Introduction to Databases
《数据库引论》

Lecture 14: System Recovery
第14讲：系统恢复

周水庚 / Shuigeng Zhou

邮件: sgzhou@fudan.edu.cn 网址：admis.fudan.edu.cn/sgzhou

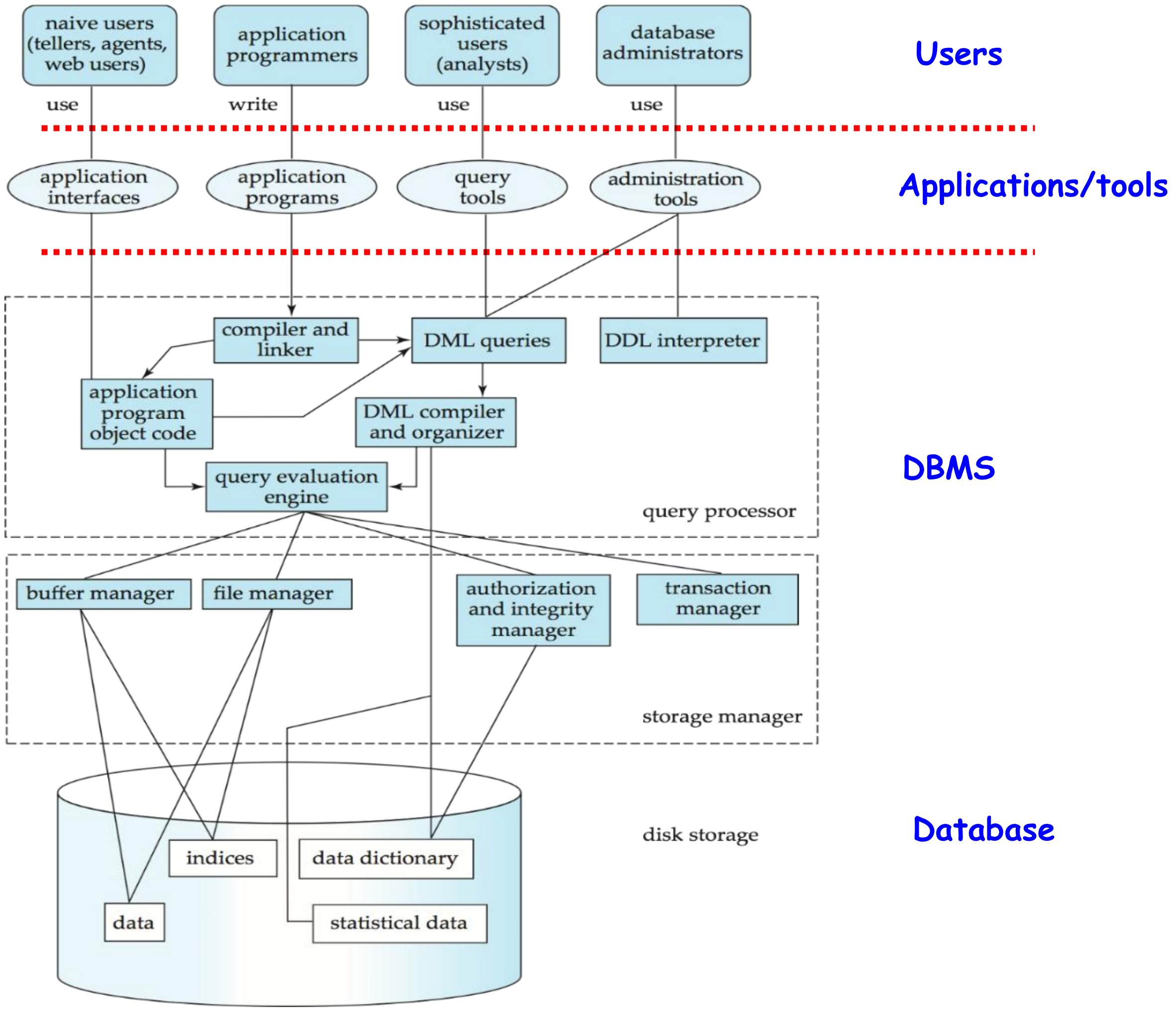
复旦大学计算机科学技术学院

Content of the Course

- **Part 0: Overview**
  - Lect. 0/1 (Feb. 20) - Ch1: Introduction
- **Part 1 Relational Databases**
  - Lect. 2 (Feb. 27) - Ch2: Relational model (data model, relational algebra)
  - Lect. 3 (Mar. 6) - Ch3: SQL (Introduction)
  - Lect. 4 (Mar. 13) – Ch4 & 5: Intermediate & Advanced SQL
- **Part 2 Database Design**
  - Lect. 5 (Mar. 20) - Ch6: Database design based on E-R model
  - Lect. 6 (Mar. 27) - Ch7: Relational database design (Part I)
  - Lect. 7 (Apr. 3) - Ch7: Relational database design (Part II)
- **Midterm exam: Apr. 10**
- **Part 3 Data Storage & Indexing**
  - Lect. 8 (Apr. 17) - Ch12/13: Storage systems & structures
  - Lect. 9 (Apr. 24) - Ch14: Indexing
- **Part 4 Query Processing & Optimization**
  - May 1, holiday, no class
  - Lect. 10 (May 8) - Ch15: Query processing
  - Lect. 11 (May 15) - Ch16: Query optimization
- **Part 5 Transaction Management**
  - Lect. 12 (May 22) - Ch17: Transactions
  - Lect. 13 (May22/29) - Ch18: Concurrency control
  - Lect. 14 (May 29/Jun. 5) - Ch19: Recovery system
  - Lect. 15 (Jun. 5) – Course review

Final exam: 13:00-15:00, Jun. 18

Database System Structure



Outline

- ☞ **Failure Classification**
- Storage
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management

Failure Classification

- **Transaction failure (事务故障)**
  - Logical errors, e.g., illegal inputs
  - System errors, e.g., dead locks
- **System crash (系统崩溃)**
  - A power failure, or other hardware and software failure causes the system to crash
- **Disk failure (磁盘故障)**
  - A head crash or similar disk failure destroys all or part of disk storage

Recovery Algorithms

- Techniques to ensure database **consistency** and transaction **atomicity** despite failures

- **Recovery algorithms have two parts**
  - **Actions taken during normal transaction processing**
    - 保证有足够的信息用于故障恢复
  - **Actions taken after a failure**
    - 恢复数据库到某个一致性状态

Outline

- Failure Classification

☞ **Storage**

- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management

Storage Structure

- **Volatile storage** **(易失性存储器)**
  - does not survive system crashes
  - e.g., main memory, cache memory
- **Nonvolatile storage** **(非易失性存储器)**
  - survives system crashes
  - e.g., disk, tape, flash memory
- **Stable storage** **(稳定存储器)**
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Data Access

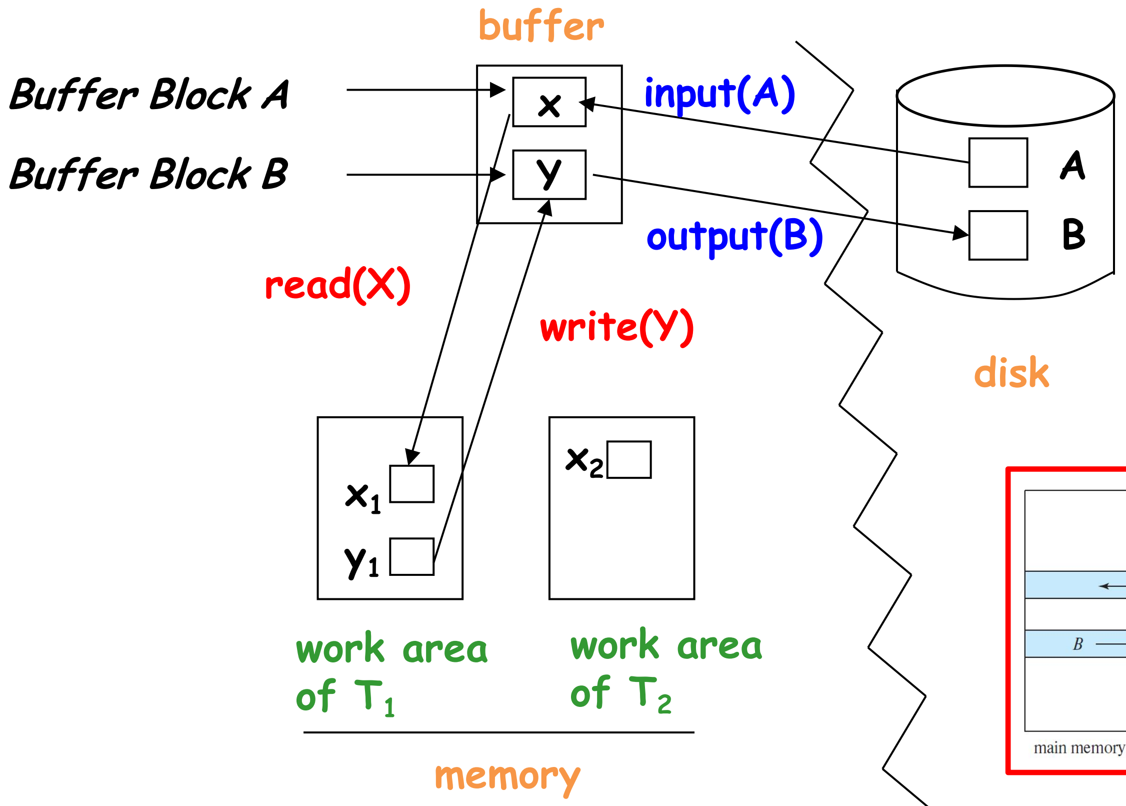
- **Physical blocks** （**物理块**）
  - the blocks residing on the disk
- **Buffer blocks** （**缓冲块**）
  - the blocks residing temporarily in main memory
- Block movements between disk and main memory
  - **input(B): physical block -> memory**
  - **output(B): buffer block -> disk**
- Each transaction T_i has its **private work-area** (私有工作区)
  - T_i's local copy of a data item X is called x_i

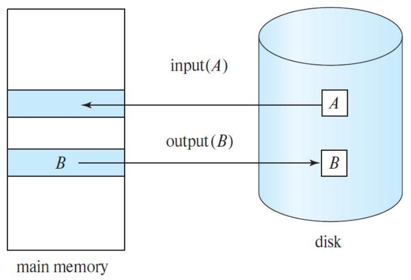
Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using
  - **read(X)**
  - **write(X)**

- Transactions
  - Perform **read(X)** while accessing **X** for the first time
  - All subsequent accesses are to the **local copy**
  - After last access, transaction executes **write(X)**
- **output(B_X)** does not need to immediately follow **write(X)**
  - System can perform the output operation when it deems fit

Example of Data Access





Outline

- Failure Classification
- Storage
- ☞ **Recovery and Atomicity**
- Recovery Algorithms
- Buffer Management

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**
  - 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**
- **Two approaches**
  - log-based recovery (基于日志的恢复)
  - shadow-paging (影子页)

Log-based Recovery

- **A log is kept on stable storage**
  - **The log is a sequence of log records**
- 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 **old value** and V_2 is the **new value**
  - When T_i finishes it's last statement, the log record **<T_i, commit>** is written
- **Two approaches using logs**
  - **Deferred database modification (延迟数据库修改)**
  - **Immediate database modification (即刻数据库修改)**

Deferred Database Modification

- Record all modifications to the log, but **defer all the writes to after partial commit**
  - 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**.
  - 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 read and used to **actually execute the previously deferred writes**

Deferred Database Modification (Cont.)

- **Recovery after a crash**
  - a transaction needs to be **redone iff both <T_i start> and < T_i commit>** are there in the log
  - **Redo(T_i)** sets the value of all data items updated by the transaction **to the new values**
- **Example:**
  - T_0 executes before T_1, and initial: **A=1000, B=2000, C=700**

```
T0: read (A)                T1: read (C)
    A: = A - 50                 C:= C- 100
    write (A)                   write (C)
    read (B)
    B:=  B + 50
    write (B)
```

Deferred Database Modification (Cont.)

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

- Recovery actions in each case above are:
  - (a) No **redo** actions need to be taken
  - (b) **redo(T_0) must be performed**
  - (c) **redo(T_0) must be performed followed by redo(T_1)**

Immediate Database Modification

- **Allows database updates of an uncommitted transaction**
  - **Update log records must be written before database items are written**
  - The 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

$<T_i, X, V_1, V_2>$,

where **V_1** is the **old value,**

and **V_2** is the **new value**

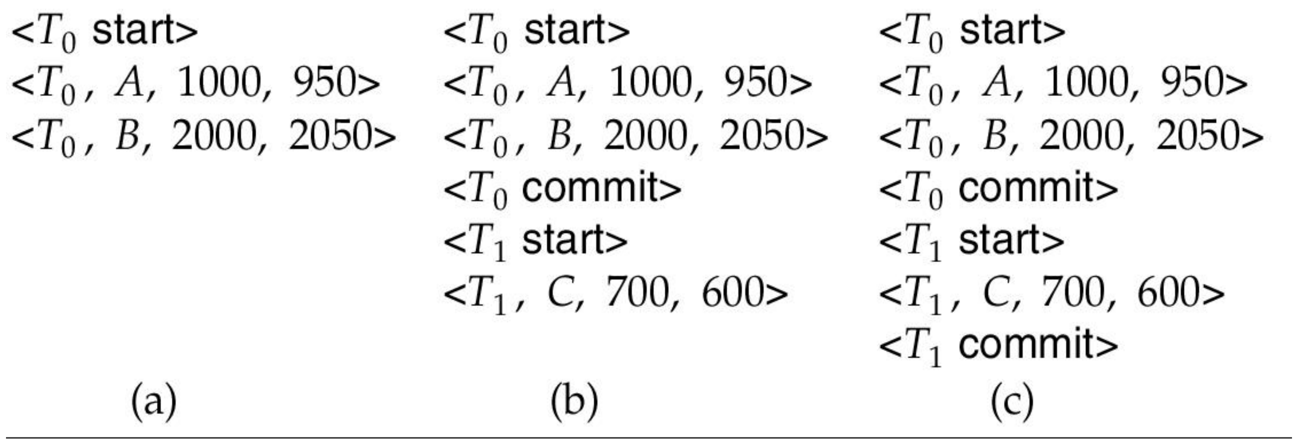
Log	**Write**	**Output**
$<T_0$ start>		
$<T_0$, A, 1000, 950>		
$<T_0$, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
$<T_0$ commit>		
$<T_1$ start>		
$<T_1$, C, 700, 600>		
	$C = 600$	
		B_B, B_C
$<T_1$ commit>		
		B_A

Note: B_X denotes block containing X

Immediate Database Modification (Cont.)

- Recovery procedure has two operations
  - **undo(T_i)**: restore the value of all data items updated by transaction T_i **to the old values**
  - **redo(T_i)**: set the value of all data items updated by transaction T_i **to the new values**
- When recovering after failure
  - Transaction T_i needs to be **undone** if **the log contains the record <T_i start>, but does not contain <T_i commit>**
  - Transaction T_i needs to be **redone** if **the log contains both the record <T_i start> and <T_i commit>**
- **Undo** operations are performed **first**, then redo operations

Immediate DB Modification Recovery Example



<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

- Recovery actions in each case above are:
  - (a) undo(T_0)
  - (b) undo(T_1) and redo(T_0)
  - (c) redo(T_0) and redo(T_1)

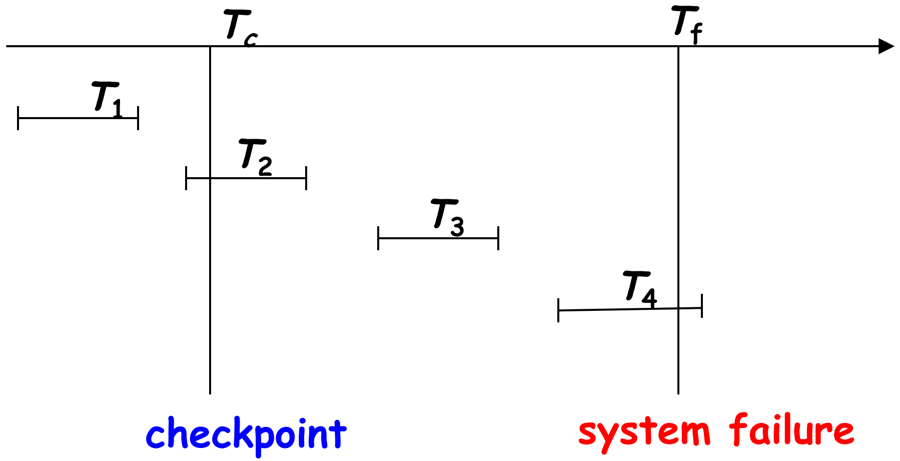
Checkpoints (检查点)

- **Problems in the recovery procedure**
  - searching the entire log is time-consuming
  - might unnecessarily redo transactions which have already output their updates to the database
- **Recovery procedure by setting checkpoints periodically**
  - **Output all log records** currently residing in main memory to stable storage
  - **Output all modified buffer blocks** to the disk
  - Write a log record **<checkpoint>** to stable storage

Checkpoints (Cont.)

- **During recovery**
  - Scan **backwards** from the end of log to find the most recent **<checkpoint>** record
  - Continue scanning **backwards** till a record **<T_i,start>** is found. We assume that all transactions are executed **serially**.
    - Need only consider the part of log following above start record
    - For all transactions (starting from T_i or later) with **no <T_i,commit>**, execute **undo(T_i)**.
    - 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 according to the checkpoint)
- T_2 and T_3 redone
- T_4 undone

Outline

- Failure Classification
- Storage
- Recovery and Atomicity
- ☞ **Recovery Algorithms**
- Buffer Management

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
- We **assume concurrency control using strict two-phase locking**
- Logging is done as described earlier
  - Log records of different transactions may be interspersed（散布）in the log
- The checkpointing technique and actions taken on recovery have to be changed

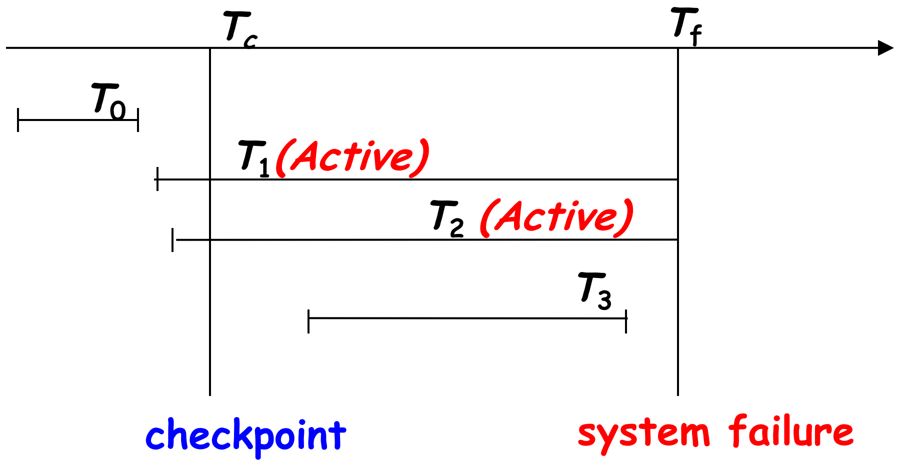
Recovery With Concurrent Transactions (Cont.)

- Checkpoints are performed as before, except that the checkpoint log record is the form ***<checkpoint L>***
  - *L* is a list of transactions **active** at the time of the **checkpoint**
  - We assume no update is in progress while the checkpoint is carried out
- When the system **recovers from a crash**
  - Initialize **undo-list** and **redo-list** to empty
  - Scan the log backwards **until a** ***<checkpoint L>*** record is found:
    - if the record is ***<T_i commit>***, add T_i to **redo-list**
    - if the record is ***<T_i start>*** and T_i is not in **redo-list**, add T_i to **undo-list**
    - for every T_i in *L*, if T_i is not in **redo-list**, add T_i to **undo-list**

Example of Recovery

- Go through 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**>
<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**>



Scan log **backwards**: Undo T1, T2 in undo-list
Scan log **forwards**: Redo T3 in redo-list

Recovery With Concurrent Transactions (Cont.)

- Recovery works as follows
  - Scan log **backwards** from the end of the log
    - During the scan, perform **undo** for each log record that belongs to a transaction in **undo-list**
  - Locate **the most recent <checkpoint L>** record
  - 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**

Outline

- Failure Classification
- Storage
- Recovery and Atomicity
- Recovery Algorithms

☞ **Buffer Management**

Log Record Buffering

- **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
  - Several log records can be output using a single output operation, thus reducing the I/O cost

Log Record Buffering (Cont.)

- **Write-ahead logging (WAL) rule for buffering log records**
  - 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

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, an existing block should be removed from buffer if the buffer is full
  - If the block chosen for removal has been updated, it must be output to disk
- No update should be in progress on a block when it is output to disk, which is 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闩锁**
  - 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

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks
  - When OS needs to evict（逐出）a page that has been modified, the page is written to swap space on disk
  - When DB decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O
    - Known as dual paging（双分页）problem
  - Ideally when swapping out a database buffer page, operating system should pass control to database, which in turn outputs page to database instead of to swap space (making sure to output log records first)
    - Dual paging can thus be avoided, but common operating systems do not support such functionality.

Failure with Loss of Nonvolatile 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
  - 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

End of Lecture 14