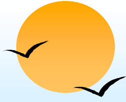
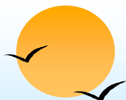


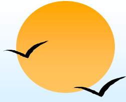
Chapter 17: Recovery System

- Failure Classification
- Log-Based Recovery and Checkpoints
- Redo Protocol!



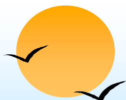
Failure Classification

- Transaction failure :
  - Logical errors: transaction cannot complete due to some internal error condition
  - System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash. It is assumed that non-volatile storage contents are not corrupted.
- Disk failure: a head crash or similar failure destroys all or part of disk storage



Storage Structure

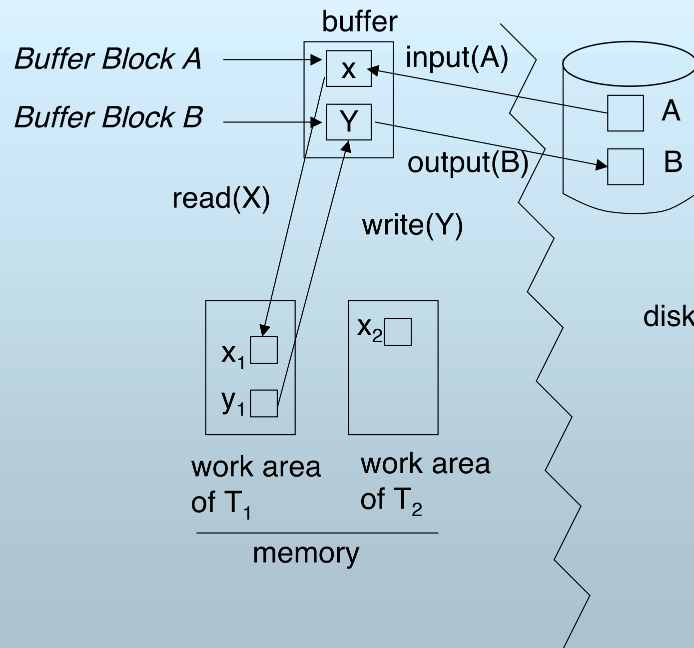
- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape
- Stable storage:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

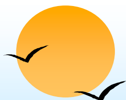


Data Access

- *Physical blocks* are those blocks residing on the disk. *Buffer blocks* are the blocks residing temporarily in main memory.
- Two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- Each transaction has its private work-area in which local copies of all data items accessed and updated by it are kept.

Example of Data Access





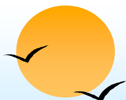
Recovery and Atomicity

- 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. A failure may occur after one of these modifications have been made but before all of them are made.
- There are two main approaches:
  - **log-based recovery** (discussed next)
  - **shadow-paging** (not covered)
- We assume (initially) that transactions run serially, that is, one after the other.

Deferred Database Modification

- This scheme ensures atomicity despite failures by recording all modifications to log, but deferring 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.
- 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, log records are used to actually execute the previously deferred writes.

©Silberschatz, Korth and Sudarshan



An Impossible log

<T_0 **start** >
<T_0, A, 950 >
<T_0, **commit**>
<T_0, B, 2050 >
<T_1 **start** >
<T_1, C, 600>

What is wrong with it ?

©Silberschatz, Korth and Sudarshan

Deferred Database Modification (Cont.)

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

< T_0 **start** >
< T_0, A, 950 >
< T_0, B, 2050 >

(a)

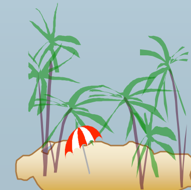
< T_0 **start** >
< T_0, A, 950 >
< T_0, B, 2050 >
< T_0 **commit**>
< T_1 **start** >
< T_1, C, 600>

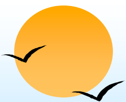
(b)

< T_0 **start** >
< T_0, A, 950 >
< T_0, B, 2050 >
< T_0 **commit**>
< T_1 start >
< T_1, C, 600 >
T_1 **commit** >

(c)

- NB: < T_0, A, 950 > means T_0 writes 950 into location A (after writing commit!)
- 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 **commit**> is present
  (c) **redo**(T_0) must be performed followed by redo(T_1) since
  < T_0 **commit**> and < T_i commit> are present



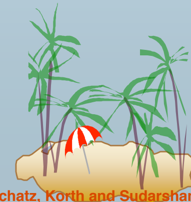


Checkpoints

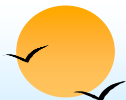
- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already
  3. output their updates to the database.
- Streamline recovery procedure by periodically performing *checkpointing*
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk—finish all the writes for transactions for which the commit has been written!
  3 Write a log record < **checkpoint**> onto stable storage.

Checkpoints (Cont.)

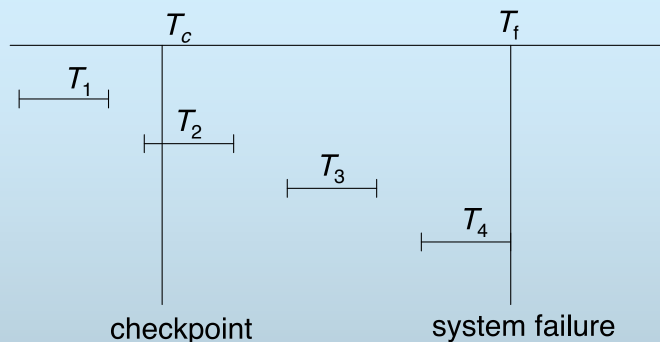
- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
- Scan backwards from end of log to find the most recent <**checkpoint**> record
- Continue scanning backwards till a record <T_i **start**> is found.
- Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
- Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo**(T_i).
  - *What could be a problem with redoing?*
  - *Have you heard of idempotence?*



©Silberschatz, Korth and Sudarshan

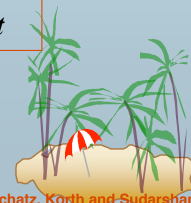


Example of Checkpoints

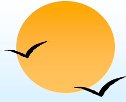


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Start* |----------| *Commit



©Silberschatz, Korth and Sudarshan



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**>
  <T_1, B, 0, 10>
  <T_2 **start**> /*Back scan stops here */
  <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**>

Failure with Loss of Nonvolatile 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
- To recover from disk failure, restore database from most recent dump. Then log is consulted and all transactions that committed since the dump are redone.
- Can be extended to allow transactions to be active during dump; known as *fuzzy* or *online* dump.