LECTURE 3

SHADOW PAGING (1/7)

- Shadow paging considers the database to be made up of a number of **fixed-size disk pages** (say n) for recovery purposes
- A **directory** with n entries is constructed, where the i^{th} entry points to the i^{th} database page on disk
- **Directory is kept in main memory if it is not too large**, and all references – reads or writes – to database pages on disk go through it

SHADOW PAGING (2/7)

- When transaction begins executing, the **current directory** – whose entries point to the most recent or current database pages on disk – is copied into a **shadow directory**
- **Shadow directory is then saved on disk while the current directory is used by the transaction**
- **During transaction execution, shadow directory is never modified**

SHADOW PAGING (3/7)

- When a *write_item* operation is performed, a new copy of the modified database page is created, but old copy is not overwritten
- New page is written elsewhere – on some previously unused disk block
- Current directory is modified to point to the new disk block, whereas shadow directory is not modified and continues to point to the old unmodified disk block

SHADOW PAGING (4/7)

- For pages updated by transaction, **two versions** are kept
  - **Old version** is **referenced** by the **shadow directory**
  - **New version** is **referenced** by the **current directory**

- To recover from failure during transaction execution, it is sufficient to free the modified database and to discard the current directory

- The state of the database before execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory

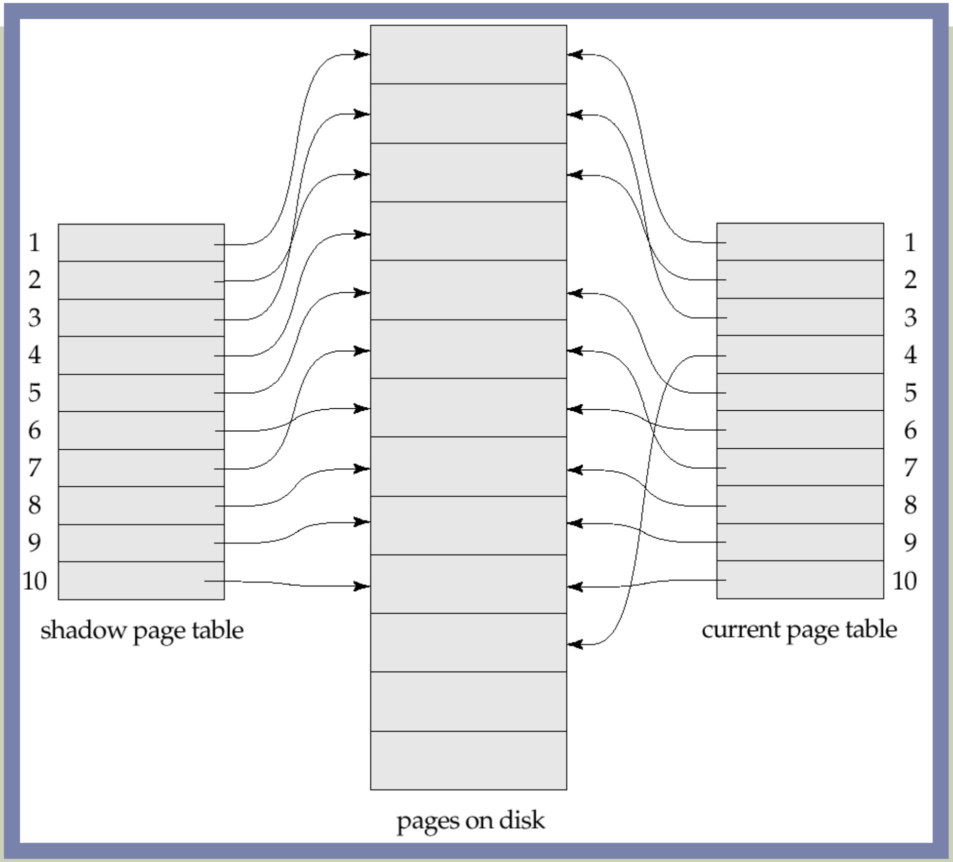
SHADOW PAGING (5/7)

- **Committing a transaction corresponds to discarding the previous shadow directory**
- This technique can be categorized into **NO_UNDO/NO-REDO technique** for recovery as neither undoing nor redoing of data items is involved
- In multi-user environment with concurrent transactions, logs and checkpoints must be incorporated into shadow paging technique

SHADOW PAGING (6/7)

- Disadvantages
  - Updated database pages change location on disk. Hence, difficult to keep related database pages close together on disk without complex storage management strategies
  - Overhead of writing shadow directories to disk as transactions commit is significant
  - Complication on handling garbage collection when a transaction commits. Old pages referenced by shadow directory that have been updated must be released and added to a list of free pages for future use

SHADOW PAGING (7/7)



Example: **Shadow and current page tables after write to page 4**

RECOVERY WITH CONCURRENT TRANSACTIONS (1/13)

- The system has a single disk buffer and a single log
- All transactions share the buffer blocks
- Allow immediate modification and permit a buffer block to have data items updated by one or more transactions

RECOVERY WITH CONCURRENT TRANSACTIONS (2/13)

- **Interaction with concurrency control**
  - Recovery scheme depends on concurrency-control scheme that is used
  - To roll back a failed transaction, must undo the updates performed by the transaction
  - Suppose, if T_0 has to be rolled back, and a data item X that was updated by T_0 has to be restored to its old value
  - Restore the value by using undo information in a log record

RECOVERY WITH CONCURRENT TRANSACTIONS (3/13)

- Suppose T_1 has performed another update on X before T_0 is rolled back
- The update performed by T_1 will be lost if T_0 is rolled back
- Hence, if T has updated X, no other transaction may update the same data item until T has committed or been rolled back
- Can ensure this by using strict 2PL, i.e. 2PL with exclusive locks held until the end of the transaction

RECOVERY WITH CONCURRENT TRANSACTIONS (4/13)

- **Transaction rollback**
  - Rollback a failed transaction T_i by using the log
  - System scans the log backward
  - For every log record of the form <T_i, X_j, V_1, V_2> found in the log, system restores the data item Xj to its old value V_1
  - Scanning of log terminates when the log record <T_i start> is found

RECOVERY WITH CONCURRENT TRANSACTIONS (5/13)

- **Scanning a log backward is important, since a transaction may have updated a data item more than once**
- Example

  <T_1, A, 10, 20>
  <T_1, A, 20, 30>

- Scanning log backwards sets A correctly to 10
- If log is scanned forward, then A would be set to 20, which is incorrect

RECOVERY WITH CONCURRENT TRANSACTIONS (6/13)

- If strict 2PL is used for concurrency control, locks held by a transaction T may be released only after T has been rolled back
- Once T has updated a data item, no other transaction could have updated the same data item, because of concurrency control

RECOVERY WITH CONCURRENT TRANSACTIONS (7/13)

- **Checkpoints**
  - The following transactions must be considered during recovery
    - Transactions that started after the most recent checkpoint
    - The one transaction, if any, that was active at the time of the most recent checkpoint
  - In a concurrent transaction processing system, we require that the checkpoint log record be of the form **<checkpoint L>**, where **L is a list of transactions active at the time of checkpoint**

RECOVERY WITH CONCURRENT TRANSACTIONS (8/13)

- Assume that transactions do not perform updates either on buffer blocks or on log while checkpoint is in progress
- **A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out**
- When system recovers from crash, it constructs two lists
  - **Undo-list** consists of transactions to be undone
  - **Redo-list** consists of transactions to be redone

RECOVERY WITH CONCURRENT TRANSACTIONS (9/13)

- Initially, both lists are empty
- System scans log backward, examining each record, until it finds the first <checkpoint> record
  - For each record found of the form <T_i commit>, it adds T_i to redo-list
  - For each record found of the form <T_i start>, if T_i is not in redo-list, then it adds T_i to undo-list
- When system has examined all the appropriate log records, it checks the list L in checkpoint record

RECOVERY WITH CONCURRENT TRANSACTIONS (10/13)

- For each T_i in L, if T_i is not in redo-list then it adds T_i to undo-list
- Once the two lists have been constructed, recovery proceeds as follows
    1. System **rescans log from the most recent record backward**, and **performs an undo** for each log record that belongs to T_i on **undo-list**. Scan **stops when the <T_i start>** records have been **found for every T_i in undo-list. Ignores records of redo-list**

    2. System **locates the most recent <checkpoint L> record on the log**. This step **may involve scanning log forward**, if checkpoint record was passed in step 1

RECOVERY WITH CONCURRENT TRANSACTIONS (11/13)

3. System **scans the log forward from the most recent <checkpoint L> record**, and **performs redo** for each log record that belongs to T_i that is on **the redo-list**. It **ignores records of undo-list**

- Step 1 is important to process log backward to ensure that the resulting state of database is correct
- After system has undone all transactions on undo-list, it redoes those on the redo-list, hence the log is processed forward

RECOVERY WITH CONCURRENT TRANSACTIONS (12/13)

- When recovery process has completed, transaction processing resumes
- Suppose, data item A has initial value 10, T_i update value of A to 20 and aborted
- Transaction rollback would restore the value of A to 10
- Now, suppose another transaction T_j then updated value of A to 30 and committed, here the system crash

RECOVERY WITH CONCURRENT TRANSACTIONS (13/13)

- State of the log at the time of crash is

  <T_i, A, 10, 20>

  <T_j, A, 20, 30>

  <T_j commit>

- Say, first redo → A set to 30

  second undo → A set to 10 *which is wrong*

- **Final value of A should be 30 which we can ensure by performing undo before performing redo**