

Unit 5

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
 - data loss on power failure or system crash
 - physical failure of the storage device
- Can differentiate storage into:
 - **volatile storage**: loses contents when power is switched off
 - **non-volatile storage**:
 - ▶ Contents persist even when power is switched off.
 - ▶ Includes secondary and tertiary storage, as well as battery-backed up main-memory.

Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- **Main memory:**
 - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
 - generally too small (or too expensive) to store the entire database
 - ▶ capacities of up to a few Gigabytes widely used currently
 - ▶ Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
 - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

Physical Storage Media (Cont.)

■ Flash memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
 - ▶ Can support only a limited number (10K – 1M) of write/erase cycles.
 - ▶ Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones, and USB keys

Physical Storage Media (Cont.)

■ Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
 - ▶ Much slower access than main memory (more on this later)
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Capacities range up to roughly 1.5 TB as of 2009
 - ▶ Much larger capacity and cost/byte than main memory/flash memory
 - ▶ Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
 - ▶ disk failure can destroy data, but is rare

Physical Storage Media (Cont.)

■ Optical storage

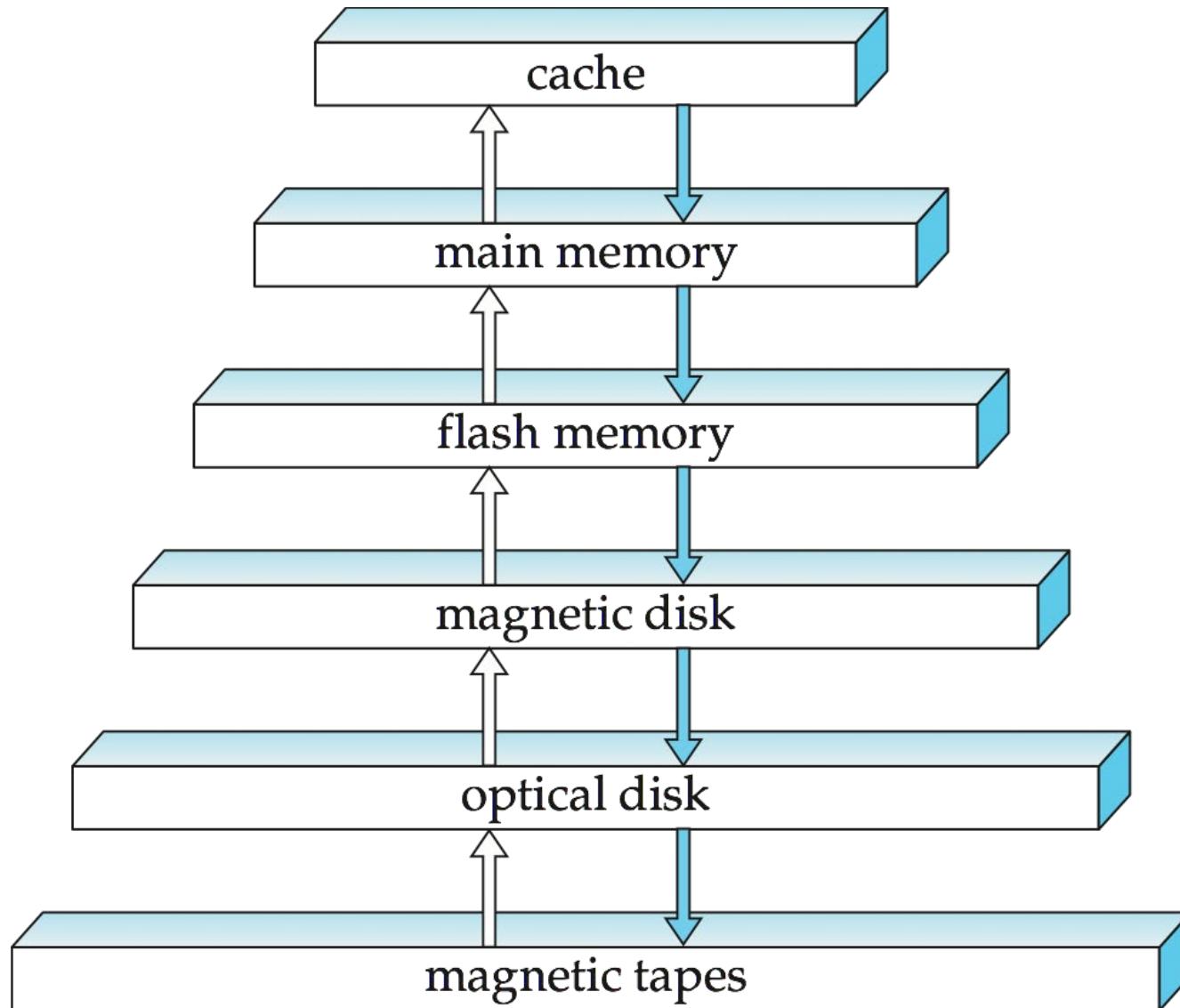
- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Physical Storage Media (Cont.)

■ Tape storage

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive \Rightarrow storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
 - ▶ hundreds of terabytes (1 terabyte = 10^9 bytes) to even multiple **petabytes** (1 petabyte = 10^{12} bytes)

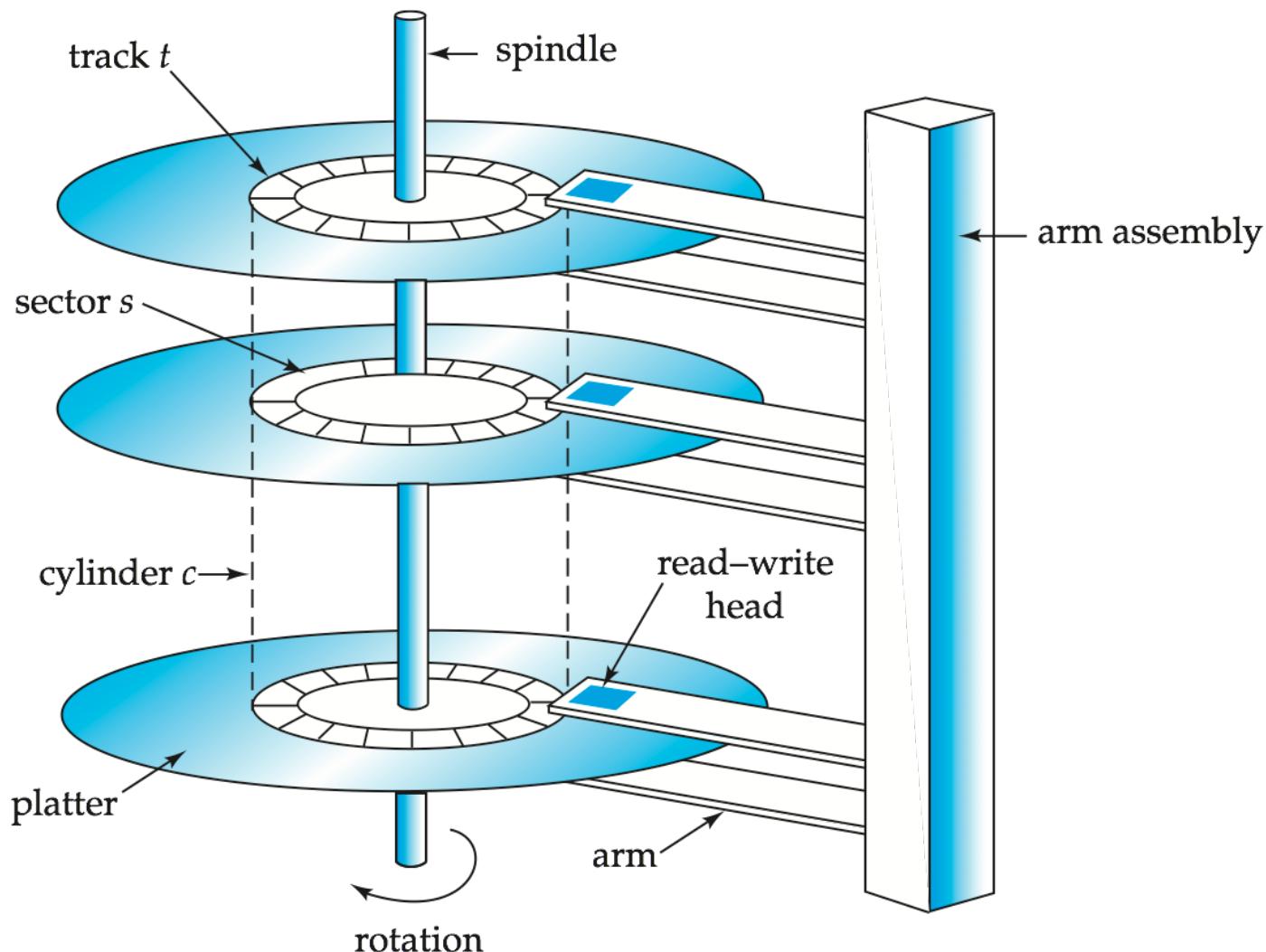
Storage Hierarchy



Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
 - also called **on-line storage**
 - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
 - also called **off-line storage**
 - E.g. magnetic tape, optical storage

Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Magnetic Disks

- **Read-write head**
 - Positioned very close to the platter surface (almost touching it)
 - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
 - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
 - multiple disk platters on a single spindle (1 to 5 usually)
 - one head per platter, mounted on a common arm.
- **Cylinder** i consists of i^{th} track of all the platters

Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
 - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
 - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - accepts high-level commands to read or write a sector
 - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - ▶ If data is corrupted, with very high probability stored checksum won't match recomputed checksum
 - Ensures successful writing by reading back sector after writing it
 - Performs **remapping of bad sectors**

Performance Measures of Disks

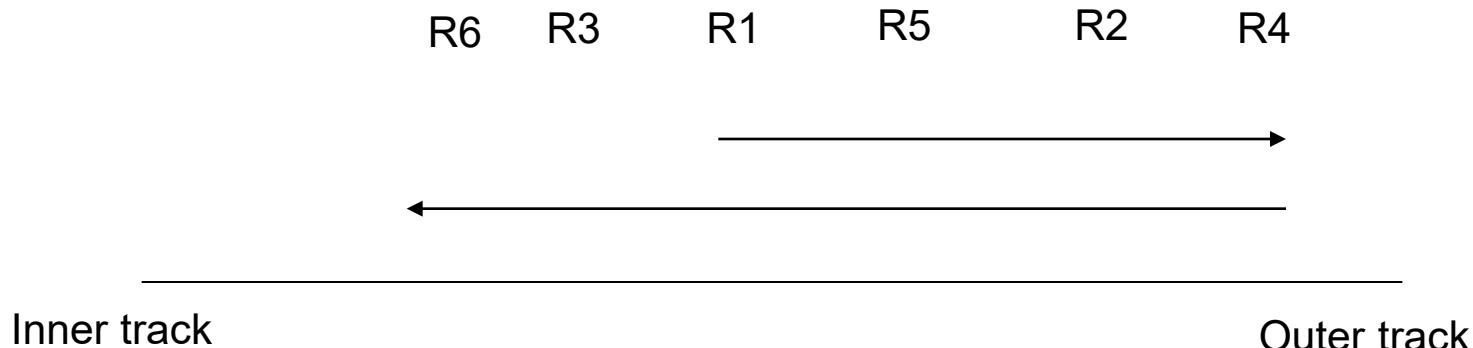
- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
 - **Seek time** – time it takes to reposition the arm over the correct track.
 - ▶ Average seek time is 1/2 the worst case seek time.
 - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
 - ▶ 4 to 10 milliseconds on typical disks
 - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
 - ▶ Average latency is 1/2 of the worst case latency.
 - ▶ 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
 - 25 to 100 MB per second max rate, lower for inner tracks
 - Multiple disks may share a controller, so rate that controller can handle is also important
 - ▶ E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
 - ▶ Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
 - ▶ Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s

Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
 - Typically 3 to 5 years
 - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
 - ▶ E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
 - MTTF decreases as disk ages

Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
 - data is transferred between disk and main memory in blocks
 - sizes range from 512 bytes to several kilobytes
 - ▶ Smaller blocks: more transfers from disk
 - ▶ Larger blocks: more space wasted due to partially filled blocks
 - ▶ Typical block sizes today range from 4 to 16 kilobytes
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
 - **elevator algorithm:**



Optimization of Disk Block Access (Cont.)

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
 - E.g. Store related information on the same or nearby cylinders.
 - Files may get **fragmented** over time
 - ▶ E.g. if data is inserted to/deleted from the file
 - ▶ Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
 - ▶ Sequential access to a fragmented file results in increased disk arm movement
 - Some systems have utilities to **defragment** the file system, in order to speed up file access

Optimization of Disk Block Access (Cont.)

- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
 - Non-volatile RAM: battery backed up RAM or flash memory
 - ▶ Even if power fails, the data is safe and will be written to disk when power returns
 - Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
 - Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - ▶ Write to log disk is very fast since no seeks are required
 - ▶ No need for special hardware (NV-RAM)
- File systems typically reorder writes to disk to improve performance
 - **Journaling file systems** write data in safe order to NV-RAM or log disk
 - Reordering without journaling: risk of corruption of file system data

Flash Storage

- NOR flash vs NAND flash
- NAND flash
 - used widely for storage, since it is much cheaper than NOR flash
 - requires page-at-a-time read (page: 512 bytes to 4 KB)
 - transfer rate around 20 MB/sec
 - **solid state disks**: use multiple flash storage devices to provide higher transfer rate of 100 to 200 MB/sec
 - erase is very slow (1 to 2 millisecs)
 - ▶ erase block contains multiple pages
 - ▶ **remapping** of logical page addresses to physical page addresses avoids waiting for erase
 - **translation table** tracks mapping
 - » also stored in a label field of flash page
 - remapping carried out by **flash translation layer**
 - ▶ after 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
 - **wear leveling**

RAID

■ RAID: Redundant Arrays of Independent Disks

- disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
 - ▶ high capacity and high speed by using multiple disks in parallel,
 - ▶ high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
 - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
 - Techniques for using redundancy to avoid data loss are critical with large numbers of disks
- Originally a cost-effective alternative to large, expensive disks
 - I in RAID originally stood for ``inexpensive''
 - Today RAIDs are used for their higher reliability and bandwidth.
 - ▶ The “I” is interpreted as independent

Improvement of Reliability via Redundancy

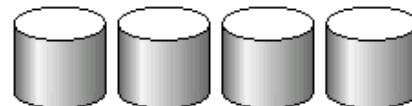
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
 - Duplicate every disk. Logical disk consists of two physical disks.
 - Every write is carried out on both disks
 - ▶ Reads can take place from either disk
 - If one disk in a pair fails, data still available in the other
 - ▶ Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
 - Probability of combined event is very small
 - » Except for dependent failure modes such as fire or building collapse or electrical power surges
- **Mean time to data loss** depends on mean time to failure, and **mean time to repair**
 - E.g. MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500×10^6 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 1. Load balance multiple small accesses to increase throughput
 2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
 - In an array of eight disks, write bit i of each byte to disk i .
 - Each access can read data at eight times the rate of a single disk.
 - But seek/access time worse than for a single disk
 - ▶ Bit level striping is not used much any more
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$
 - Requests for different blocks can run in parallel if the blocks reside on different disks
 - A request for a long sequence of blocks can utilize all disks in parallel

RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
 - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0:** Block striping; non-redundant.
 - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
 - Offers best write performance.
 - Popular for applications such as storing log files in a database system.



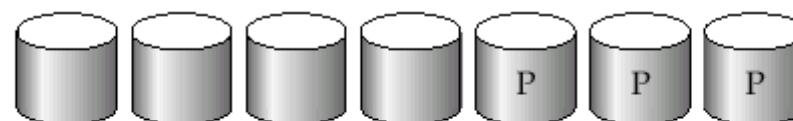
(a) RAID 0: nonredundant striping



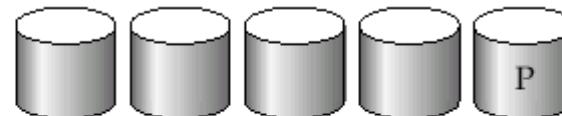
(b) RAID 1: mirrored disks

RAID Levels (Cont.)

- **RAID Level 2:** Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- **RAID Level 3:** Bit-Interleaved Parity
 - a single parity bit is enough for error correction, not just detection, since we know which disk has failed
 - ▶ When writing data, corresponding parity bits must also be computed and written to a parity bit disk
 - ▶ To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)



(c) RAID 2: memory-style error-correcting codes



(d) RAID 3: bit-interleaved parity

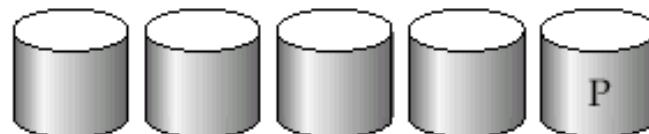
RAID Levels (Cont.)

■ RAID Level 3 (Cont.)

- Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.
- Subsumes Level 2 (provides all its benefits, at lower cost).

■ RAID Level 4: Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from N other disks.

- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
- To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.



(e) RAID 4: block-interleaved parity

RAID Levels (Cont.)

■ RAID Level 4 (Cont.)

- Provides higher I/O rates for independent block reads than Level 3
 - ▶ block read goes to a single disk, so blocks stored on different disks can be read in parallel
- Provides high transfer rates for reads of multiple blocks than no-striping
- Before writing a block, parity data must be computed
 - ▶ Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
 - ▶ Or by recomputing the parity value using the new values of blocks corresponding to the parity block
 - More efficient for writing large amounts of data sequentially
- Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - E.g., with 5 disks, parity block for n th set of blocks is stored on disk $(n \bmod 5) + 1$. with the data blocks stored on the other 4 disks.



(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

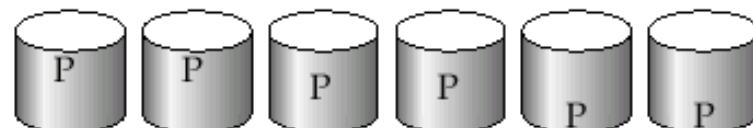
RAID Levels (Cont.)

■ RAID Level 5 (Cont.)

- Higher I/O rates than Level 4.
 - ▶ Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.

■ RAID Level 6: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.

- Better reliability than Level 5 at a higher cost; not used as widely.



(g) RAID 6: P + Q redundancy

Choice of RAID Level

- Factors in choosing RAID level
 - Monetary cost
 - Performance: Number of I/O operations per second, and bandwidth during normal operation
 - Performance during failure
 - Performance during rebuild of failed disk
 - ▶ Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
 - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications

Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
 - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
 - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
 - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
 - I/O requirements have increased greatly, e.g. for Web servers
 - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
 - ▶ so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications

Hardware Issues

- **Software RAID:** RAID implementations done entirely in software, with no special hardware support
- **Hardware RAID:** RAID implementations with special hardware
 - Use non-volatile RAM to record writes that are being executed
 - Beware: power failure during write can result in corrupted disk
 - ▶ E.g. failure after writing one block but before writing the second in a mirrored system
 - ▶ Such corrupted data must be detected when power is restored
 - Recovery from corruption is similar to recovery from failed disk
 - NV-RAM helps to efficiently detect potentially corrupted blocks
 - » Otherwise all blocks of disk must be read and compared with mirror/parity block

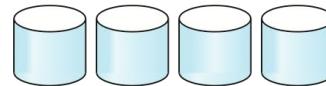
Hardware Issues (Cont.)

- **Latent failures:** data successfully written earlier gets damaged
 - can result in data loss even if only one disk fails
- **Data scrubbing:**
 - continually scan for latent failures, and recover from copy/parity
- **Hot swapping:** replacement of disk while system is running, without power down
 - Supported by some hardware RAID systems,
 - reduces time to recovery, and improves availability greatly
- Many systems maintain **spare disks** which are kept online, and used as replacements for failed disks immediately on detection of failure
 - Reduces time to recovery greatly
- Many hardware RAID systems ensure that a single point of failure will not stop the functioning of the system by using
 - Redundant power supplies with battery backup
 - Multiple controllers and multiple interconnections to guard against controller/interconnection failures

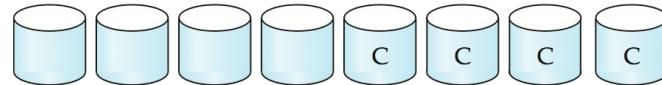
Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
 - Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
 - Transfer rates from few to 10s of MB/s
- Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
 - limited to sequential access.
 - Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
 - Multiple petabytes (10^{15} bytes)

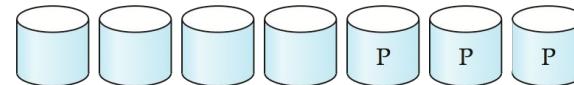
Figure 10.03



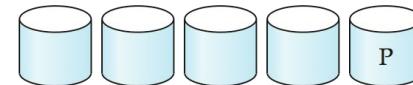
(a) RAID 0: nonredundant striping



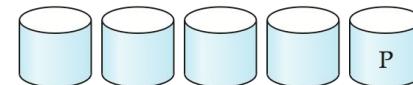
(b) RAID 1: mirrored disks



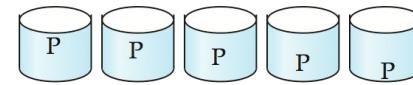
(c) RAID 2: memory-style error-correcting codes



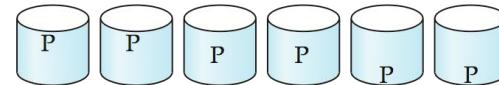
(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

Figure 10.18

Disk 1	Disk 2	Disk 3	Disk 4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
:	:	:	:

Figure in-10.1

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer \$50 from account A to account B:
 1. **read**(A)
 2. $A := A - 50$
 3. **write**(A)
 4. **read**(B)
 5. $B := B + 50$
 6. **write**(B)
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions

Required Properties of a Transaction

- Consider a transaction to transfer \$50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**
- **Atomicity requirement**
 - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
 - ▶ Failure could be due to software or hardware
 - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Required Properties of a Transaction (Cont.)

- **Consistency requirement** in above example:
 - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
 - ▶ Explicitly specified integrity constraints such as primary keys and foreign keys
 - ▶ Implicit integrity constraints
 - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
 - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
<ol style="list-style-type: none">1. read(A)2. $A := A - 50$3. write(A)4. read(B)5. $B := B + 50$6. write(B)	read(A), read(B), print(A+B)

- Isolation can be ensured trivially by running transactions **serially**
 - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

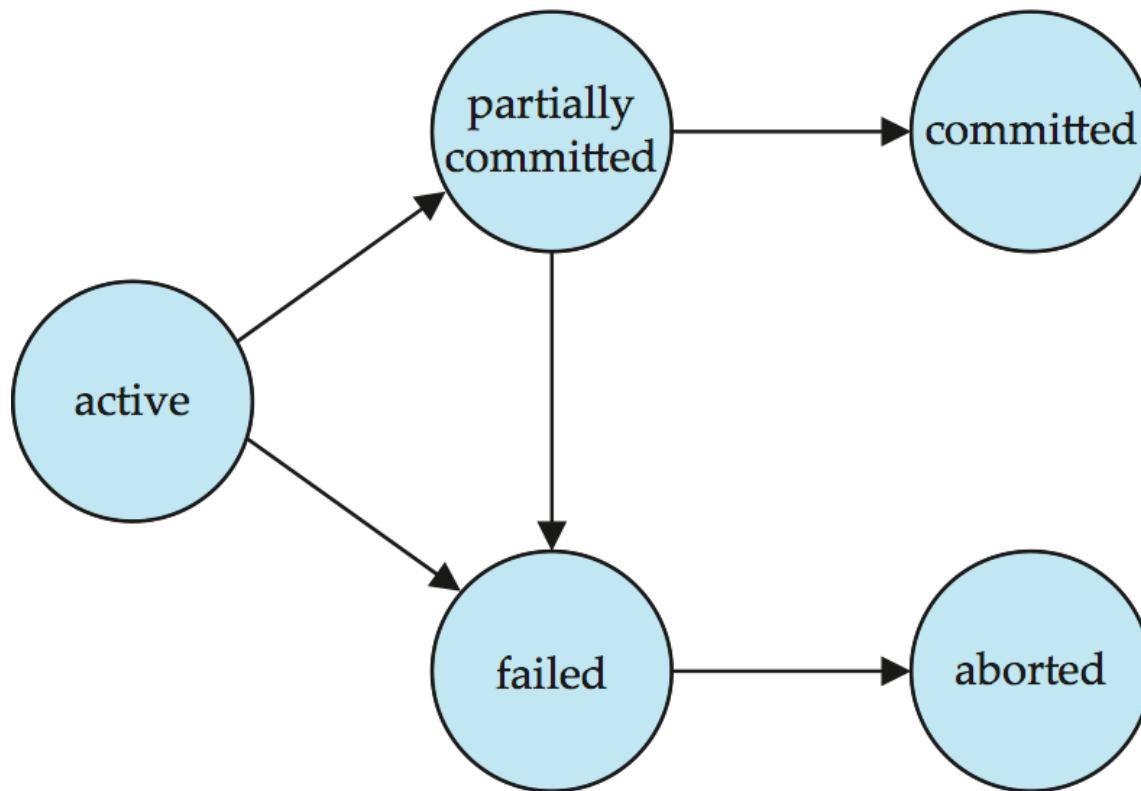
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
 - That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:
 - Restart the transaction
 - ▶ can be done only if no internal logical error
 - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrency Control

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
 1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
 2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

■ Lock-compatibility matrix

	S	X
S	true	false
X	false	false

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

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

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

- Locking as above is not sufficient to guarantee serializability
 - if A and B get updated in-between the read of A and B , the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules.
- 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
- 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).

The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
 - Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Lock Conversions

- Two-phase locking with lock conversions:
 - First Phase:
 - can acquire a lock-S on item
 - can acquire a lock-X on item
 - can convert a lock-S to a lock-X (upgrade)
 - Second Phase:
 - can release a lock-S
 - can release a lock-X
 - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read(D)** is processed as:

```
if  $T_i$  has a lock on  $D$ 
  then
    read( $D$ )
  else begin
    if necessary wait until no other
      transaction has a lock-X on  $D$ 
    grant  $T_i$  a lock-S on  $D$ ;
    read( $D$ )
  end
```

Automatic Acquisition of Locks (Cont.)

- **write(D)** is processed as:
 - if T_i has a **lock-X** on D
 - then**
 - write(D)**
 - else begin**
 - if necessary wait until no other transaction has any lock on D ,
 - if T_i has a **lock-S** on D
 - then**
 - upgrade lock on D to lock-X**
 - else**
 - grant T_i a lock-X on D**
 - write(D)**
 - end;**
 - All locks are released after commit or abort

Deadlocks

- 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.

Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks, there is a possibility of **starvation**.
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
 - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
 - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

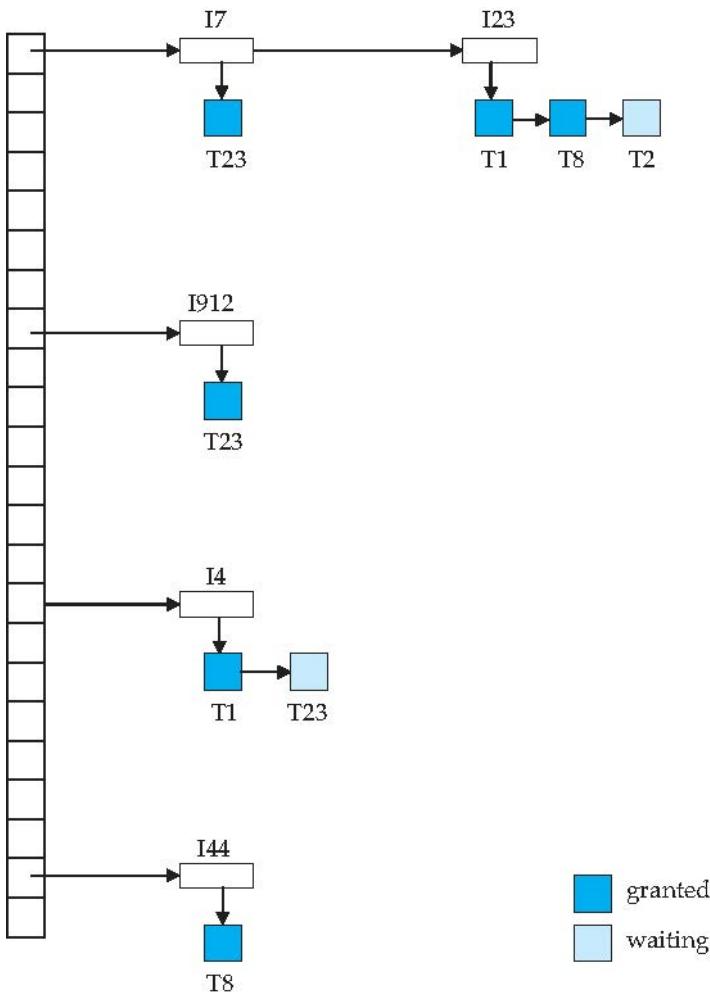
Deadlocks (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

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



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
 - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
 - Require that each transaction locks all its data items before it begins execution (predeclaration).
 - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
 - older transaction may wait for younger one to release data item.
(older means smaller timestamp) Younger transactions never
Younger transactions never wait for older ones; they are rolled
back instead.
 - a transaction may die several times before acquiring needed data
item
- **wound-wait** scheme — preemptive
 - older transaction *wounds* (forces rollback) of younger transaction
instead of waiting for it. Younger transactions may wait for older
ones.
 - may be fewer rollbacks than *wait-die* scheme.

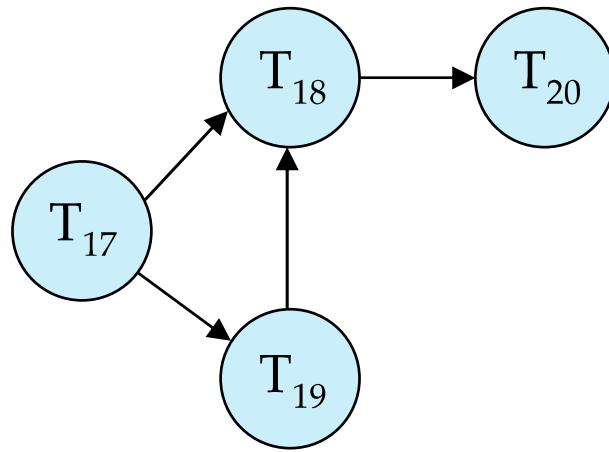
Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **Timeout-Based Schemes:**
 - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
 - Thus, deadlocks are not possible
 - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

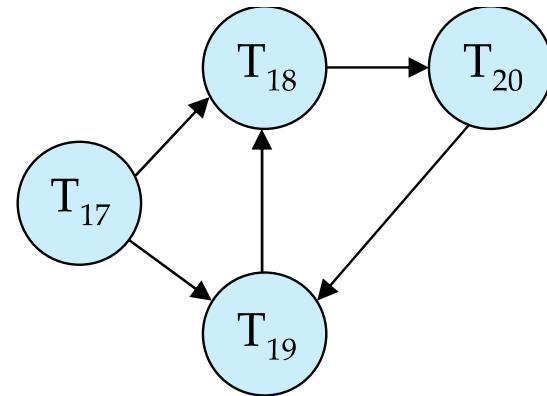
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
 - V is a set of vertices (all the transactions in the system)
 - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E , then there is a directed edge from T_i to T_j , implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i .
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
 - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
 - Rollback -- determine how far to roll back transaction
 - ▶ Total rollback: Abort the transaction and then restart it.
 - ▶ More effective to roll back transaction only as far as necessary to break deadlock.
 - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp $\text{TS}(T_i)$, a new transaction T_j is assigned time-stamp $\text{TS}(T_j)$ such that $\text{TS}(T_i) < \text{TS}(T_j)$.
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- 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.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read(Q)**
 1. If $TS(T_i) \leq W\text{-timestamp}(Q)$, 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 $TS(T_i) \geq W\text{-timestamp}(Q)$, then the **read** operation is executed, and $R\text{-timestamp}(Q)$ is set to **max**($R\text{-timestamp}(Q)$, $TS(T_i)$).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write(Q)**.
 1. If $TS(T_i) < R\text{-timestamp}(Q)$, 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 **write** operation is rejected, and T_i is rolled back.
 2. If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q.
 - Hence, this **write** operation is rejected, and T_i is rolled back.
 3. Otherwise, the **write** operation is executed, and $W\text{-timestamp}(Q)$ is set to $TS(T_i)$.

Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

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

Deadlocks

- Consider the following two transactions:

T_1 : write (X)

write(Y)

T_2 : write(Y)

write(X)

- Schedule with deadlock

T_1	T_2
lock-X on A write (A) wait for lock-X on B	lock-X on B write (B) wait for lock-X on A

Concurrency Control and Recovery

- With concurrent transactions, 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 that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
 - i.e. the updates of uncommitted transactions should not be visible to other transactions
 - ▶ Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
 - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

- **Undo** of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **old** value V_1 to X
- **Redo** of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
 - **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 ,
 - ▶ each time a data item X is restored to its old value V a special log record $\langle T_i, X, V \rangle$ is written out
 - ▶ when undo of a transaction is complete, a log record $\langle T_i, \text{abort} \rangle$ is written out.
 - **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
 - ▶ No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
 - Transaction T_i needs to be undone if the log
 - ▶ contains the record $\langle T_i \text{ start} \rangle$,
 - ▶ but does not contain either the record $\langle T_i \text{ commit} \rangle$ or $\langle T_i \text{ abort} \rangle$.
 - Transaction T_i needs to be redone if the log
 - ▶ contains the records $\langle T_i \text{ start} \rangle$
 - ▶ and contains the record $\langle T_i \text{ commit} \rangle$ or $\langle T_i \text{ abort} \rangle$
- Note that If transaction T_i was undone earlier and the $\langle T_i \text{ abort} \rangle$ record written to the log, and then a failure occurs, on recovery from failure T_i is redone
 - such a redo **redoes all the original actions *including the steps that restored old values***
 - ▶ Known as **repeating history**
 - ▶ Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

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

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 1000, 950 \rangle$

$\langle T_0, B, 2000, 2050 \rangle$

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 1000, 950 \rangle$

$\langle T_0, B, 2000, 2050 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 700, 600 \rangle$

$\langle T_0 \text{ start} \rangle$

$\langle T_0, A, 1000, 950 \rangle$

$\langle T_0, B, 2000, 2050 \rangle$

$\langle T_0 \text{ commit} \rangle$

$\langle T_1 \text{ start} \rangle$

$\langle T_1, C, 700, 600 \rangle$

$\langle T_1 \text{ commit} \rangle$

(a)

(b)

(c)

Recovery actions in each case above are:

- (a) undo (T_0): B is restored to 2000 and A to 1000, and log records $\langle T_0, B, 2000 \rangle$, $\langle T_0, A, 1000 \rangle$, $\langle T_0, \text{abort} \rangle$ are written out
- (b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records $\langle T_1, C, 700 \rangle$, $\langle T_1, \text{abort} \rangle$ are written out.
- (c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

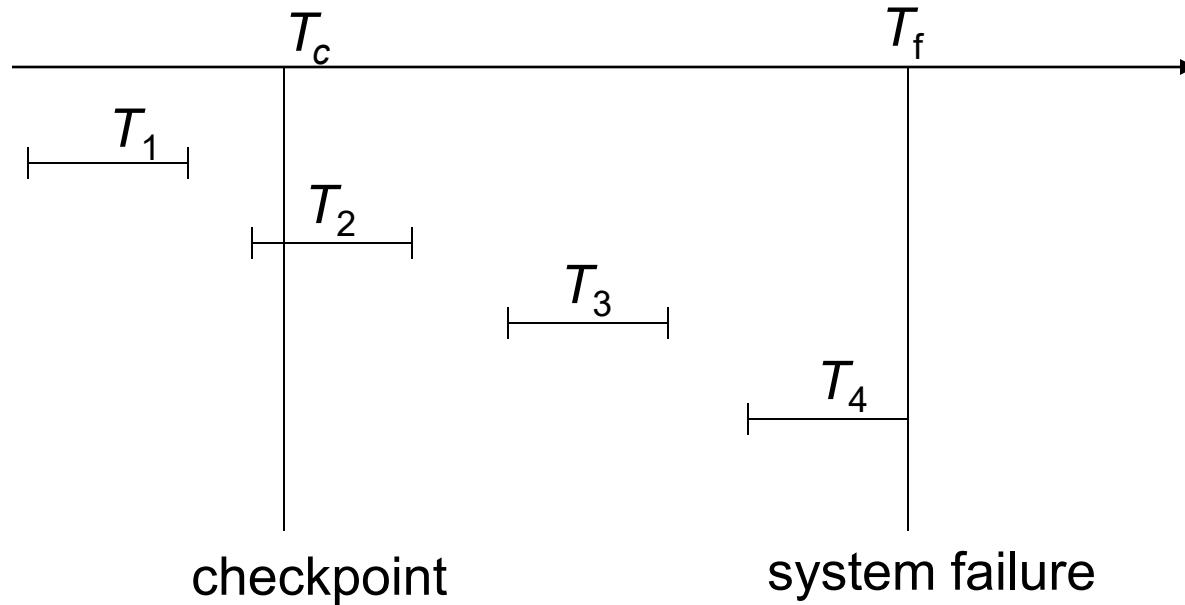
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
 1. processing the entire log is time-consuming if the system has run for a long time
 2. we might unnecessarily redo transactions which have already 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.
 3. Write a log record < **checkpoint L** > onto stable storage where L is a list of all transactions active at the time of checkpoint.
 - All updates are stopped while doing checkpointing

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 .
 1. 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
 1. 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



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

Recovery Algorithm

- **So far:** we covered key concepts
- **Now:** we present the components of the basic recovery algorithm
- **Later:** we present extensions to allow more concurrency

Recovery Algorithm

- **Logging (during normal operation):**

- $\langle T_i \text{ start} \rangle$ at transaction start
 - $\langle T_i, X_j, V_1, V_2 \rangle$ for each update, and
 - $\langle T_i \text{ commit} \rangle$ at transaction end

- **Transaction rollback (during normal operation)**

- Let T_i be the transaction to be rolled back
 - Scan log backwards from the end, and for each log record of T_i of the form $\langle T_i, X_j, V_1, V_2 \rangle$
 - ▶ perform the undo by writing V_1 to X_j ,
 - ▶ write a log record $\langle T_i, X_j, V_1 \rangle$
 - such log records are called **compensation log records**
 - Once the record $\langle T_i \text{ start} \rangle$ is found stop the scan and write the log record $\langle T_i \text{ abort} \rangle$

Recovery Algorithm (Cont.)

■ Recovery from failure: Two phases

- **Redo phase:** replay updates of **all** transactions, whether they committed, aborted, or are incomplete
- **Undo phase:** undo all incomplete transactions

■ Redo phase:

1. Find last <checkpoint L > record, and set undo-list to L .
2. Scan forward from above <checkpoint L > record
 1. Whenever a record $\langle T_i, X_j, V_1, V_2 \rangle$ or $\langle T_i, X_j, V_2 \rangle$ is found, redo it by writing V_2 to X_j
 2. Whenever a log record $\langle T_i, \text{start} \rangle$ is found, add T_i to undo-list
 3. Whenever a log record $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

■ Undo phase:

1. Scan log backwards from end
 1. Whenever a log record $\langle T_i, X_j, V_1, V_2 \rangle$ is found where T_i is in undo-list perform same actions as for transaction rollback:
 1. perform undo by writing V_1 to X_j .
 2. write a log record $\langle T_i, X_j, V_1 \rangle$
 2. Whenever a log record $\langle T_i, \text{start} \rangle$ is found where T_i is in undo-list,
 1. Write a log record $\langle T_i, \text{abort} \rangle$
 2. Remove T_i from undo-list
 3. Stop when undo-list is empty
 - i.e. $\langle T_i, \text{start} \rangle$ has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery

