Monday.
  - Questions similar to the homeworks.
  - Closed book.
  - Homework 1-4 solutions posted tonight.
- No homework this week.

# Recap: Transactions

- OLTP Workloads: Lots of data updates.
- Transactions group database operations.
  - … is a sequence of reads and writes.
  - … is executed <u>atomically</u>.
  - … preserves integrity constraints.
    - … even if the system crashes.

# Crash Recovery Preview

- The ARIES recovery algorithm (3 Phases)
  - **Analysis**: Scan through the log to identify all xacts active at time of crash and all dirty pages in the buffer pool.
  - **Redo**: Redo all updates to dirty pages in the buffer pool to ensure that logged updates are carried out.
  - **Undo**: Use 'before' value from log to cancel out the writes of all xacts that were active at the time of crash. (need to guard vs crashing during recovery)

# Schedules

- An order in which a sequence of operations are executed.
- A schedule for a set of transactions is an interleaving of all operations in the transaction.
  - The only restriction is that operations within a transaction are executed in order.

# Recap: Scheduling

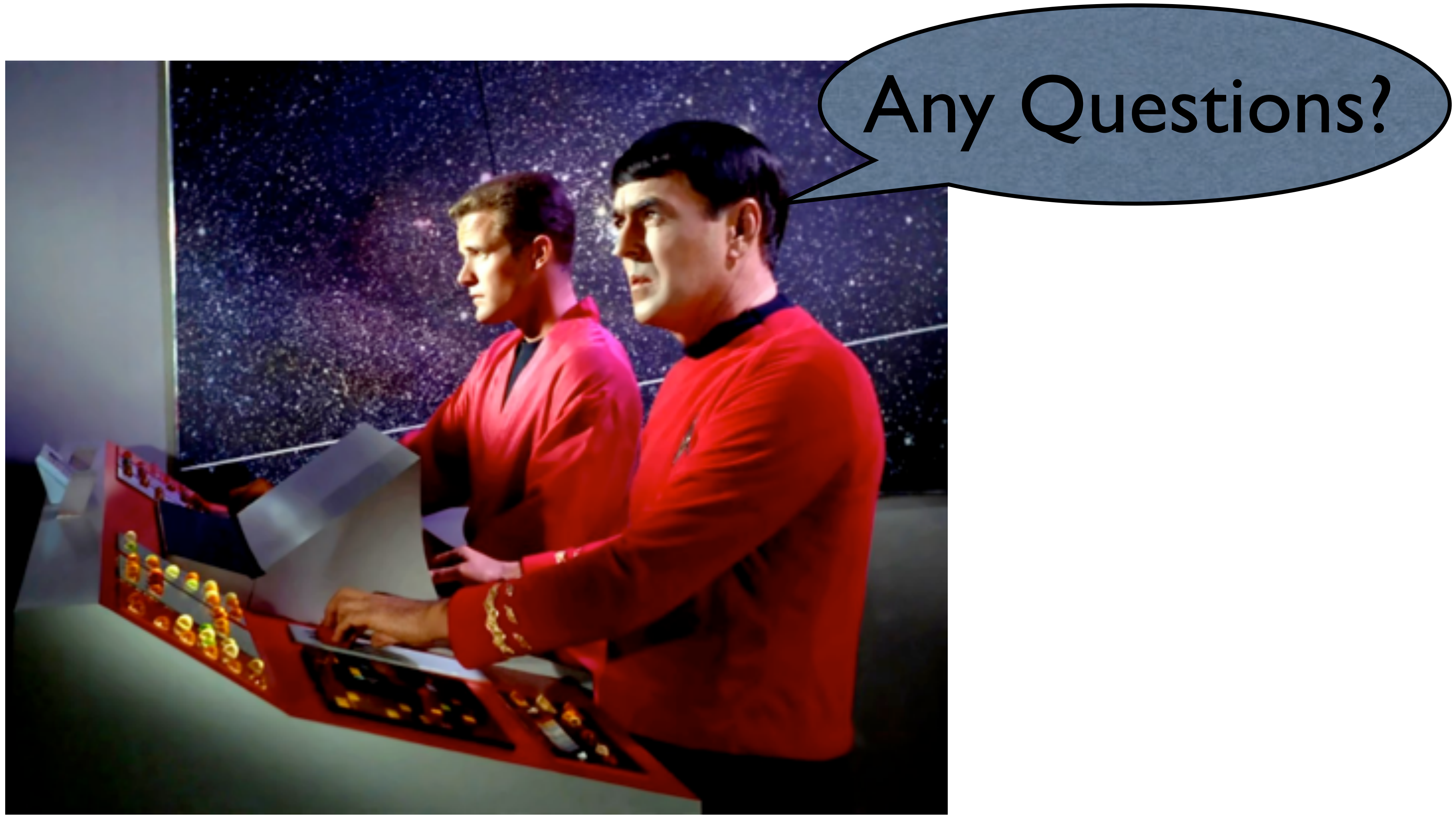
- **Serial Schedule**: A schedule that does not interleave the actions of different transactions.
- (Two) **Equivalent Schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second.
- **Serializable Schedule**: A schedule that is equivalent to some serial execution of the transactions.
  - If the transactions preserve consistency, every serializable schedule does too.

# Ensuring Serializable Schedules

- **Option 1**: Use reader/writer locks.
  - … latency/throughput cost (pessimistic).
- **Option 2**: Abort conflicting transactions.
  - … fairness/throughput cost (optimistic).
- **Option 3**: Predeclare reads/writes, and schedule operations before execution.
  - … flexibility cost.

# Ensuring Serializable Schedules

- **Option 1**: Use reader/writer locks.
  - … latency/throughput cost (pessimistic).
- **Option 2**: Abort conflicting transactions.
  - … fairness/throughput cost (optimistic).
- **Option 3**: Predeclare reads/writes, and schedule operations before execution.
  - … flexibility cost.


Any Questions?

How do we define correctness precisely?

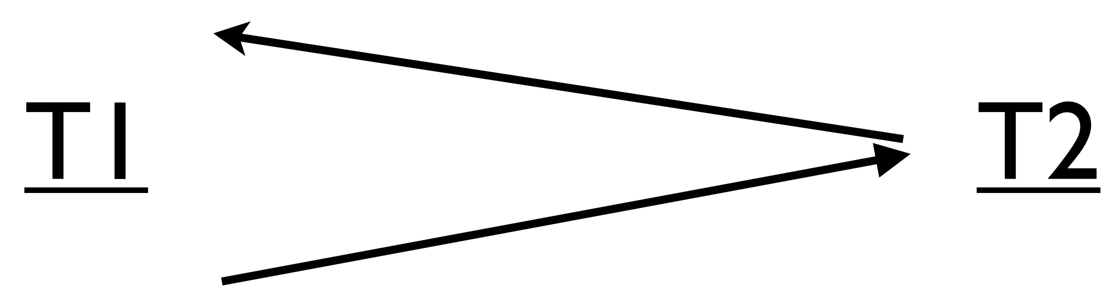
# Conflict Serializability

- Two schedules are conflict equivalent if:
  - … they involve the same set of operations from the same set of transactions.
  - … every pair of conflicting actions between two transactions is ordered in the same way.
- Not quite the same as regular equivalence (it's stronger).
  - Defined over operations rather than possible effects.
- A schedule is conflict serializable if it is conflict-equivalent to some serial schedule.

# Example

| Time | T1 | T2 |
|---|---|---|
| | `R(A)` | |
| | `W(A)` | |
| | | `R(A)` |
| | | `W(A)` |
| | | `R(B)` |
| | | `W(B)` |
| | `R(B)` | |
| | `W(B)` | |

# Example






Friday, March 1, 13

Ordering is bad. Bank loses 12 bucks.

# Example

$$\begin{tikzcd} \underline{T1} \arrow[r, shift right] & \underline{T2} \arrow[l, shift right] \end{tikzcd}$$

(Dependency Graph)





Ordering is bad.  Bank loses 12 bucks.

# Example


(Dependency Graph)

The cycle in the graph reveals a problem:
T1's output depends on T2 and visa versa.

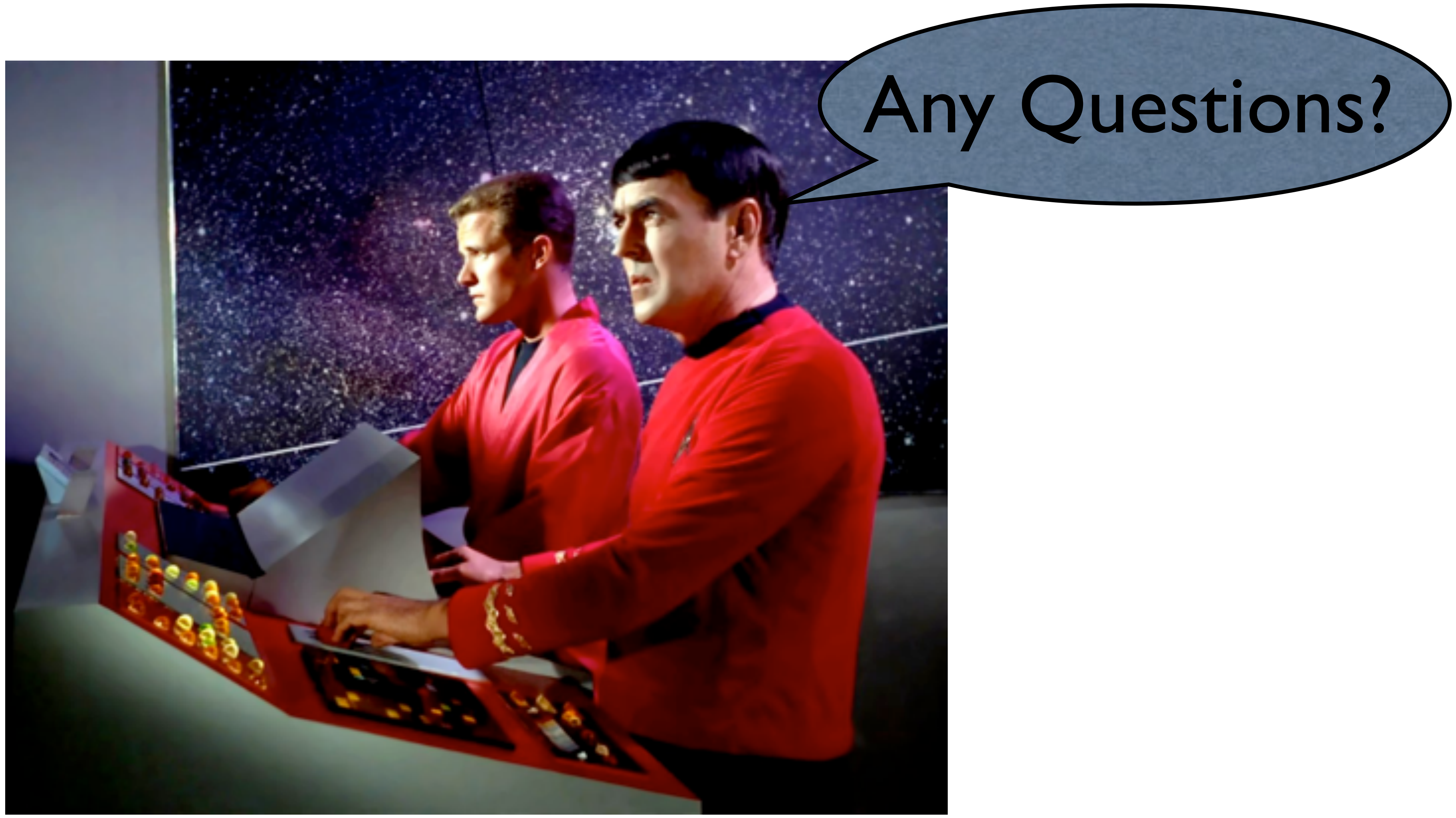




Ordering is bad. Bank loses 12 bucks.

# Dependency Graphs

- One <u>node</u> per transaction
- One <u>edge</u> (from $T_i$ to $T_k$) if $T_k$ reads/writes an object most recently written by $T_i$.
- **Claim**: A schedule is conflict serializable if and only if its dependency graph is <u>acyclic</u>.


Any Questions?

# View Serializability

- Schedules in terms of pre- and post-conditions.
- Schedules S1 and S2 are view-equivalent if:
  - … $T_i$ reads the initial value of A in S1, then it reads the initial value of A in S2.
  - … $T_i$ reads a value of A written by $T_k$ in S1 then it reads the value of A written by $T_k$ in S2.
  - …$T_i$ writes the final value of A in S1 then $T_i$ writes the final value of A in S2.

# Example

| Time | <u>T1</u> | <u>T2</u> | <u>T3</u> |
|---|---|---|---|
| | `R(A)` | | |
| | `W(A)` | | |
| | | `W(A)` | |
| | | | `W(A)` |

# Example

| Time | T1 | T2 | T3 |
| --- | --- | --- | --- |
| | R(A) | | |
| | | W(A) | |
| | W(A) | | |
| | | | W(A) |

# Example

Write order irrelevant
(T3 overwrites either way)

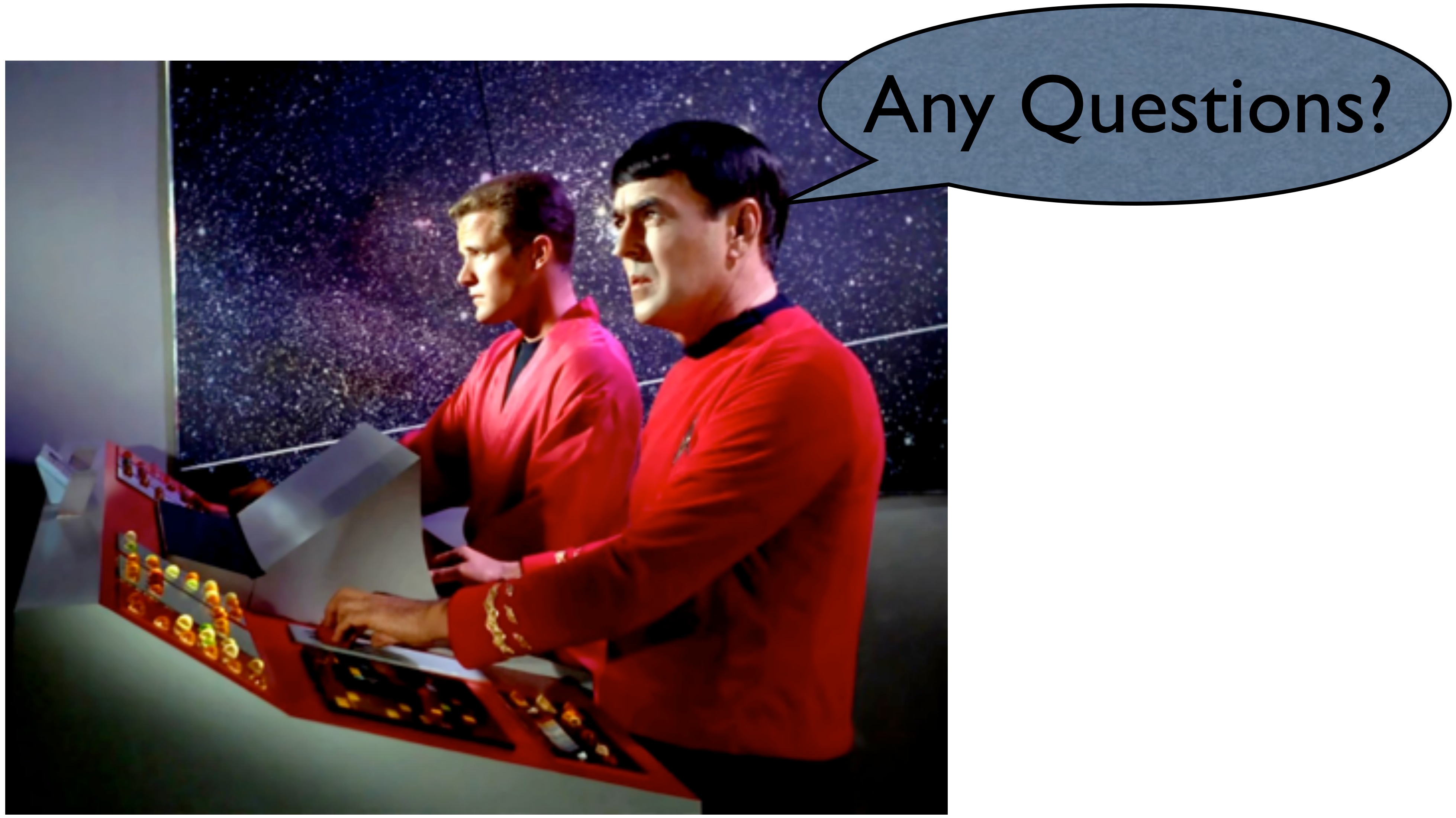
| Time | T1 | T2 | T3 |
|---|---|---|---|
| | `R(A)` | | |
| | | `W(A)` | |
| | `W(A)` | | |
| | | | `W(A)` |


Any Questions?

# Implementing Locks

# Take 1: 2-Phase Locking

- To Recap:
  - Obtain reader/writer locks on objects.
  - Once an xact releases a lock, it can no longer acquire any new locks.
- **Claim**: 2-phase locking allows only conflict-serializable schedules.
  - If xact A modifies a value, no other xact can read/modify that value until A 'completes'.


Any Questions?

# Lock Management

- Lock Manager (sometimes part of the Transaction manager) handles lock/unlock requests.
- Lock table entry:
  - # of transactions currently holding a lock
  - Type: Shared (reader) or Exclusive (writer) Lock
  - Pointer to <u>queue</u> of lock requests
- Locking/Unlocking implemented as atomic ops
- Lock Upgrades: Convert Shared Lock into Exclusive.

# Deadlocks

- Deadlock: A cycle of transactions waiting on each other's locks
    - Problem in 2PL; xact can't release a lock until it completes
- How do we handle deadlocks?
    - **Anticipate**: Prevent deadlocks before they happen.
    - **Detect**: Identify deadlock situations and abort one of the deadlocked xacts.

# Deadlock Detection

- **Baseline**: If a lock request can not be satisfied, the transaction is blocked and must wait until the resource is available.
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from $T_i$ to $T_k$ if $T_i$ is waiting for $T_k$ to release a lock.
- Periodically check for cycles in the graph.

# Example

| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |

# Example

| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |

**T1** **T2**

**T4** **T3**

# Example

| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |

**T1** **T2**

**T4** **T3**

# Example

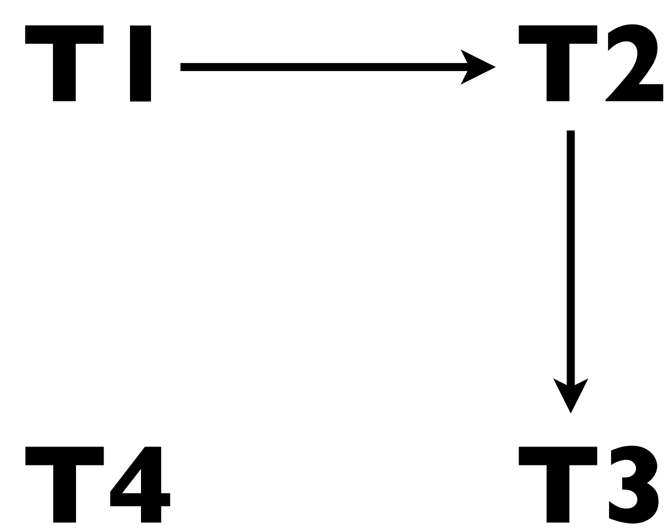
| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |

**T1** → **T2**

**T4** **T3**

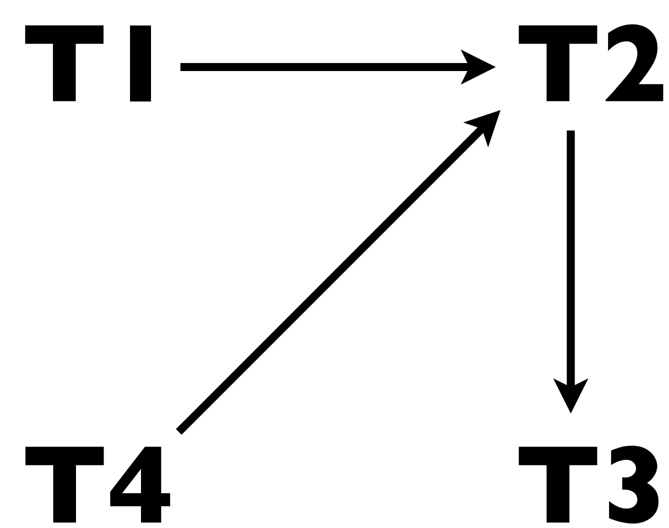
# Example

| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |



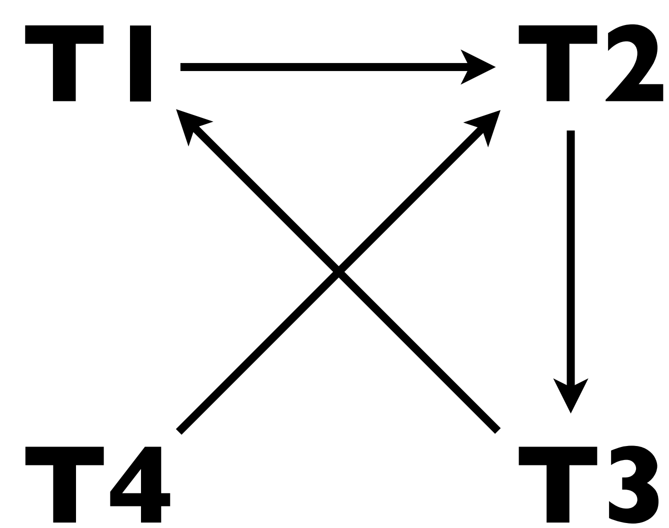
# Example

| Time | T1 | T2 | T3 | T4 |
| --- | --- | --- | --- | --- |
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |



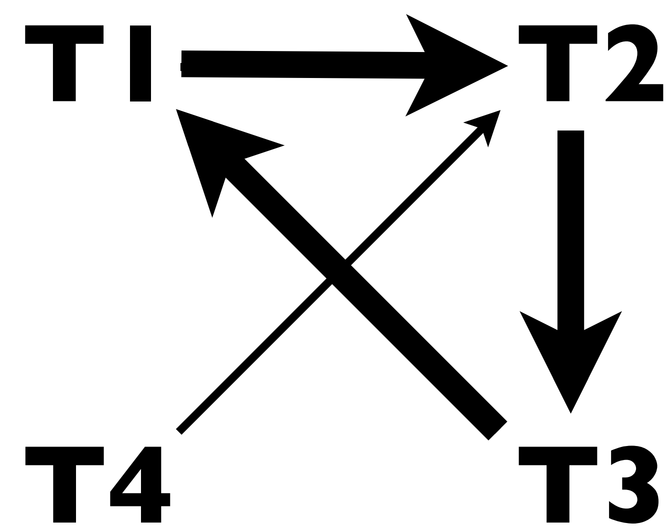
# Example

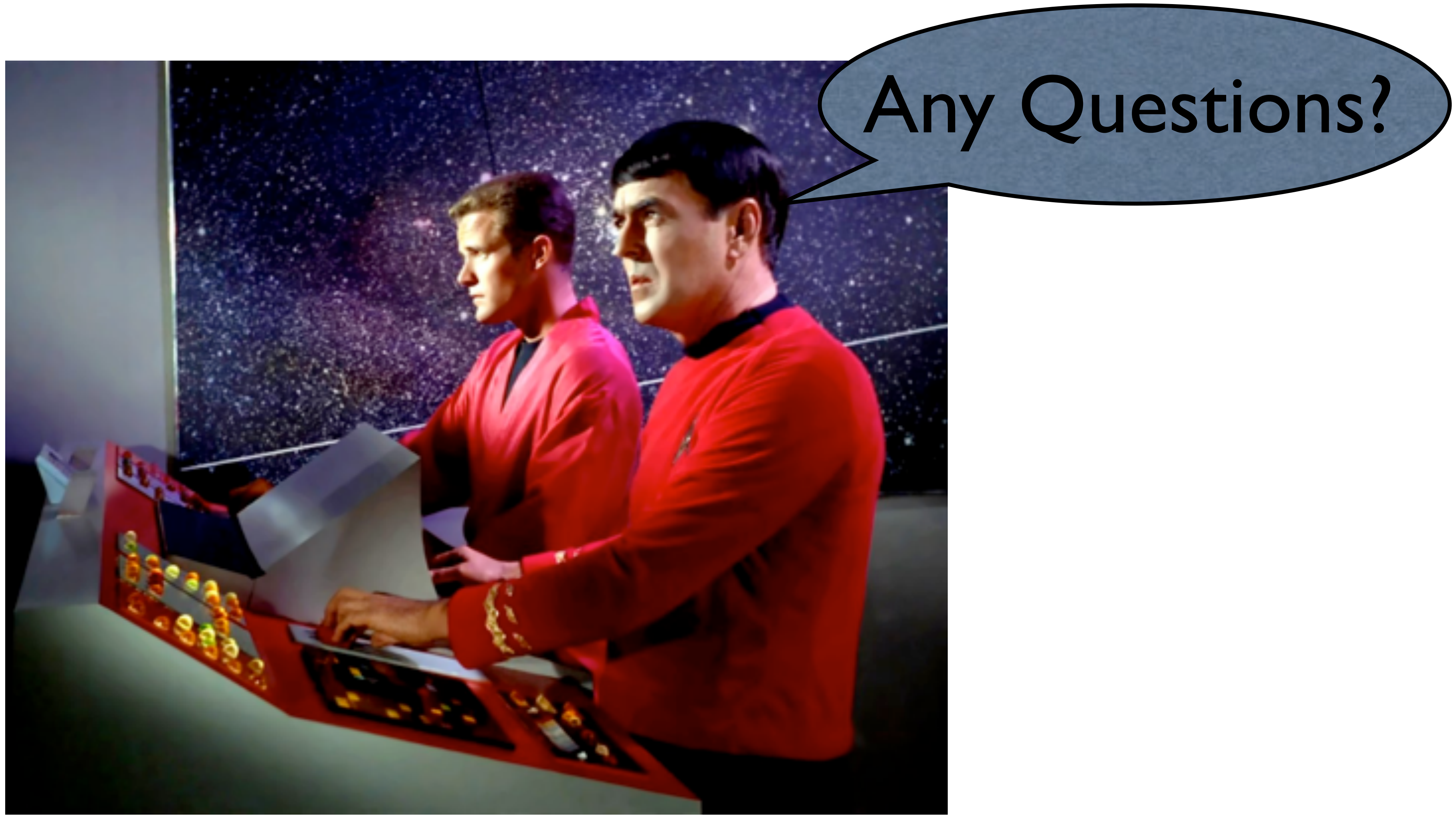
| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |



# Example

| Time | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| | S(A) | | | |
| | R(A) | | | |
| | | X(B) | | |
| | | W(B) | | |
| | S(B) | | | |
| | | | S(C) | |
| | | | R(C) | |
| | | X(C) | | |
| | | | | X(B) |
| | | | X(A) | |




Any Questions?

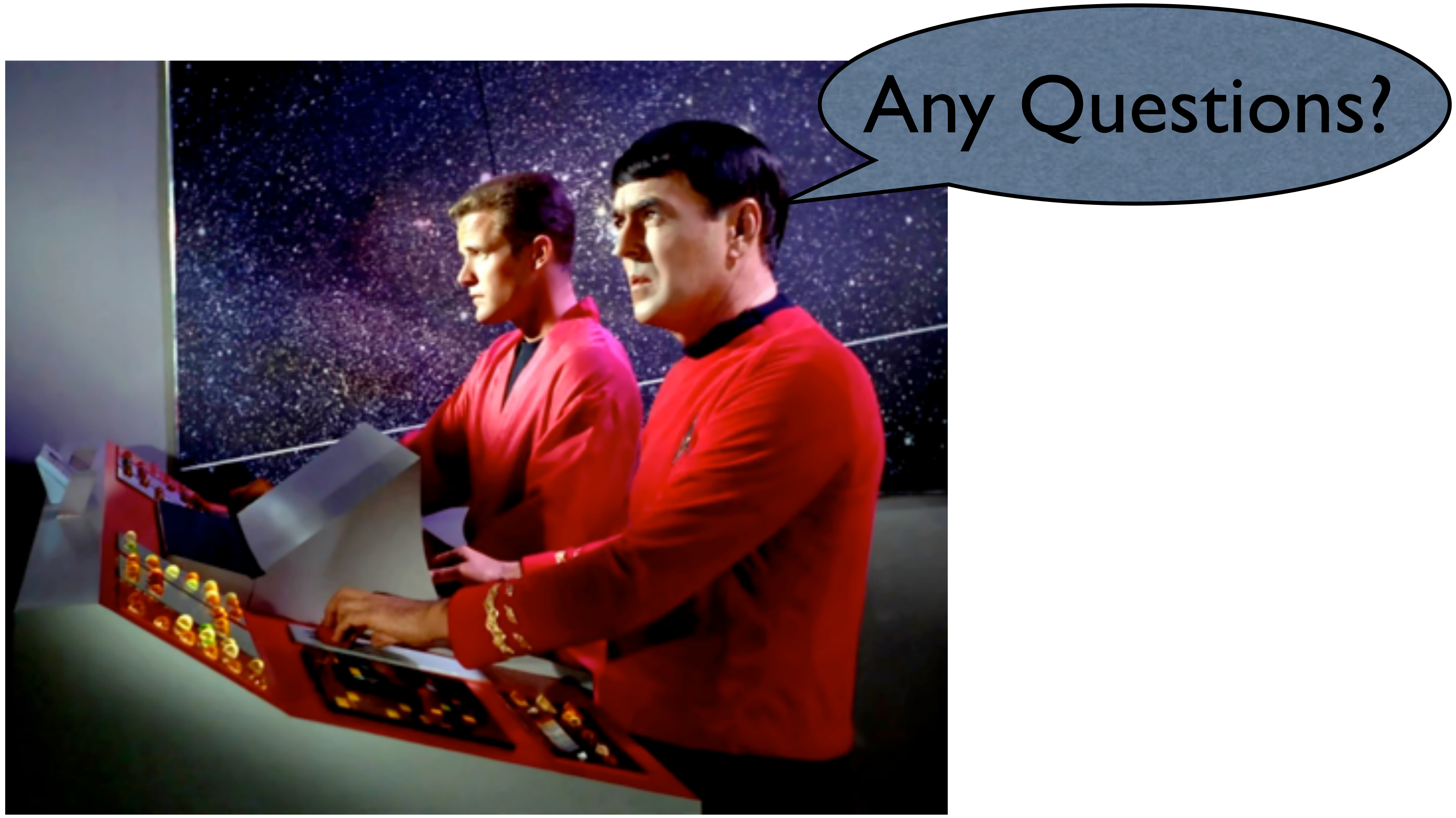
# Deadlock Prevention

- Ensure that dependencies are monotonic (and consequently acyclic)
- Assign each transaction a priority based on the timestamp at which it starts.
- When a transaction fails to acquire a lock:
  - Wait if monotonicity would be preserved.
  - Kill one transaction otherwise.

# Deadlock Prevention

- Policy 1 (Wait-Die): If $T_i$ has a higher priority, wait for $T_k$, otherwise $T_i$ aborts.
- Policy 2 (Wait-Wound): If $T_i$ has a higher priority, $T_k$ aborts, otherwise $T_i$ waits.

- Protect fairness by restarting the aborted transaction with its original timestamp.


Any Questions?