Recovery System

Instructor : Nitesh Kumar Jha

niteshjha@soa.ac.in

ITER,S'O'A(DEEMED TO BE UNIVERSITY)

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition, e.g. bad input, data not found etc.
  - **System errors**: system has entered an undesirable state(e.g., deadlock) for which transaction can't continue.
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted as result of a system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers $50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, which maintains information about update activities on the database.
- When transaction T_i starts, it registers itself by writing a record <T_i **start**> to the log
- *Before* T_i executes a **write**(q), a log record of the form <T_i, Q, V_{old}, V_{new}> is written
  - where V_{old} is the value of Q before the write, and V_{new} is the value to be written to Q.
- When T_i finishes it last statement, the log record of the form <T_i **commit**> is written.
- When there is an abnormal termination, the log record of the form <T_i **abort**> is written.

Atomicity Preservation

- In the event of a failure, the system scans the log from bottom to top in order to determine the transaction whose atomicity/durability properties are at risk.
- the recovery scheme performs following operations.
  - If for a transaction <T_i start> log record is found but <T_i commit> record not found then this transaction need to be rolled back
  - To preserve atomicity, undo(T_i) is executed.
  - Undo(T_i): restores all modified data items to their old values as depicted in the corresponding modification log records of transaction T_i

Durability Preservation

- All the transactions who have completed execution and subsequently committed by the time failure occurs have their durability property at risk.
- Procedure
  - The logs are scanned backward to find the transactions having both <Ti start> and <Ti commit> records are present in the log
  - To preserve durability execute Redo(T_i)
  - It sets the value of each modified data item of transaction T_i to its new value as found in all modified log records of transaction T_i.
  - Both Undo(T_i), and Redo(T_i) operation are idempotent, i.e. undoing or redoing a transaction several times ensures the same final outcome.

Preservation example

- T_1:
  - <T_1 start>
  - Read (A)
  - <T_1, A, 1000,950 >
  - A=A-50
  - Write(A)
  - Read(B)
  - <T_1, B, 500,550 >
  - B=B+50 ← F_1
  - Write(B)
  - <T_1commit>
- T_2:
  - <T_2 start>
  - Read (C)
  - <T_2, C, 300,400 >
  - C=C+100 ← F_2
  - Write(C)

<T_2 commit>

<T_3 start>

- T3:
  - Read (D)
  - Read(E)
  - Display(D+E) ← F_3

<T_3 commit>

- F_1:
  - Undo(T_1)
- F_2:
  - Undo(T_2)
  - Redo(T_1)
- F_3:
  - Undo (T_3)
  - Redo(T_2)
  - Redo(T_1)

Recovery Procedure

Approaches to log based recovery

- Immediate database modification: allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
  - In this scheme, transaction needs to undergo Undo(Ti) operation in case of failure to preserve atomicity.
- Deferred database modification: performs updates to buffer/disk only at the time of transaction commit
  - In this scheme, transaction does not need to perform Undo(Ti) operation in the event of failure.
  - The recovery procedure in this case needs to ignore and delete corresponding modification log record of the failed transaction.

Database Modification Example

Not IMP

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_o, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B , B_C
<T_1 **commit**>		
		B_A

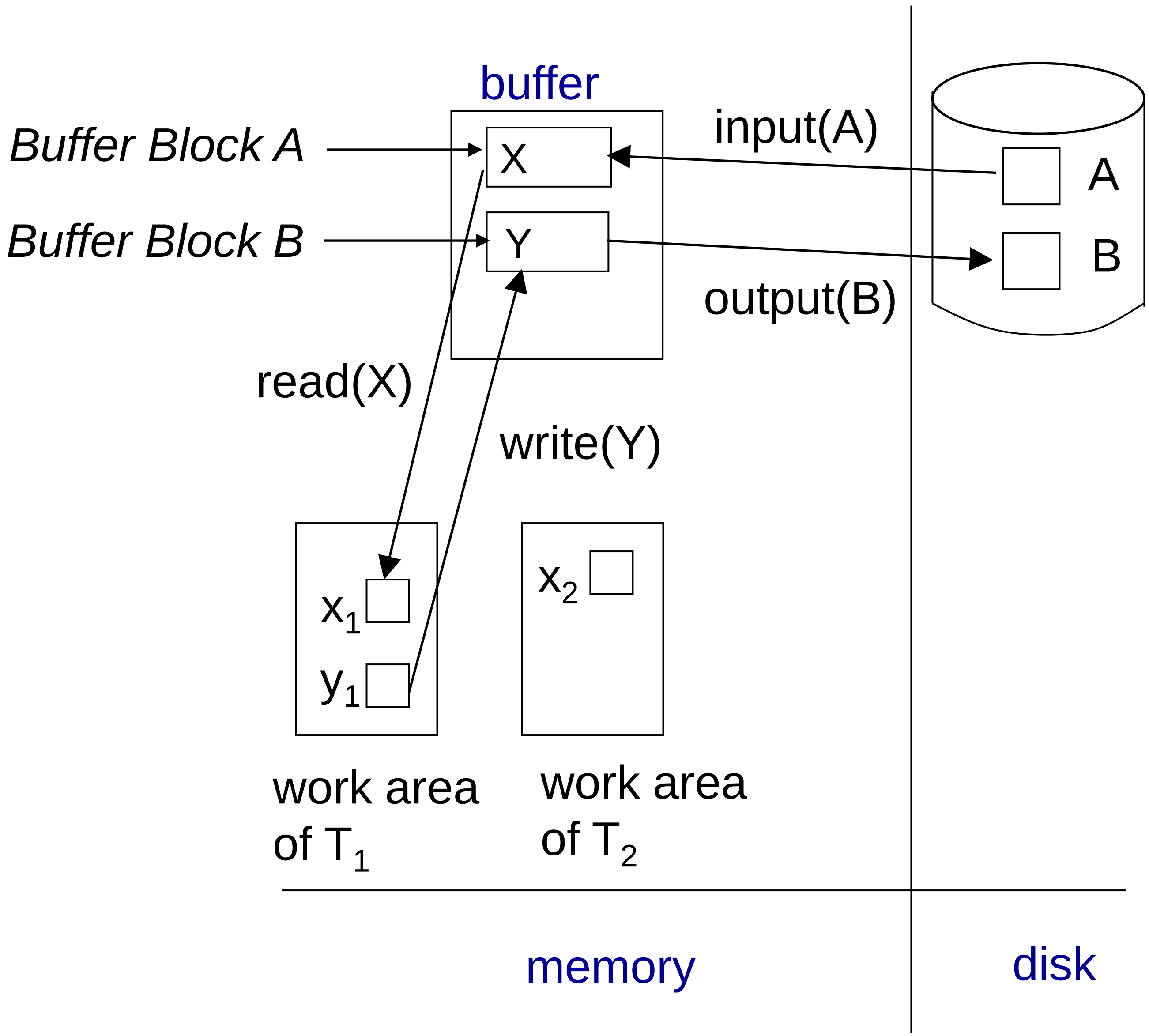
B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Data Access with Concurrent transactions

Not IMP



Checkpoints

Not IMP

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - unnecessary redo of transactions which have already output their updates to the database.
- To get rid of above overheads, checkpoints are introduced and **checkpointing** is performed periodically.
- All updates are stopped while doing checkpointing
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.

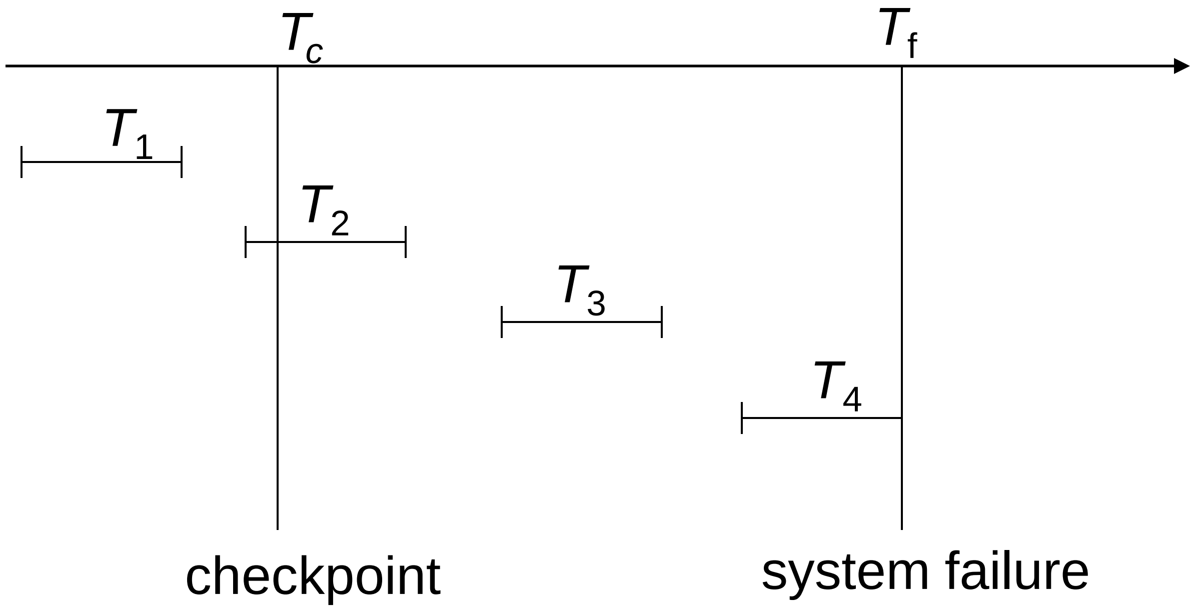
Checkpoints (Cont.)

Not IMP

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints

Not IMP



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

Checkpoints (contd)

Not IMP

- With the use of checkpoints, the recovery procedure become efficient and straight line
  - When failure occurs, the log only needs to be scanned up to the latest checkpoint.
  - During this scan, those transactions whose commit record are found (<T_i commit>) are determined to be redone.
  - Transactions without any commit record found during this scan are undone.

End of Chapter 16