Transaction Concept

- Definition: A transaction is a operation or a collection of operations that accesses and possibly updates various data items
- Ex: Funds transfer in which one account (say *A)* is debited and another account (say *B)* is credited

- The database system has to maintain the following properties (ACID) for the transactions:

- Atomicity: Either all operations of the transaction are reflected properly in the database, or none
  --Debiting & Crediting in funds transfer

- Consistency: Execution of the transaction must preserve the consistency of the database

  --Sum of *A* + *B* must be preserved after execution of funds transfer

- Isolation: Each transaction is unaware of other transactions executing concurrently in the system

  --When T_1 and T_2 executing concurrently, T_1 is unaware of T_2 is executing and T_2 is unaware of T_1 is executing

- Durability: After transaction completes successfully, the changes it has made to the database persist, even if there are system failures

  --New values of accounts *A* and *B* must persist in funds transfer

- Programmer is responsible for ensuring consistency

- Transaction management component is responsible for ensuring atomicity

- Recovery-management component is responsible for ensuring durability

- Concurrency control component is responsible for ensuring isolation

- Let T_i be a transaction that transfers $50 from account A to account B

- Writing of transaction

T_i : read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B)

Transaction State

- While executing transaction, it goes to different states
- A transaction that completes its execution successfully is called committed
- If a transaction does not complete its execution successfully is called failed
- Failed transaction has to roll back
- Roll back means undo the changes made by the transaction

- Once a transaction has committed, we cannot undo changes

- A transaction must be in one of the following states:

- **Active,** the initial state; the transaction stays in this state while it is executing

- **Partially committed,** after the final statement has been executed

- **Failed,** when there is failure of transaction

- **Aborted,** after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

- **Committed,** after successful completion of transaction

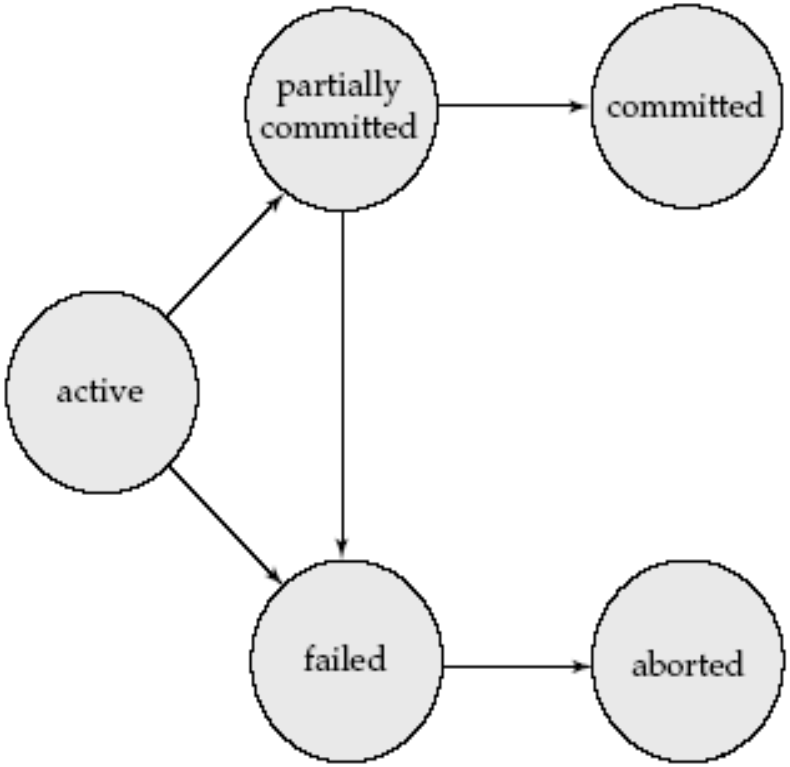


Figure 15.1 State diagram of a transaction.

- A transaction starts in the **active** state

- When it finishes its final statement, it enters the **partially committed** state

- At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may prevent its successful completion

- After writing information to the disk, the transaction enters the **committed** state

- A transaction enters the **failed** state when there is a failure

- Failed transaction must be rolled back. Then, it enters the **aborted** state

- At this point, the system has two options:

1.It can **restart** the transaction

2. It can **kill** the transaction

Implementation of Atomicity and Durability

- The transaction-management component and recovery-management component of a database system can support atomicity and durability by a variety of schemes

shadow copy scheme: It is based on making copies of the database, called shadow copies

- The scheme assumes that the database is simply a file on disk

- A pointer called db-pointer is maintained on disk; it points to the current copy of the database

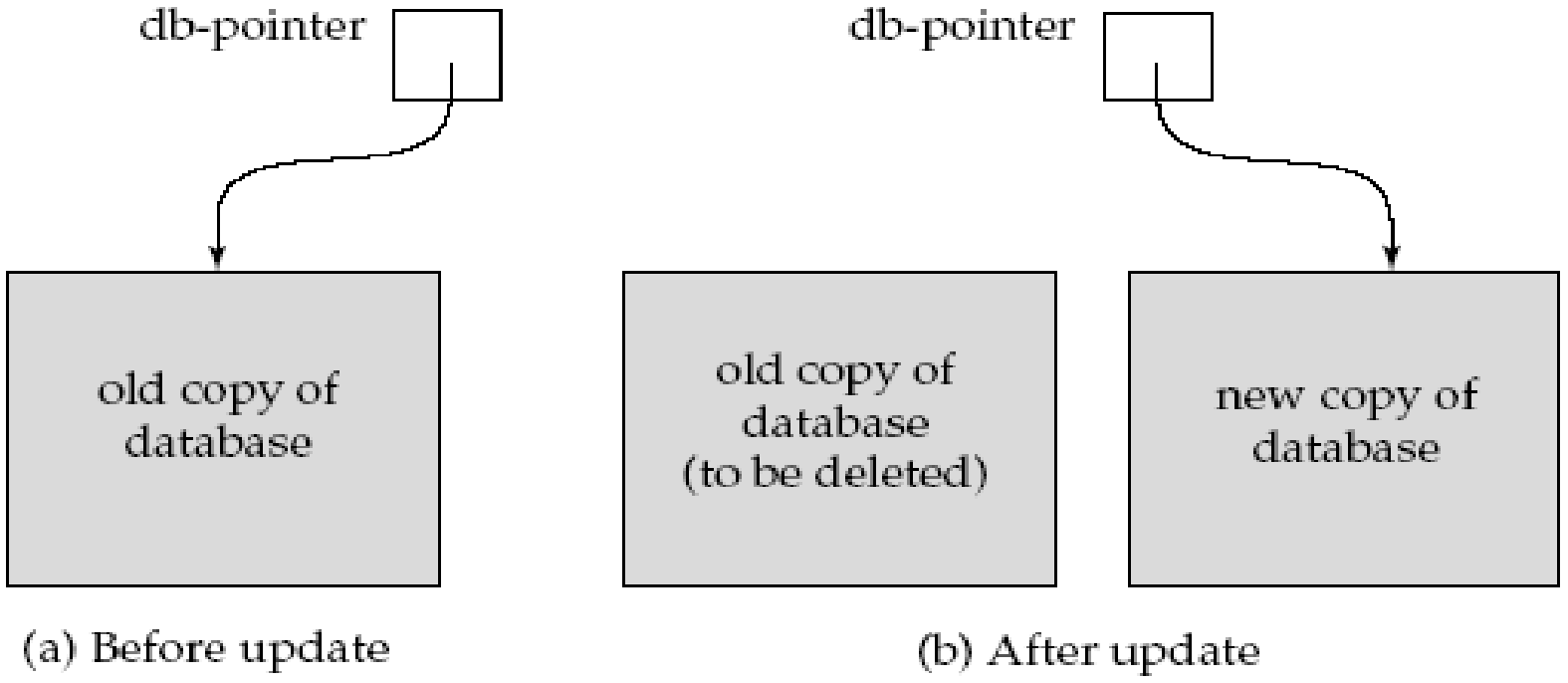


Figure 15.2 Shadow-copy technique for atomicity and durability.

- In this scheme, a transaction that wants to update the database first creates a complete copy of the database

- All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched

- If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected

- If the transaction completes, it is committed as follows:

- First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk

- After the operating system has written all the pages to disk, the database system updates the db-pointer to point to the new copy of the database

- The new copy then becomes the current copy of the database

- The old copy of the database is then deleted

- Figure 15.2 depicts the scheme

- The transaction is said to have been committed at the point where the updated db-pointer is written to disk

- Now consider how the technique handles transaction and system failures:

Transaction failure: If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected

- We can abort the transaction by just deleting the new copy of the database

- Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer

System failure: Suppose that the system fails at any time before the updated db-pointer is written to disk

- When the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database

- Next, suppose that the system fails after db-pointer has been updated on disk

- Before the pointer is updated, all updated pages of the new copy of the database were written to disk

- We assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure

Concurrent Executions

- Transaction-processing systems usually allow multiple transactions to run concurrently

- Allowing multiple transactions to update data concurrently causes several complications with consistency of the data

- There are two good reasons for allowing concurrency:

1. Improved throughput and resource utilization:

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity

- The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU

- All of this increases the throughput of the system-that is, the number of transactions executed in a given amount of time

- Correspondingly, the processor and disk utilization also increase

2. Reduced waiting time:

- There may be a mix of transactions running on a system, some short and some long

- If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete

- Which can lead to unpredictable delays in running a transaction

- If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them

- It also reduces the average response time: the average time for a transaction to be completed after it has been submitted

- The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database

Example

- Transaction T1 transfers $50 from account A to account B. It is defined as

```
T1:  read(A);
     A := A − 50;
     write(A);
     read(B);
     B := B + 50;
     write(B)
```

- Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as

```
T2:   read(A);
      temp := A * 0.1;
      A := A − temp;
      write(A);
      read(B); B := B + temp; write(B)
```

- Suppose the current values of accounts A and B are $1000 and $2000, respectively

- Schedule: It is a collection of transactions

- When transactions are run consecutively, that **schedule** is a **serial schedule**

- When the operations of a transaction overlap, that **schedule** is a **non-serial schedule (concurrent schedule)**

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Figure 15.3 Schedule 1—a serial schedule in which T_1 is followed by T_2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Figure 15.4 Schedule 2—a serial schedule in which T_2 is followed by T_1.

▪Suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 15.5.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Figure 15.5 Schedule 3—a concurrent schedule equivalent to schedule 1.

- In Schedules 1, 2 and 3, the sum A + B is preserved
- The following concurrent schedule does not preserve the value of (A + B)

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Figure 15.6 Schedule 4—a concurrent schedule.

Serializability

- **Basic Assumption** – Each transaction preserves database consistency

- Thus serial execution of a set of transactions preserves database consistency

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule

- Different forms of schedule equivalence give rise to the notions of:
    1. **conflict serializability**
    2. **view serializability**

- We ignore operations other than **read** and **write** instructions

- Our simplified schedules consist of only **read** and **write** instructions

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions write Q
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them

- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**

- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

- Schedule 3 can be transformed into Schedule 5, by series of swaps of non-conflicting instructions

- Therefore Schedule 3 is conflict serializable

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

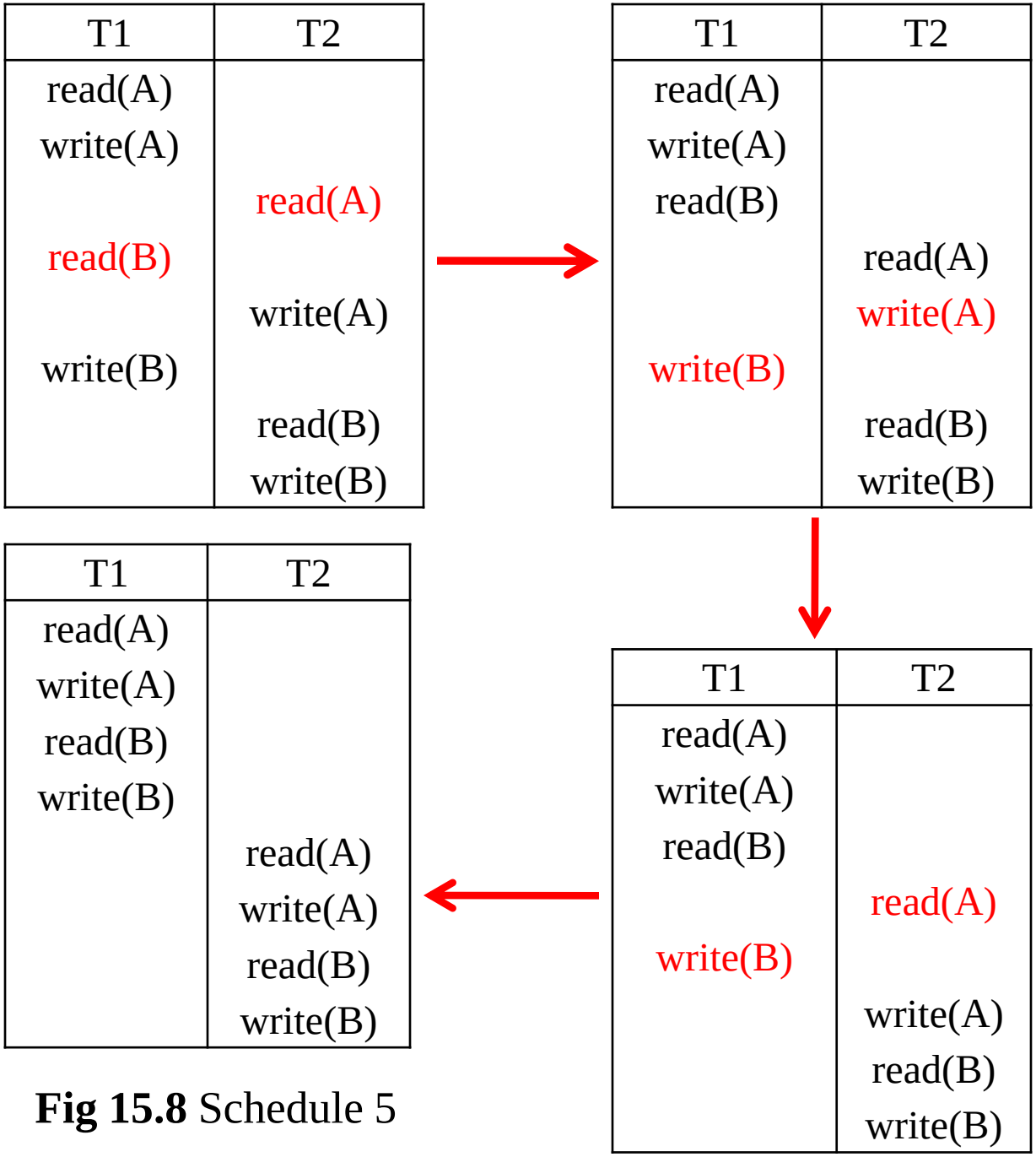


Fig 15.8 Schedule 5

- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >

- **Note:** Serializability is used to check whether concurrent schedule ensures database consistency or not

Which of the following schedules are conflict-serializable schedules?

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)

2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)

3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)

4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met:

1. If T_i reads the initial value of data item Q in S, it must also read the initial value of Q in S'

2. If T_i reads a value of Q written by T_j in S, it must also read the value of Q written by T_j in S'

3. For each data item Q, the transaction (if any) that performs the final write on Q in S must also perform the final write on Q in S'

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule

- Every conflict serializable schedule is also view serializable

- In our previous examples, schedule 1 (Fig 15.3) is not view equivalent to schedule 2 (Fig 15.4) , since, in schedule 1, the value of account A read by transaction T_2 was produced by T_1, whereas this case does not hold in schedule 2

- However, schedule 1 is view equivalent to schedule 3 (Fig 15.5)

Recoverability

- We now address the effect of transaction failures during concurrent execution

- If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction

- In a concurrent execution, any transaction T_j that is dependent on T_i (that is, T_j has read data written by T_i) is also aborted
- There are two types of schedules:

1. Recoverable Schedules
2. Cascadeless Schedules

Recoverable Schedules

- Consider schedule 11 in Figure 15.13

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Figure 15.13 Schedule 11.

- Suppose that the system allows T_9 to commit immediately after executing the read(A) instruction

- Thus, T_9 commits before T_8 does

- Now suppose that T_8 fails before it commits

- Since T_9 has read the value of data item A written by T_8, we must abort T_9 to ensure transaction atomicity

- However, T_9 has already committed and cannot be aborted

- Thus, we have a situation where it is impossible to recover correctly from the failure of T_8

- Schedule 11, with the commit happening immediately after the read(A) instruction, is an example of a non recoverable schedule

- Database system require that all schedules be recoverable

- A **recoverable schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j

Cascading rollback

- A single transaction failure leads to a series of transaction rollbacks is called cascading rollback

- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Cascadeless schedules

- A **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j

- Every cascadeless schedule is also recoverable

- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur

Implementation of Isolation

- A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed

- While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released

- As a result of the **locking policy**, only **one transaction can execute at a time**

- Therefore, only serial schedules are generated

Testing for Serializability

- When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable

- We can determine whether a schedule is serializable or not using precedence graph

Precedence graph

- A direct graph where the vertices are the transactions (names)

- We draw an arc from T_i to T_j if the two transactions conflict

Figure 15.15 Precedence graph for (a) schedule 1 and (b) schedule 2.

- The precedence graph for schedule 4 appears in Figure 15.16
- It contains the edge T1→T2, because T1 executes read(A) before T2 executes write(A). It also contains the edge T2 → T1, because T2 executes read(B) before T1 executes write(B)

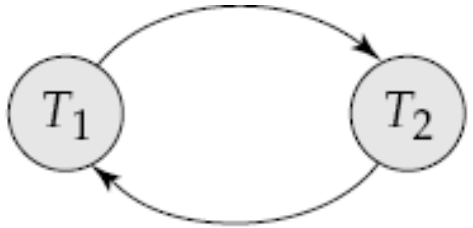


Figure 15.16 Precedence graph for schedule 4

- If the **precedence graph contain cycles**, then schedule S is not conflict serializable
- If the **precedence graph contains no cycles**, then the schedule S is conflict serializable
- Schedules in **Fig 15.15** are conflict serializable
- Schedule in **Fig 15.16** is not conflict serializable

- If precedence graph is acyclic, the **serializability order** of the transactions can be obtained through **topological sorting**

- For example, the graph of Figure 15.17a has the two acceptable linear orderings shown in Figures 15.17b and 15.17c

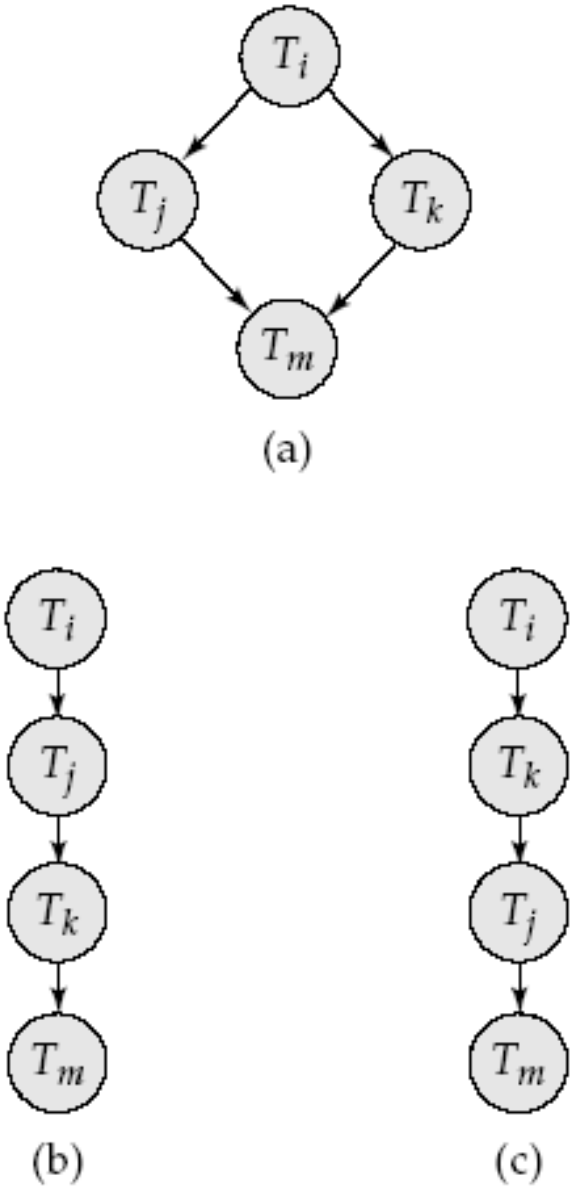
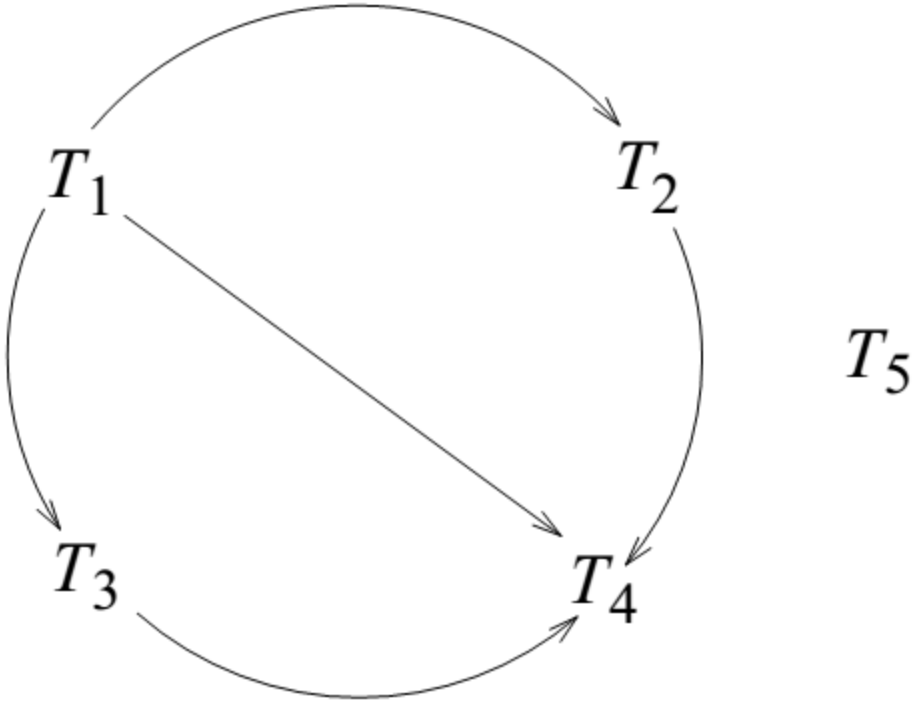


Figure 15.17 Illustration of topological sorting.

Example Schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				write(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule A



Schedule-A is conflict serializable. The serializability order for Schedule-A would be

$$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$$

Concurrency-control schemes

- There are different schemes for controlling the execution of concurrent transactions

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item

- In this method data items are accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item

- Data items can be locked in two modes :

1. exclusive (X): Data item can be both read as well as write

X-lock is requested using **lock-X** instruction

2. shared (S): Data item can only be read. S-lock is requested using **lock-S** instruction

- Lock requests are made to concurrency-control manager

- Transaction can proceed only after request is granted

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- Only shared mode is compatible with shared mode

- At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item

- A transaction may be granted a lock on an data item if the requested lock is compatible with locks already held on the data item by other transactions

- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item

- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted

- Examples: transactions performing locking:

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

Figure 16.2 Transaction T_1.

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Figure 16.3 Transaction T_2

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress - executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*

- Such a situation is called a **deadlock**. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released

The Two-Phase Locking Protocol

- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks

- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed

- Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests

- The protocol assures serializability

- It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock)

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests

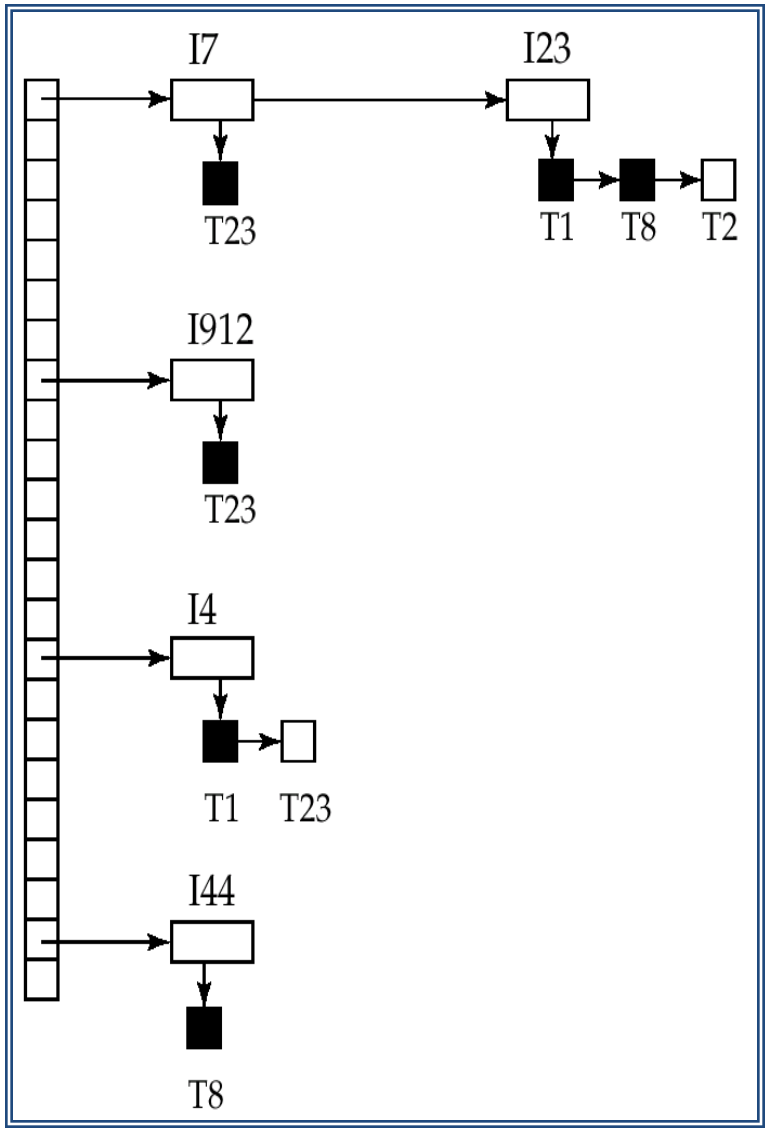
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)

- The requesting transaction waits until its request is answered

- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests

▪The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests

- Lock table also records the type of lock granted or requested

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j)

- Time-stamps determine the serializability order

- Use value of the **system clock** as timestamp. a transaction's timestamp is equal to the value of the clock when the transaction enters the system

- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:

  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully

  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully

- Any conflicting **read** and **write** operations are executed in timestamp order

- Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
   - Hence, the **read** operation is rejected, and T_i is rolled back.

2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

- Suppose that transaction T_i issues **write**(Q)

1. If TS(T_i)<R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back

2. If TS(T_i) <W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i)

Thomas' Write Rule

There is modification to the timestamp-ordering protocol

1. If TS(T_i)<R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back

2. If TS(T_i)<W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i)

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.

1. **Read and execution phase**: Transaction T_i writes only to temporary local variables

2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability

3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back

- Each transaction T_i has 3 timestamps

  Start(T_i) : the time when T_i started its execution

  Validation(T_i): the time when T_i entered its validation phase

  Finish(T_i) : the time when T_i finished its write phase

Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state

- Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all

- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself

- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**

- We assume (initially) that transactions run serially, that is, one after the other
- Stable storage: storage that survives all failures

Log-Based Recovery

- A **log** is kept on stable storage
  - The log is a sequence of **log records**, and maintains a record of update activities on the database
- When transaction T_i starts, it registers itself by writing a **<T_i start>** log record
- Before T_i executes **write**(X), a log record **<T_i, X, V_1, V_2>** is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X
- **< T_i commit>** Transaction T_i has committed
- **< T_i abort>** Transaction T_i has aborted
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- This scheme records all modifications to the log, but defers all the **write**s to after partial commit
- Assume that transactions execute serially
- Transaction starts by writing <T_i ***start***> record to log
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred

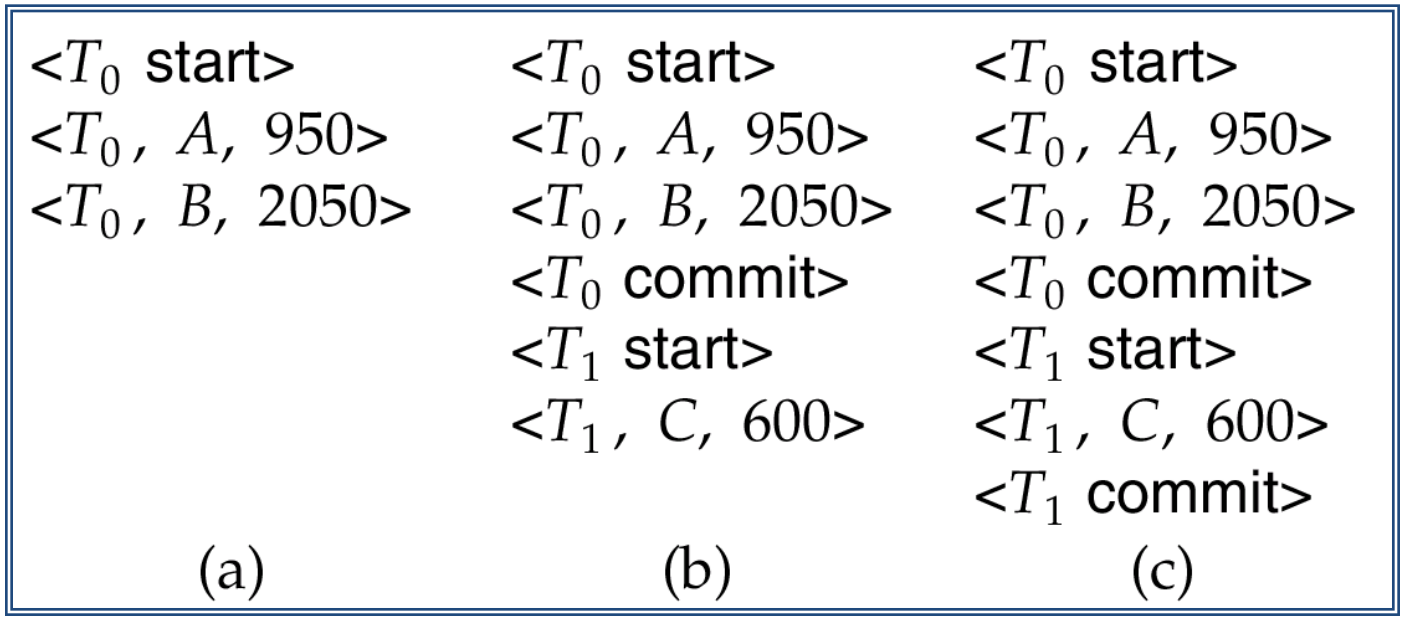
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are used to actually execute the previously deferred writes
- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and <T_i **commit**> are there in the log
- Redoing a transaction T_i (redo T_i) sets the value of all data items updated by the transaction to the new values
- Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken

- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A);
$A := A - 50$;
Write (A);
read (B);
$B := B + 50$;
write (B);

T_1 : **read** (C);
$C := C - 100$;
write (C);

- Below we show the log as it appears at three instances of time



- If log on stable storage at time of crash is as in case:

  (a) No redo actions need to be taken

  (b) redo(T_0) must be performed since <T_0 **commi**t> is present

  (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_1 **commit**> are present

Immediate Database Modification

- This scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written

- Output of updated blocks can take place at any time before or after transaction commit

- Order in which blocks are output can be different from the order in which they are written

Immediate Database Modification Example

Log	Database
$<T_0$ start>	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit>	
$<T_1$ start>	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit>	

State of system log and database corresponding to T_0 and T_1

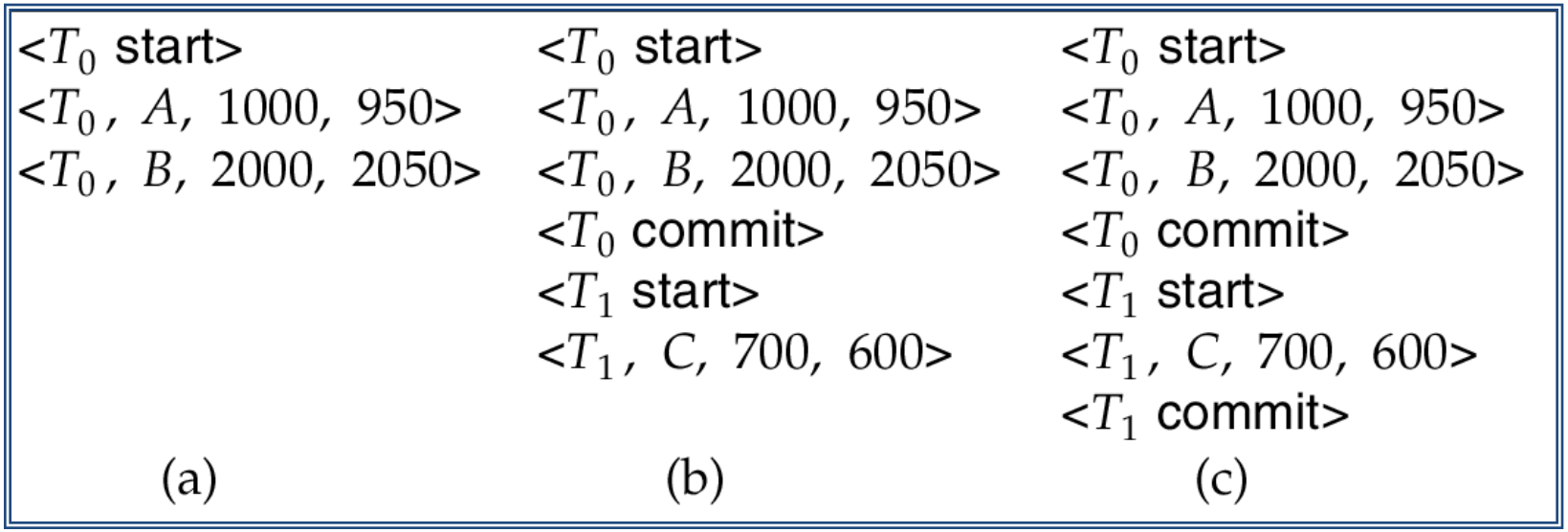
- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery

- When recovering after failure:
  Transaction T_i needs to be undone if the log contains the record $<T_i$ start$>$, but does not contain the record $<T_i$ commit$>$
  Transaction T_i needs to be redone if the log contains both the record $<T_i$ start$>$ and the record $<T_i$ commit$>$

- Undo operations are performed first, then redo operations

Immediate DB Modification Recovery Example

- Below we show the log as it appears at three instances of time



- Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are

set to 950 and 2050 respectively

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050

respectively. Then *C* is set to 600

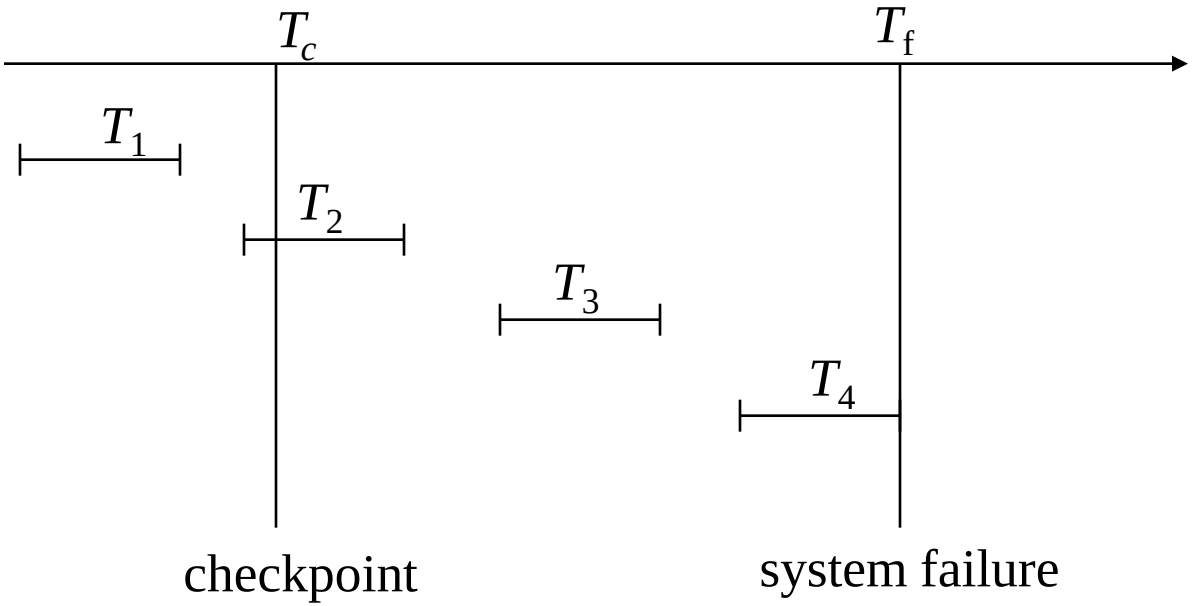
Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage
  2. Write outputs to the disk
  3. Write a log record <**checkpoint**> onto stable storage

- During recovery we need to consider only the most recent transaction $\mathbf{T_i}$ that started before the checkpoint, and transactions that started after $\mathbf{T_i}$
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <$\boldsymbol{T_i}$ **start**> is found
  3. Consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired
  4. For all transactions (starting from T_i or later) with no <$\boldsymbol{T_i}$ **commit**>, execute **undo(*T_i*)**. (Done only in case of immediate modification)

5. Scanning forward in the log, for all transactions starting from T_i or later with a **<T_i commit>**, execute **redo(T_i)**

Example of Checkpoints



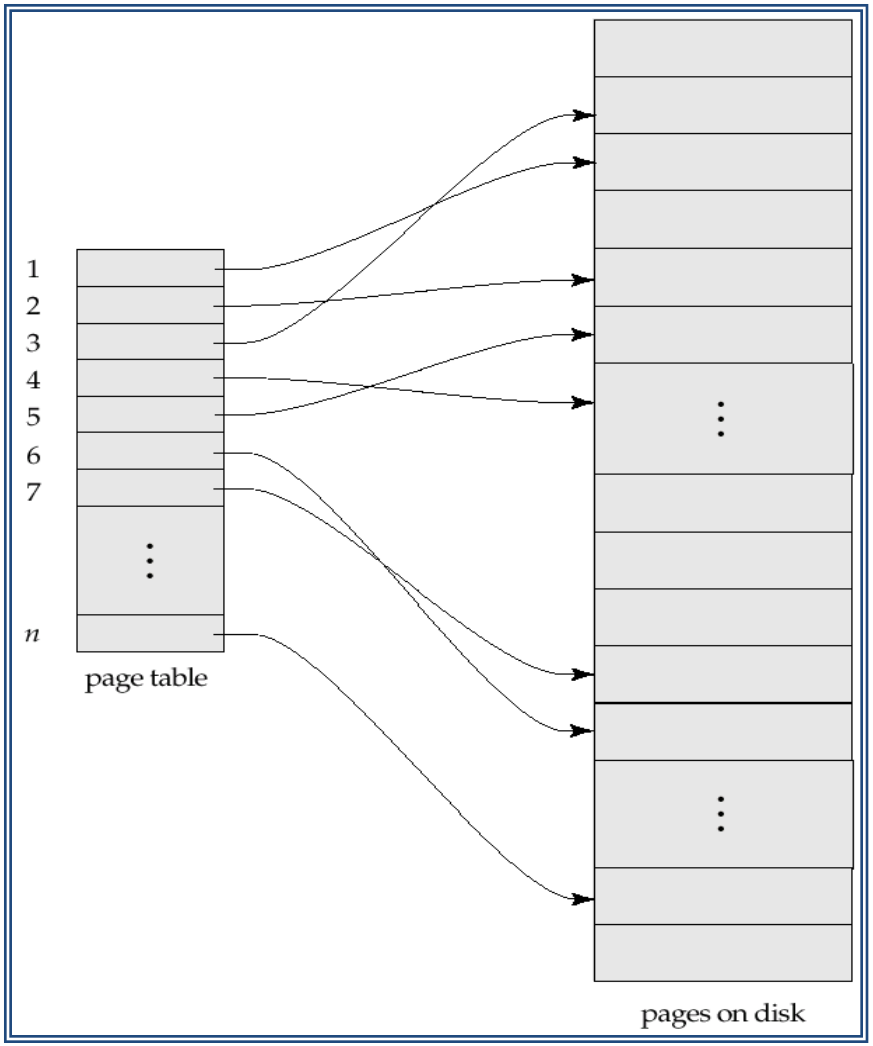
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

Shadow Paging

- database is partitioned into some number of fixed-length blocks, which are referred to as pages
- The page table has n entries—one for each database page. Each entry contains a pointer to a page on disk
- Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction –the current page table, and the shadow page table
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction
- When a write operation is performed, a new copy of the modified database page is created

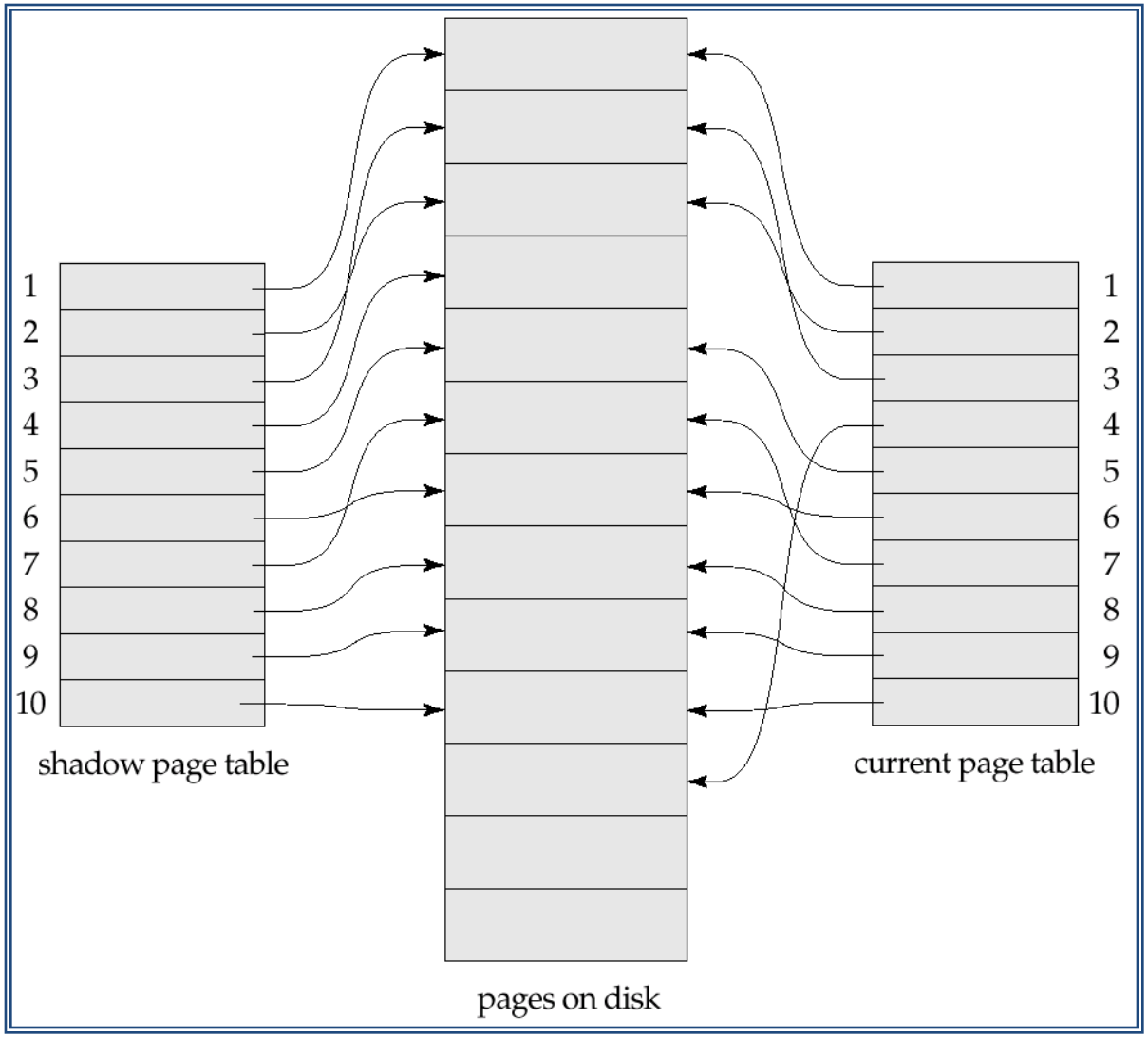
- The current page table is then made to point to the copy
- The update is performed on the copy

Sample Page Table



Example of Shadow Paging

Shadow and current page tables after write to page 4



- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table
- Pages not pointed to from current/shadow page table should be freed (garbage collected)

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions

- Logging is done as described earlier
  - Log records of different transactions may be interspersed in the log

- The checkpointing technique and actions taken on recovery have to be changed
  - since several transactions may be active when a checkpoint is performed

- Checkpoints are performed as before, except that the checkpoint log record is now of the form <checkpoint *L*> where *L* is the list of transactions active at the time of the checkpoint
  - We assume no updates are in progress while the checkpoint is carried out (will relax this later)

- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty

2. Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found. For each record found during the backward scan:
   - if the record is <T_i **commit**>, add T_i to *redo-list*
   - if the record is <T_i **start**>, then if T_i is not in *redo-list*, add T_i to *undo-list*
3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

▪At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone

- Recovery now continues as follows:

1. Scan log backwards from most recent record, stopping when <T_i **start**> records have been encountered for every T_i in *undo-list*

   During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*

2. Locate the most recent <**checkpoint** *L*> record

3. Scan log forwards from the <**checkpoint** *L*> record till the end of the log

   During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**> /* Scan at step 1 comes up to here */
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 **commit**>

Buffer Management

Log Record Buffering

- Log records are buffered in main memory, instead of being output directly to stable storage
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows

- Before writing a data item, transaction acquires exclusive lock on block containing the data item
- Lock can be released once the write is completed
  - Such locks held for short duration are called latches
- Before a block is output to disk, the system acquires an exclusive latch on the block
  - Ensures no update can be in progress on the block

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory

Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage
    - Output all buffer blocks onto the disk
    - Copy the contents of the database to stable storage
    - Output a record <**dump**> to log on stable storage

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed