# COP 4600 – Summer 2011

# Introduction To Operating Systems

## Mass Storage Systems


Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2011

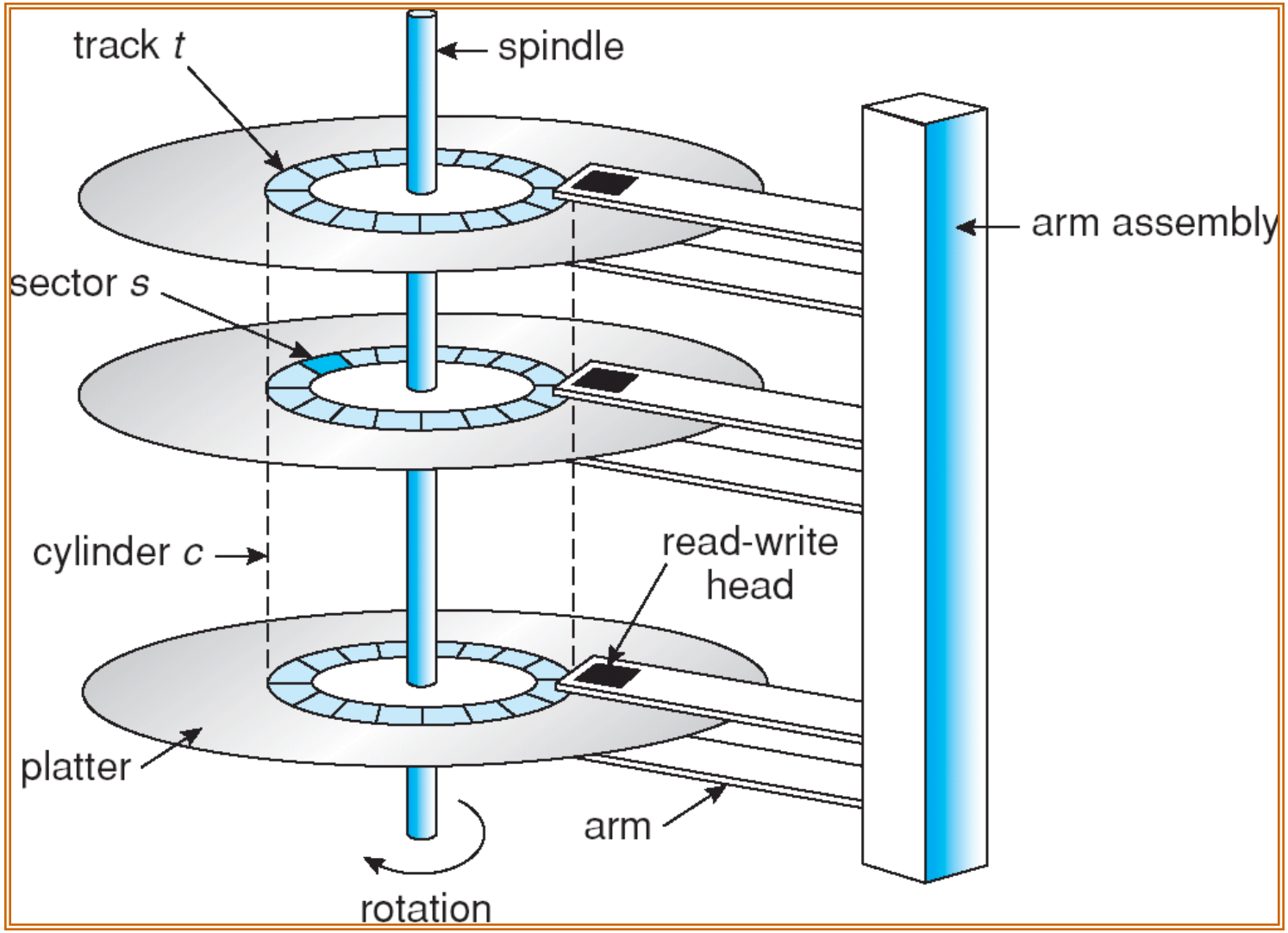
Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida


# Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 200 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
    - That's bad
- Disks can be removable
- Drive attached to computer via **I/O bus**
  - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
  - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array



# Moving-head Disk Mechanism






# Overview of Mass Storage Structure

- Magnetic tape
  - Was early secondary-storage medium
  - Relatively permanent and holds large quantities of data
  - Access time slow
  - Random access ~1000 times slower than disk
  - Mainly used for backup, storage of infrequently-used data, transfer medium between systems
  - Kept in spool and wound or rewound past read-write head
  - Once data under head, transfer rates comparable to disk
  - 20-200GB typical storage
  - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT



# Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.

- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.



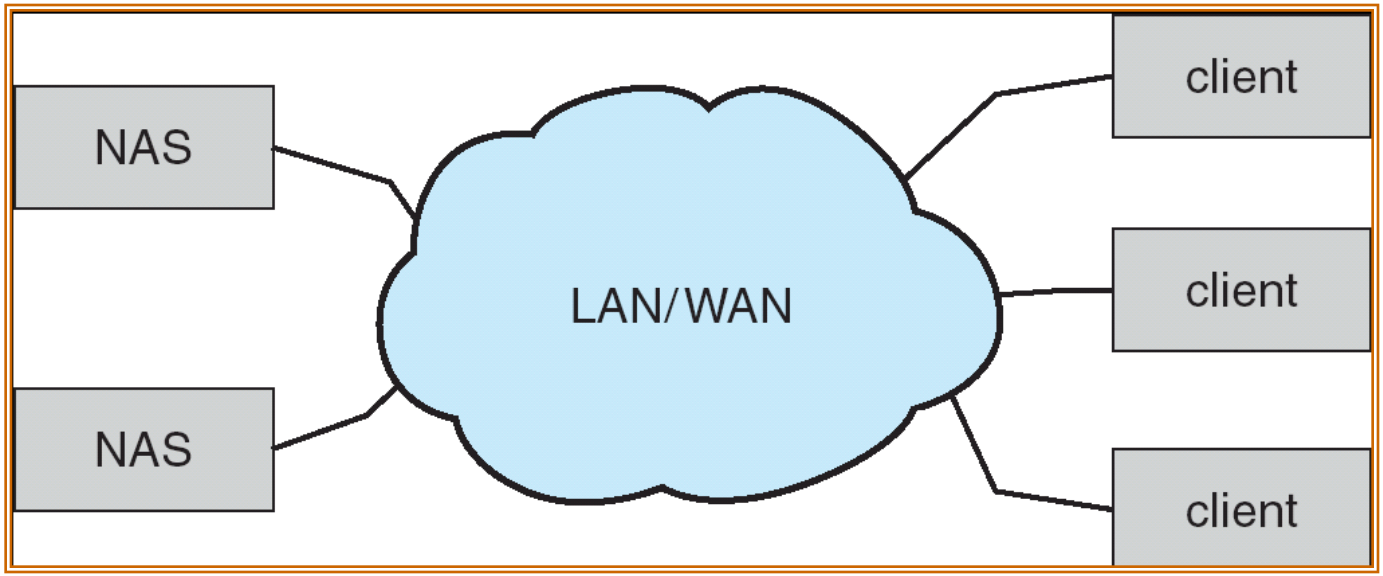
# Disk Attachment

- Host-attached storage accessed through I/O ports talking to I/O busses
- SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks
  - Each target can have up to 8 **logical units** (disks attached to device controller
- FC is high-speed serial architecture
  - Can be switched fabric with 24-bit address space – the basis of **storage area networks** (**SAN**s) in which many hosts attach to many storage units
  - Can be **arbitrated loop** (**FC-AL**) of 126 devices



# Network-Attached Storage

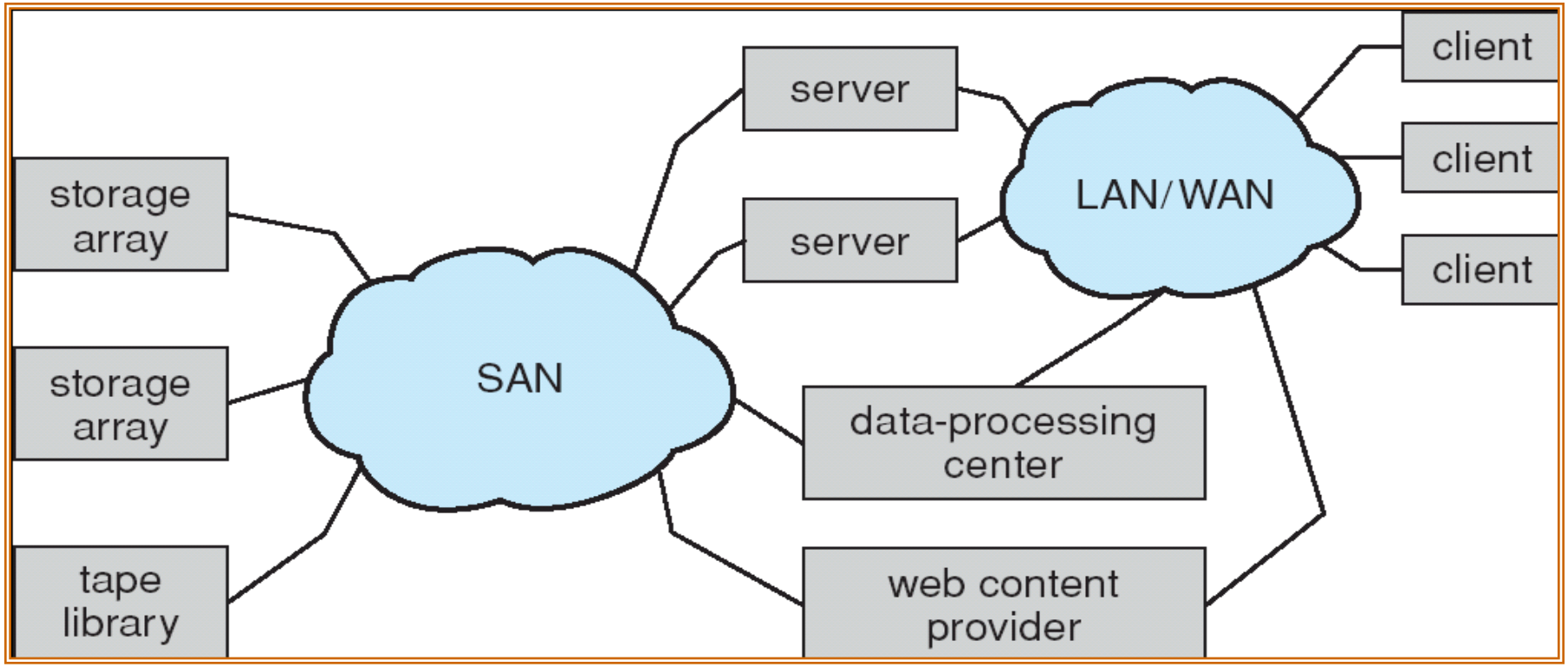
- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage
- New iSCSI protocol uses IP network to carry the SCSI protocol






# Storage Area Network

- Common in large storage environments (and becoming more common)
- Multiple hosts attached to multiple storage arrays - flexible






# Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.
  - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time ≈ seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.



# Disk Scheduling

- Several algorithms exist to schedule the servicing of disk I/O requests.
- The following example illustrate the various scheduling algorithms using a request queue (0-199) as follows with the current position pointer for the disk head on track 53.

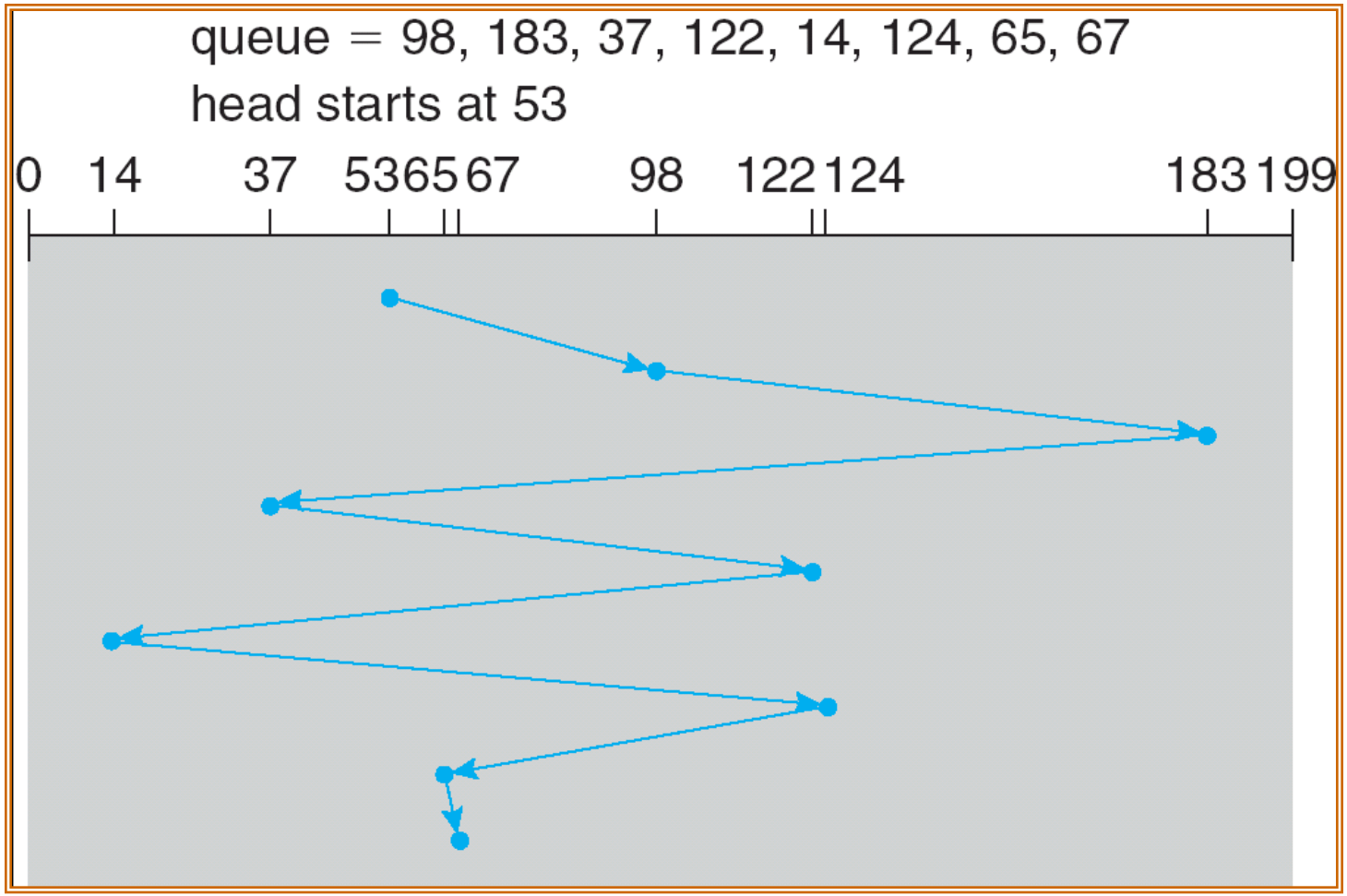
98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53



# FCFS – First Come First Served

Illustration shows total head movement of 640 cylinders.



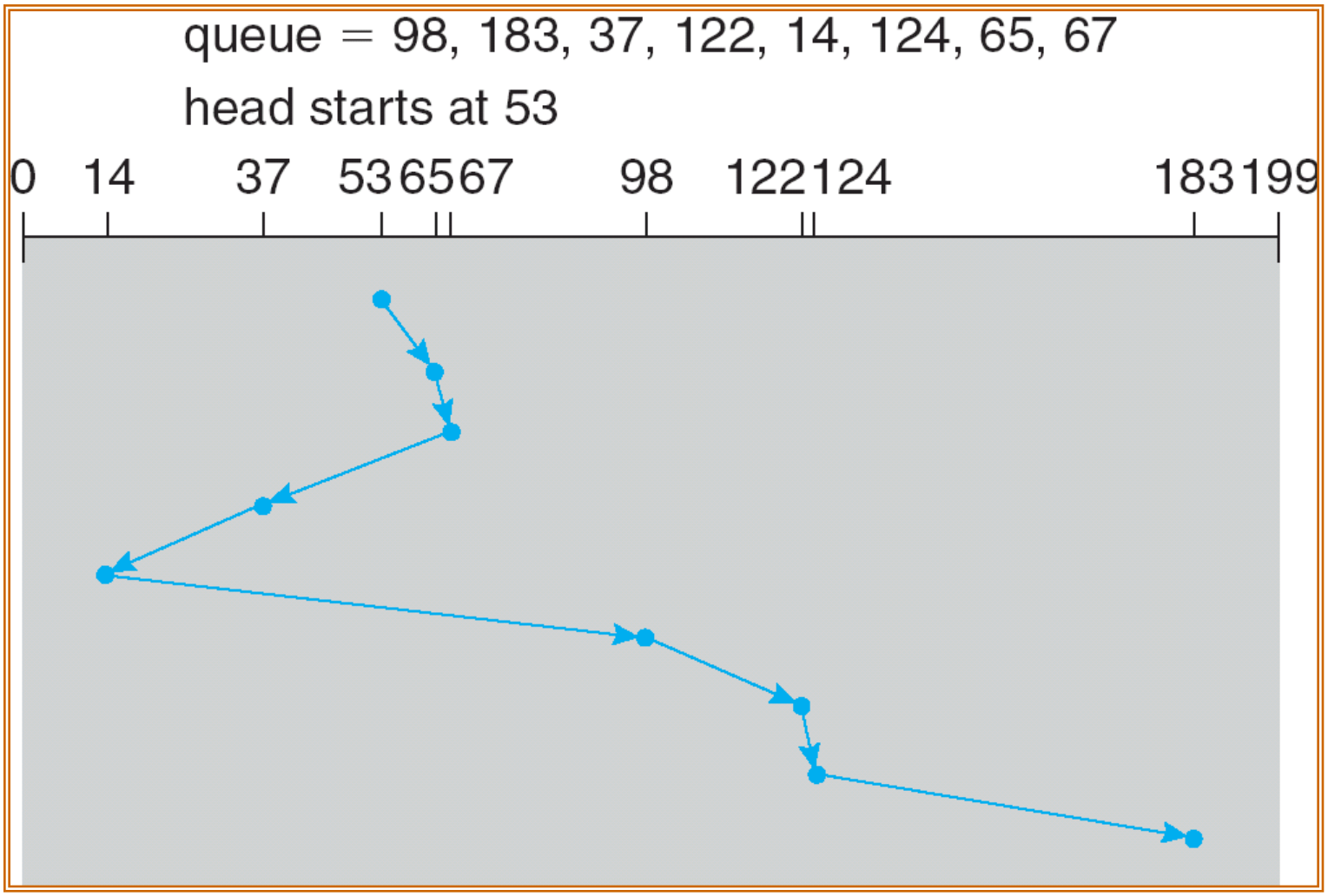



# SSTF – Shortest Seek Time First

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.



# SSTF


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

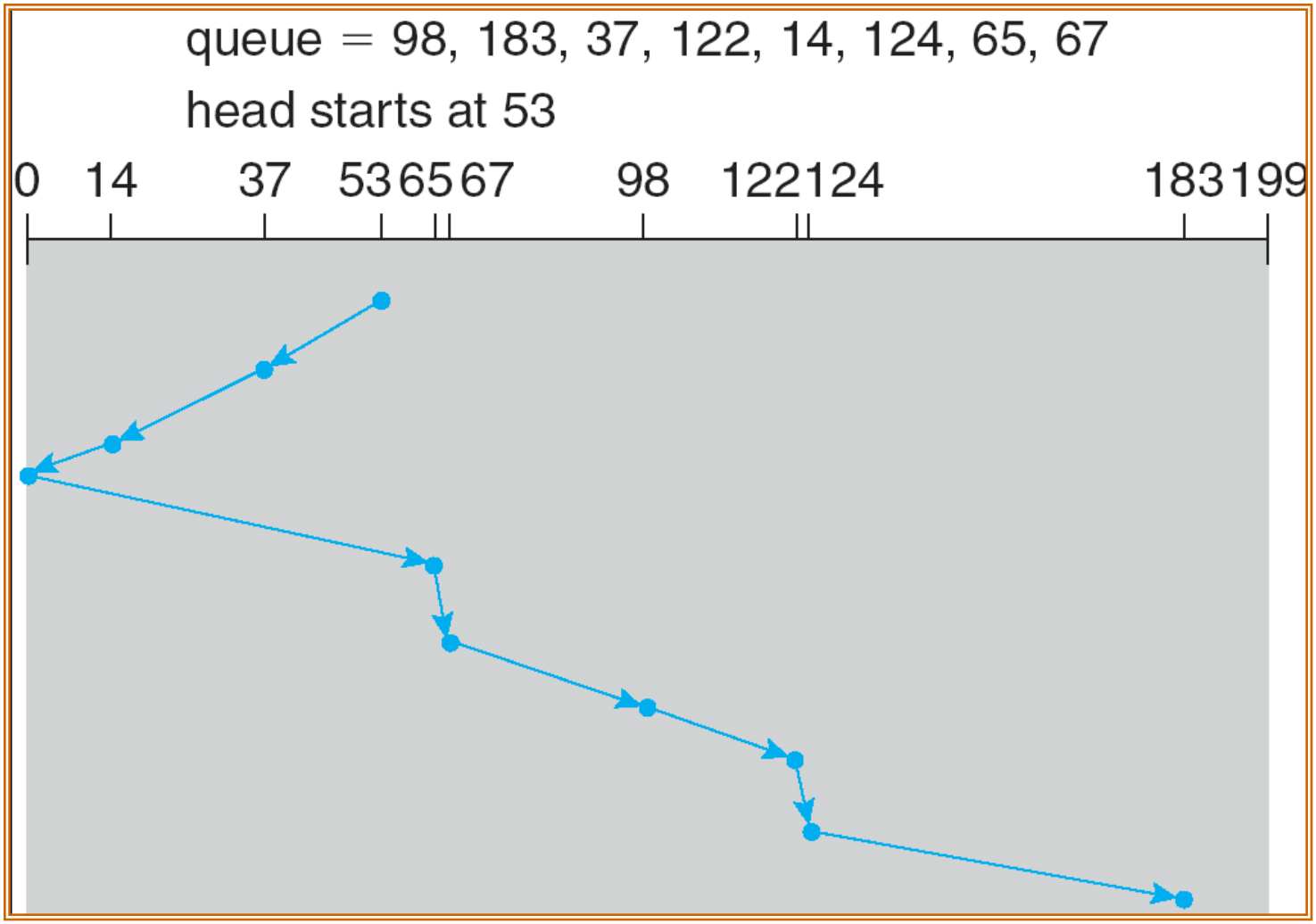


# SCAN – Elevator Algorithm

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.



# SCAN


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

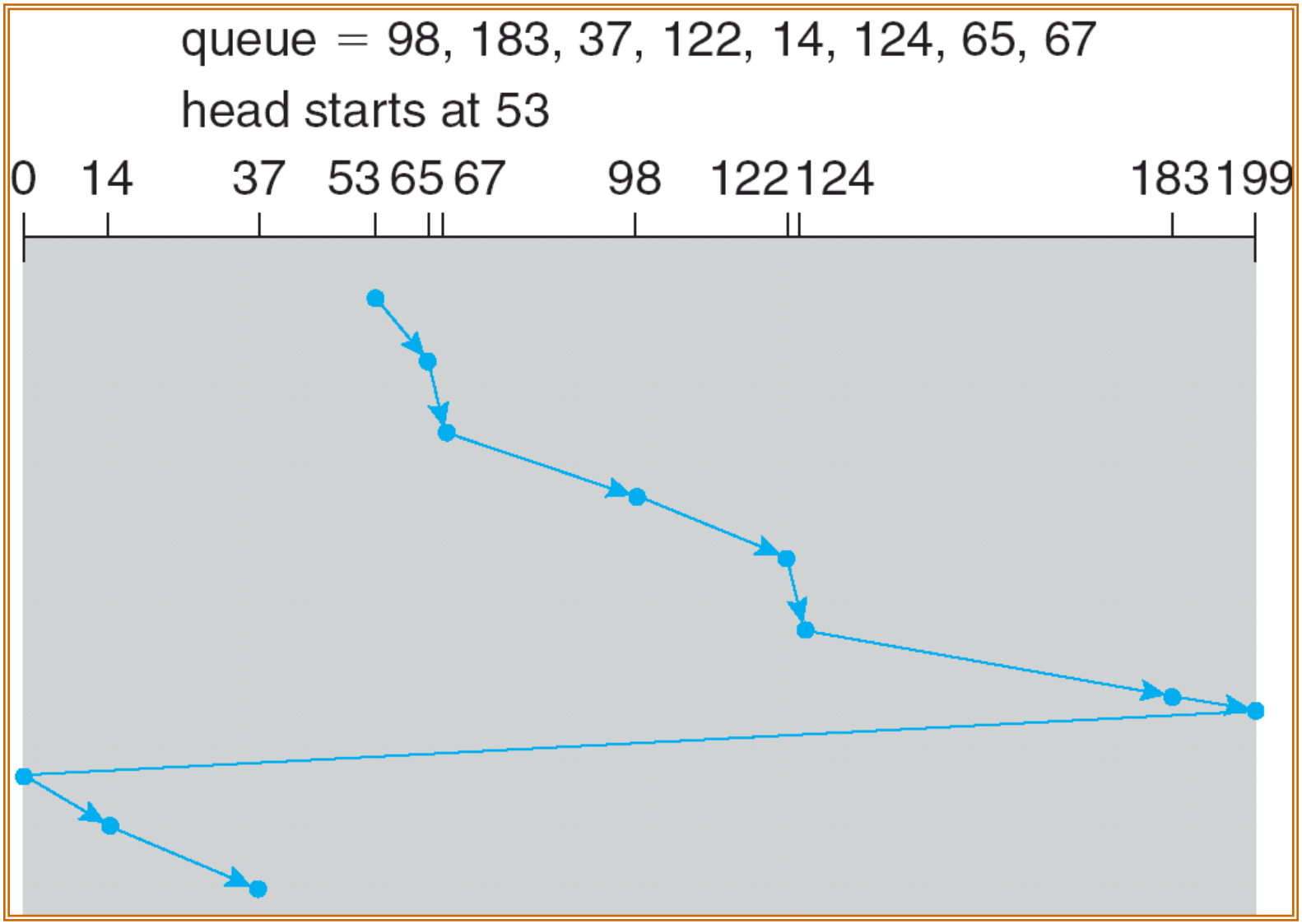


# C-SCAN

- C-SCAN (Circular SCAN) Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.



# C-SCAN


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

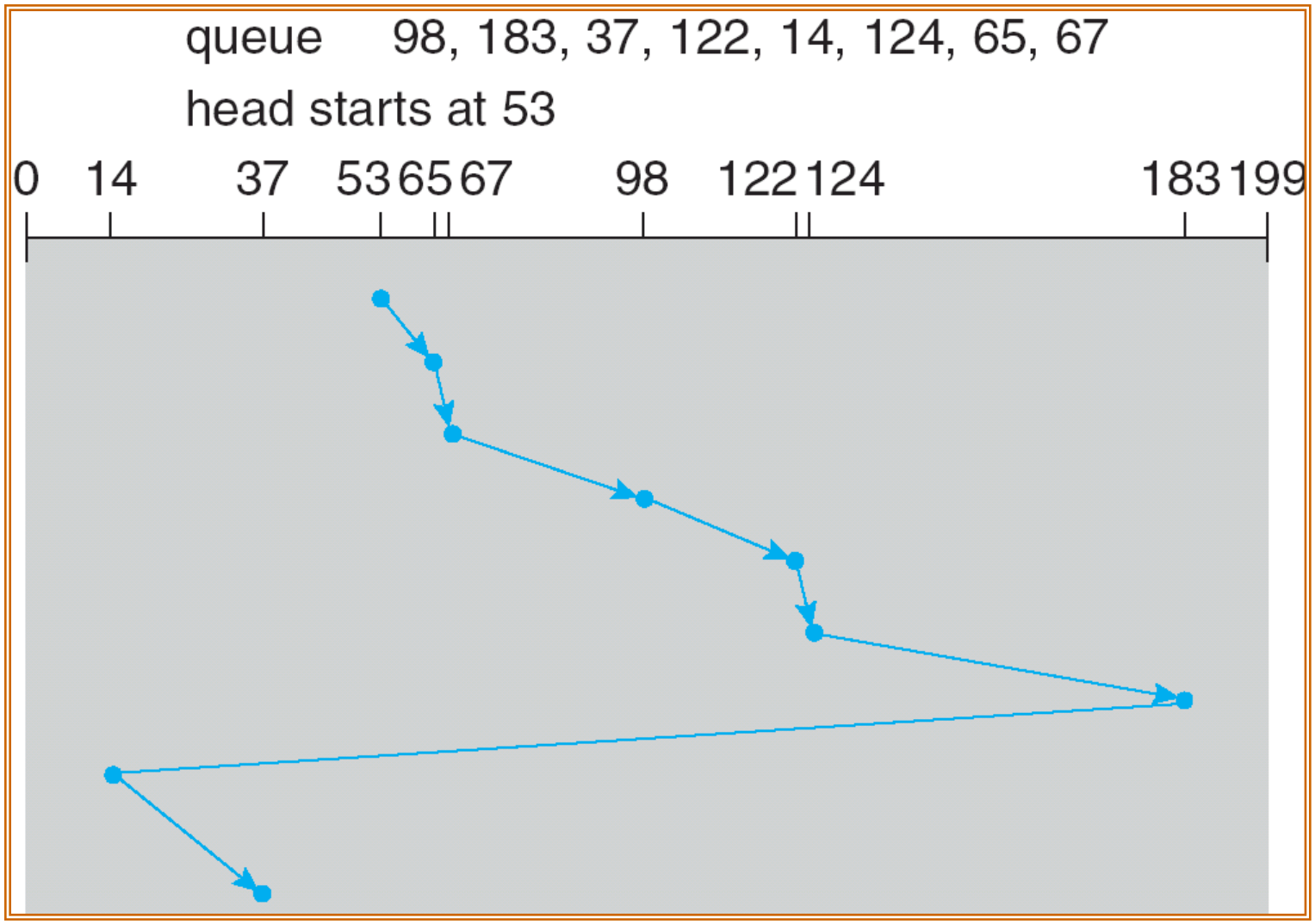


# C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



# C-LOOK


queue 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199



# Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

# Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
  - *Partition* the disk into one or more groups of cylinders.
  - *Logical formatting* or "making a file system".
- Boot block initializes system.
  - The bootstrap is stored in ROM.
  - *Bootstrap loader* program.
- Methods such as *sector sparing* used to handle bad blocks.

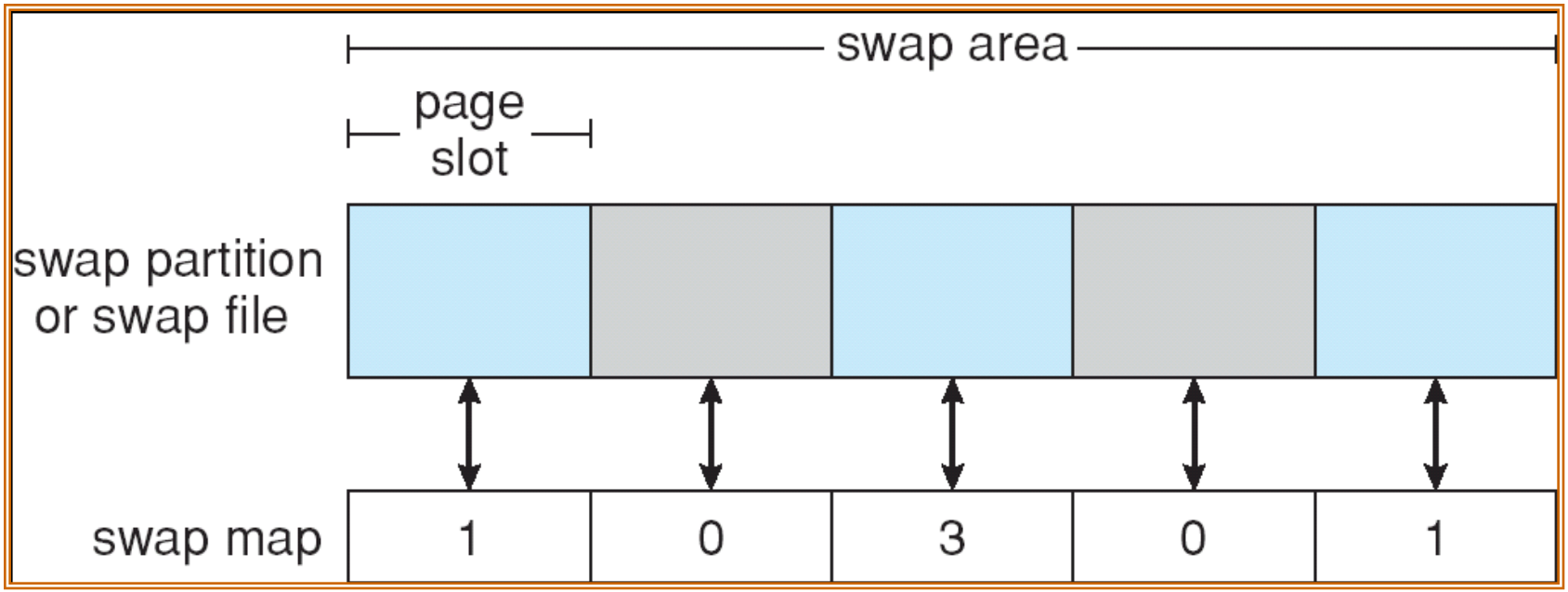


# Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space can be carved out of the normal file system,or, more commonly, it can be in a separate disk partition.
- Swap-space management
  - 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment.*
  - Kernel uses *swap maps* to track swap-space use.
  - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.



# Data Structures for Swapping on Linux Systems



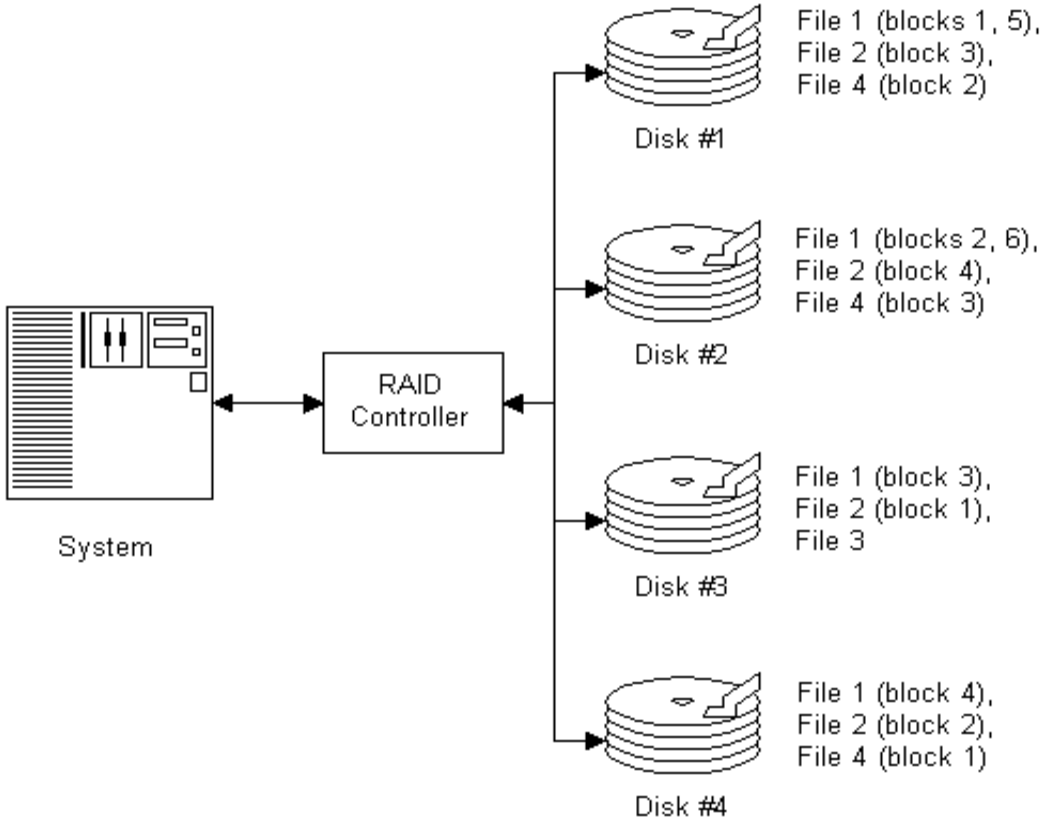



# RAID

- **RAID: Redundant Arrays of Independent (Inexpensive) Disks (Devices)**
  - 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, and
    - 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. 11.5 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



# RAID (cont.)




Block diagram of a RAID striping configuration. One controller (which can be hardware or software) splits files into blocks or bytes and distributes them across several hard disks. The block size determines how many "pieces" files will be split into. In this example, the first block of file 1 is sent to disk #1, then the second block to disk #2, etc. When all four disks have one block of file 1, the fifth block goes back to disk #1, and this continues until the file is completed. Note that file 3 is only on one disk; this means it was smaller than the block size in this case.

# RAID (cont.)

- Originally a cost-effective alternative to large, expensive disks
  - The “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 Through Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- **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 of Reliability Through 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* mod *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

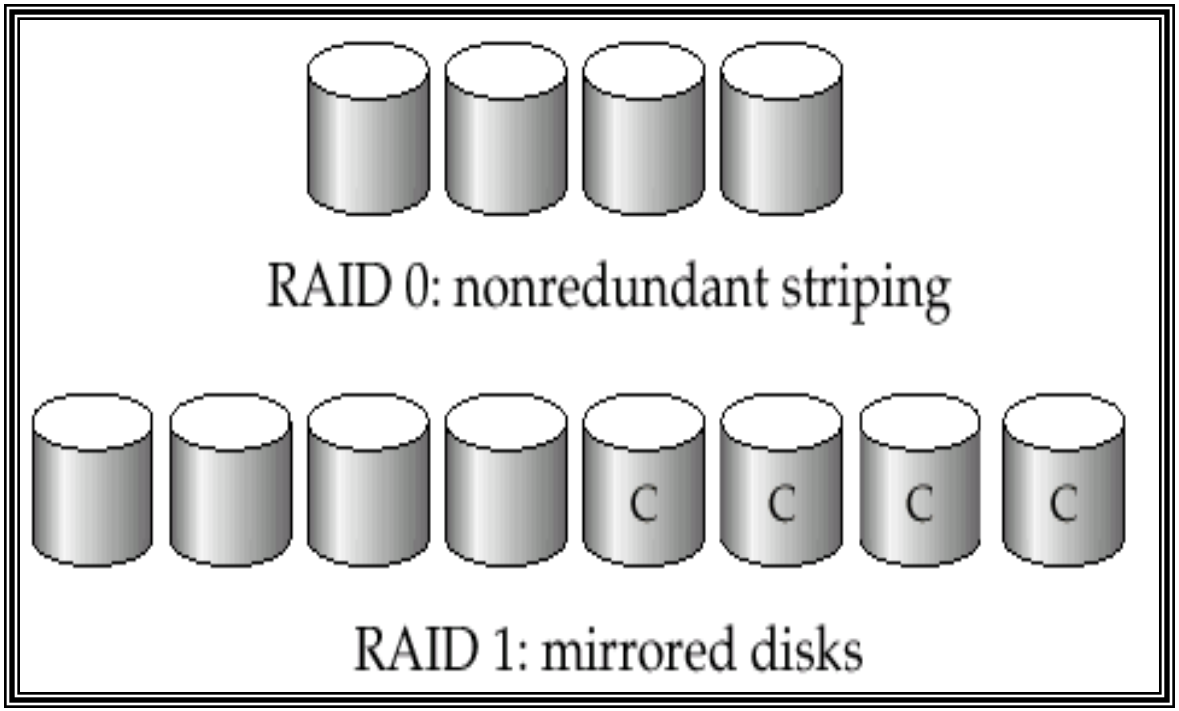


# RAID Levels

- 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.

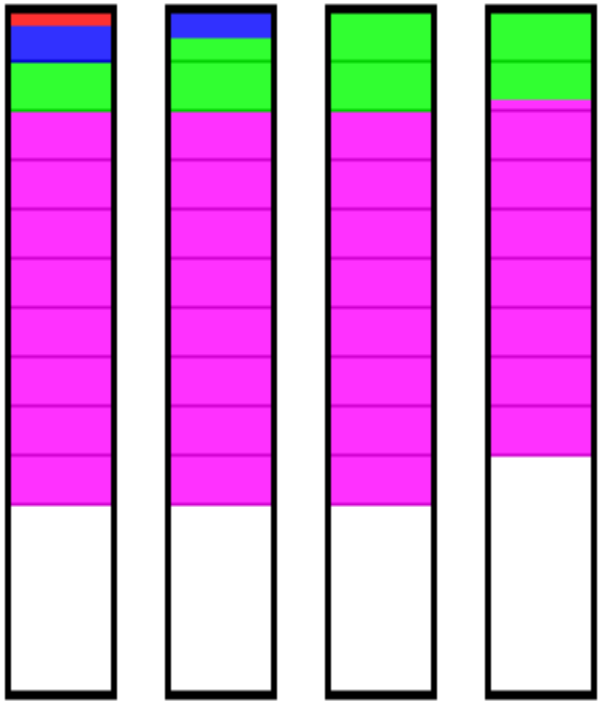


# RAID Levels (cont.)






# RAID Levels (cont.)

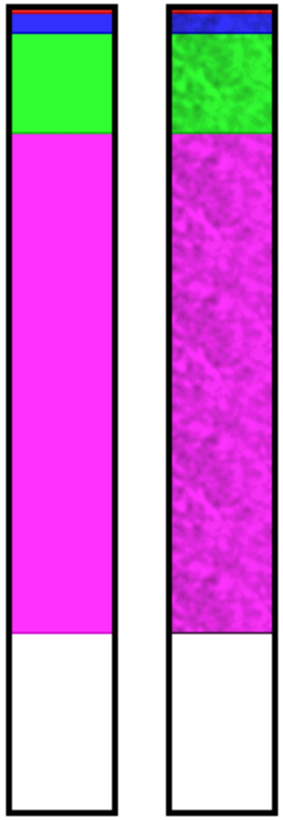


Raid 0

This illustration shows how files of different sizes are distributed between the drives on a four-disk, 16 kB stripe size RAID 0 array. The red file is 4 kB in size; the blue is 20 kB; the green is 100 kB; and the magenta is 500 kB. They are shown *drawn to scale* to illustrate how much space they take up in relative terms in the array--one vertical pixel represents 1 kB. Thus, the 500 kB files needs a total of 31+ stripes, the 100 kB file needs 6+ stripes.



# RAID Levels (cont.)



Raid 1

Illustration of a pair of mirrored hard disks, showing how the files are duplicated on both drives. (The files are the same as those in the RAID 0 illustration on the previous page, except that the scale is reduced here so one vertical pixel represents 2 kB.)



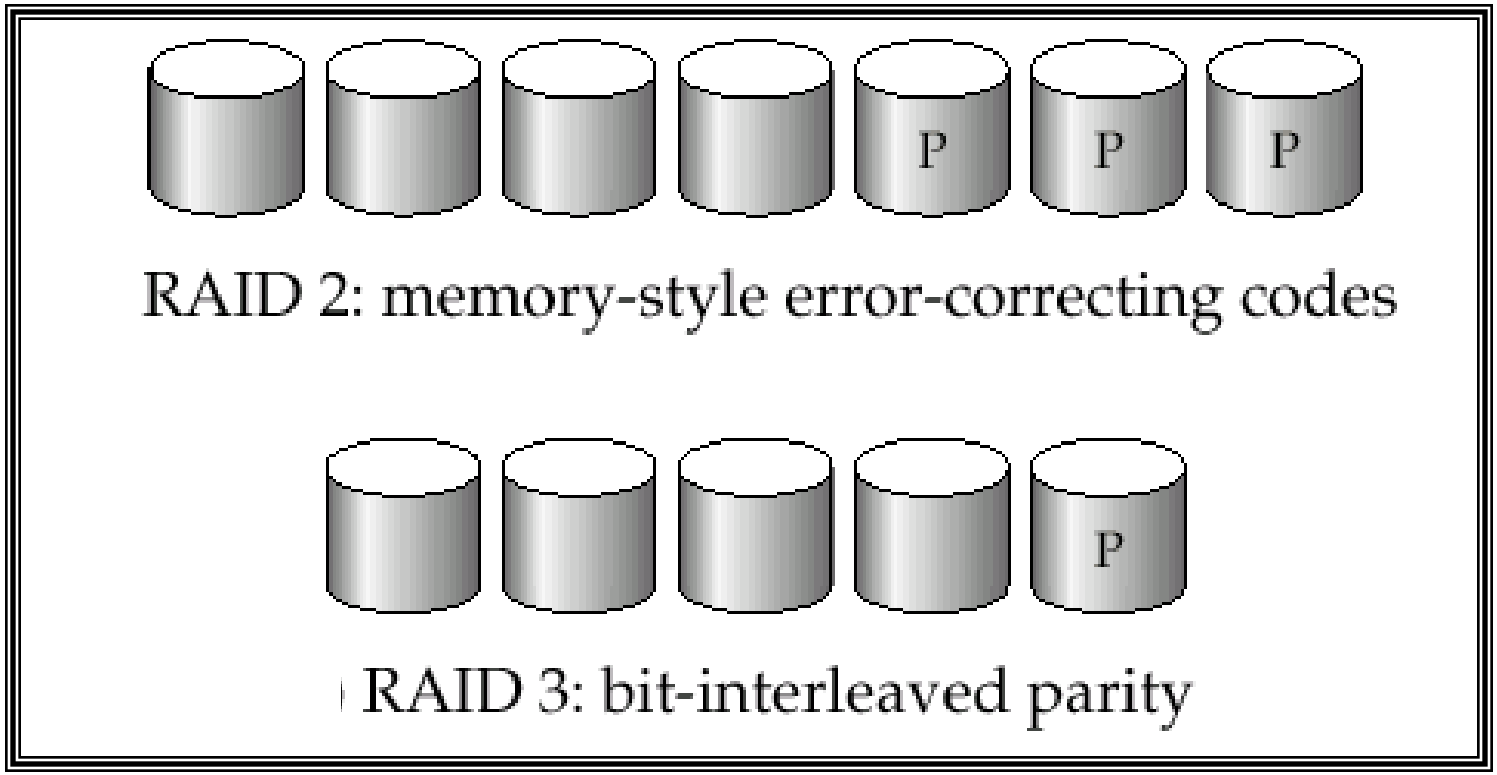
# RAID Levels (cont.)

- **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)
  - 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).

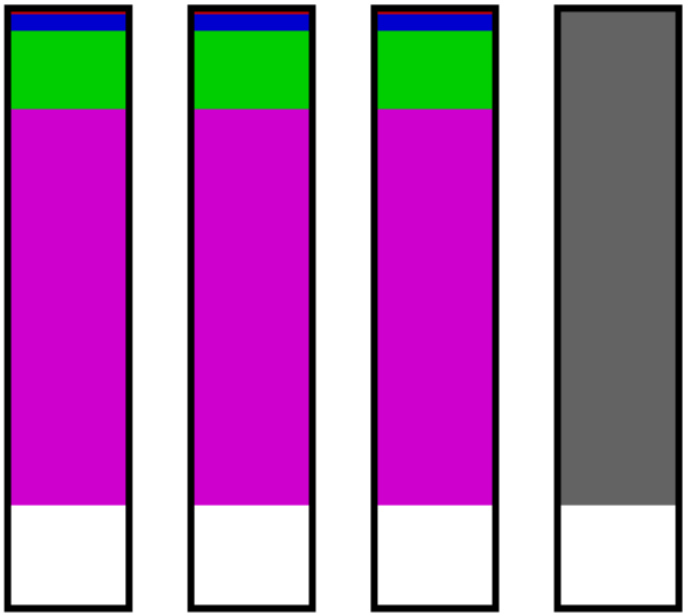


# RAID Levels (cont.)






# RAID Levels (cont.)



Raid 3

This illustration shows how files of different sizes are distributed between the drives on a four-disk, byte-striped RAID 3 array. As with the RAID 0 illustration, the red file is 4 kB in size; the blue is 20 kB; the green is 100 kB; and the magenta is 500 kB, with each vertical pixel representing 1 kB of space. Notice that the files are evenly spread between three drives, with the fourth containing parity information (shown in dark gray). Since the blocks are so tiny in RAID 3, the individual boundaries between stripes can't be seen. You may want to compare this illustration to the one for RAID 4 on page 34.

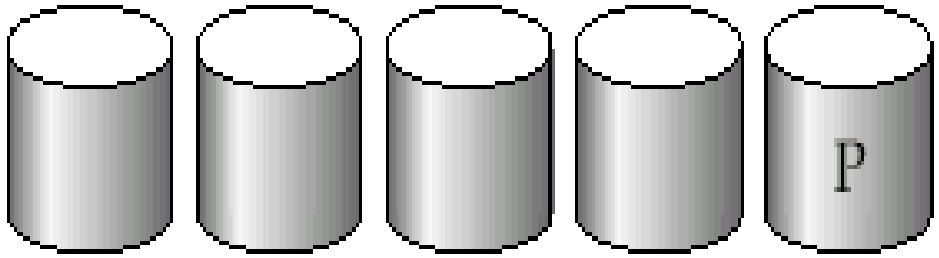


# RAID Levels (cont.)

- **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.
  - 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.



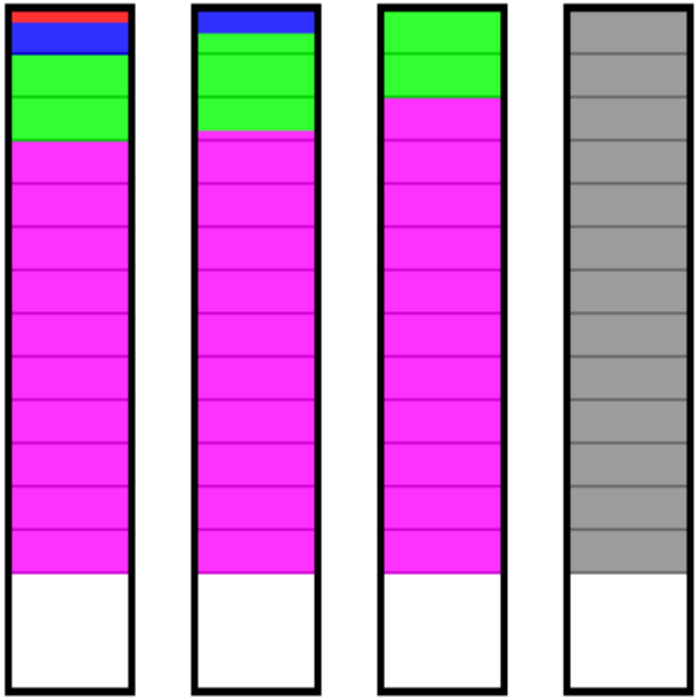
# RAID Levels (cont.)




RAID 4: block-interleaved parity



# RAID Levels (cont.)



Raid 4

This illustration shows how files of different sizes are distributed between the drives on a four-disk RAID 4 array using a 16 kB stripe size. As with the RAID 0 illustration, the red file is 4 kB in size; the blue is 20 kB; the green is 100 kB; and the magenta is 500 kB, with each vertical pixel representing 1 kB of space. Notice that as with RAID 3, the files are evenly spread between three drives, with the fourth containing parity information (shown in gray). You may want to contrast this illustration to the one for RAID 3 (which is very similar except that the blocks are so tiny you can't see them) and the one

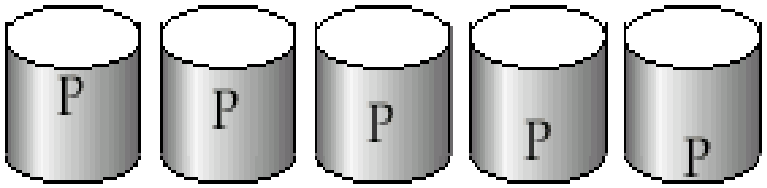


# RAID Levels (cont.)

- **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.
  - 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.



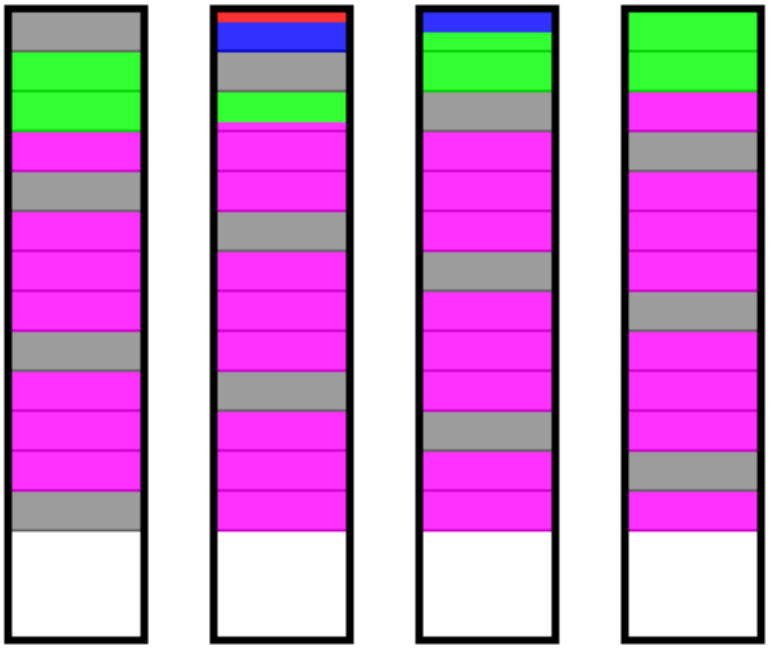
# RAID Levels (cont.)




(f) RAID 5: block-interleaved distributed parity



# RAID Levels (cont.)



Raid 5

This illustration shows how files of different sizes are distributed between the drives on a four-disk RAID 5 array using a 16 kB stripe size. As with the RAID 0 illustration, the red file is 4 kB in size; the blue is 20 kB; the green is 100 kB; and the magenta is 500 kB, with each vertical pixel representing 1 kB of space. Contrast this diagram to the one for RAID 4, which is identical except that the data is only on three drives and the parity (shown in gray) is exclusively on the fourth drive.

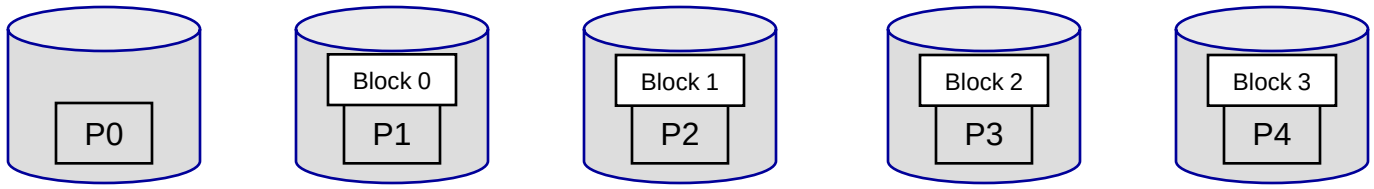


# RAID Level 5 Example

| | | | | |
|---|---|---|---|---|
| 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 |

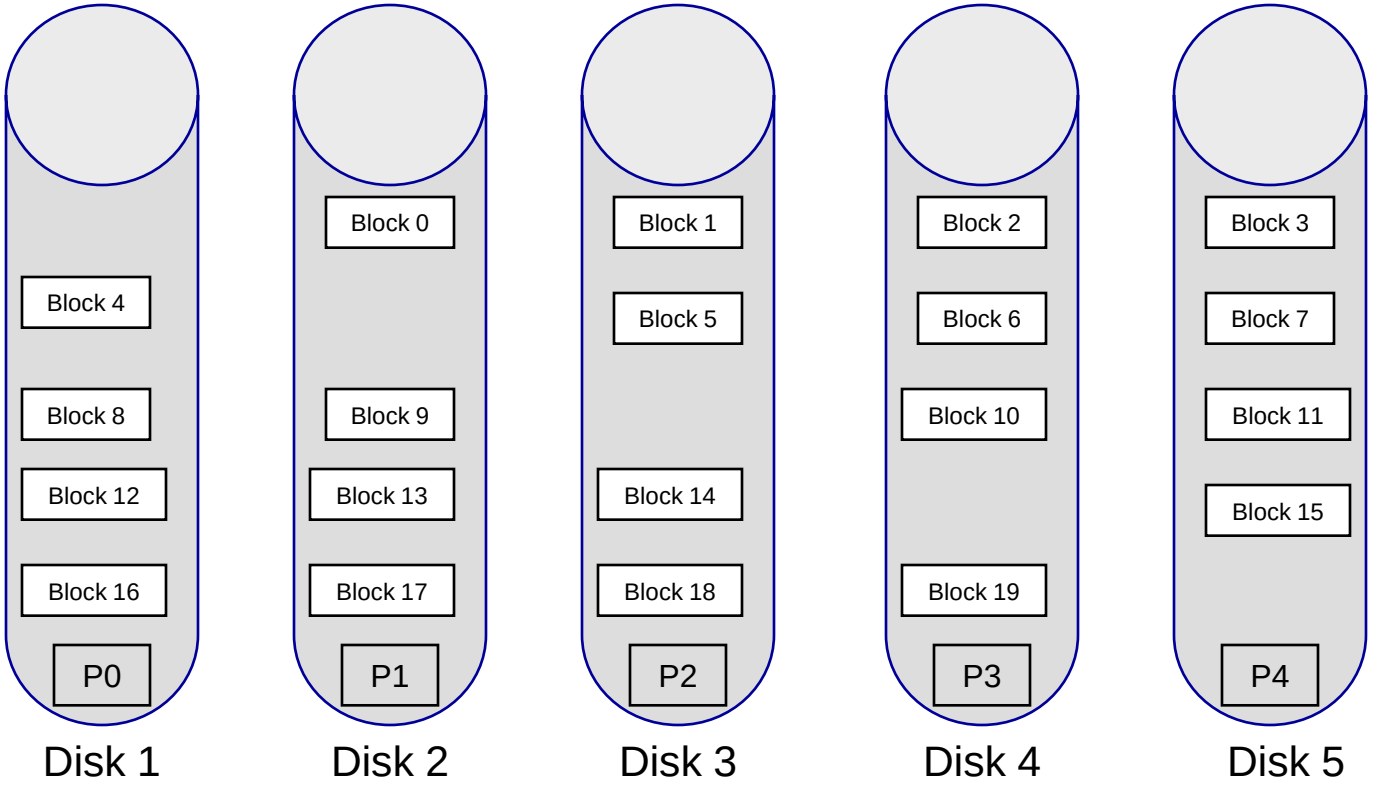
Assume an array of 5 disks.
The parity block, labeled Pk, for logical blocks 4k, 4k+1, 4k+2, and 4k+3 is stored in disk (k mod 5)+1
The corresponding blocks of the other four disks store the 4 data blocks 4k to 4k+3.
The table on the left illustrates how the first 20 blocks, numbered 0 to 19, and their parity blocks are laid out in the disk array.

Example: parity block P0, for logical blocks 0, 1, 2, and 3, is stored in disk (0 mod 5)+1 = 1
parity block P1, for logical blocks 4, 5, 6, and 7, is stored in disk (1 mod 5)+1 = 2
parity block P2, for logical blocks 8, 9, 10, and 11, is stored in disk (2 mod 5)+1 = 3
parity block P3, for logical blocks 12, 13, 14, and 15, is stored in disk (3 mod 5)+1 = 4
parity block P4, for logical blocks 16, 17, 18, and 19, is stored in disk (4 mod 5)+1 = 5






# RAID Level 5 Example (cont.)



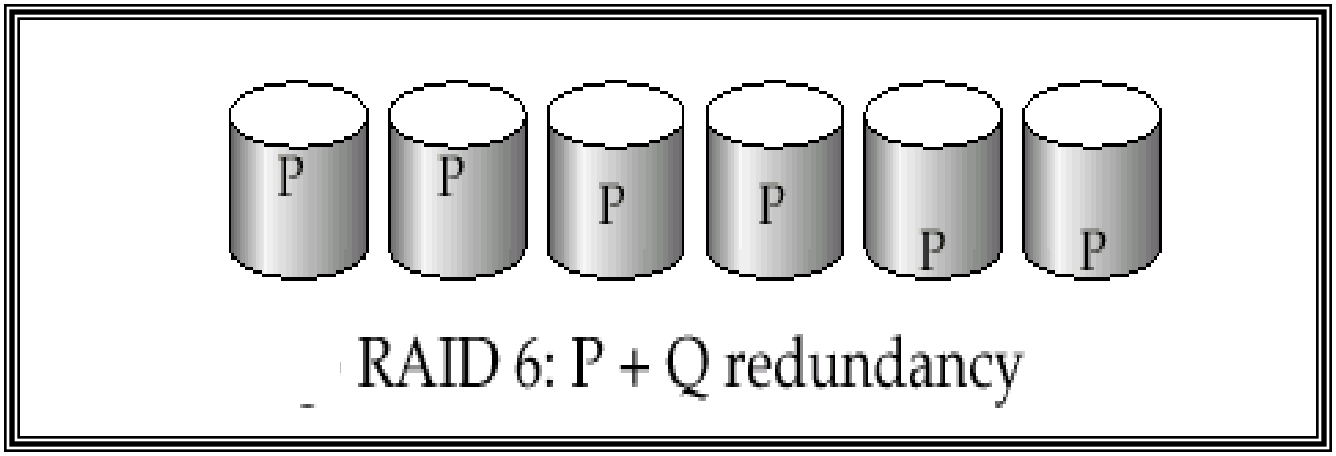



# RAID Levels (cont.)

- **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.



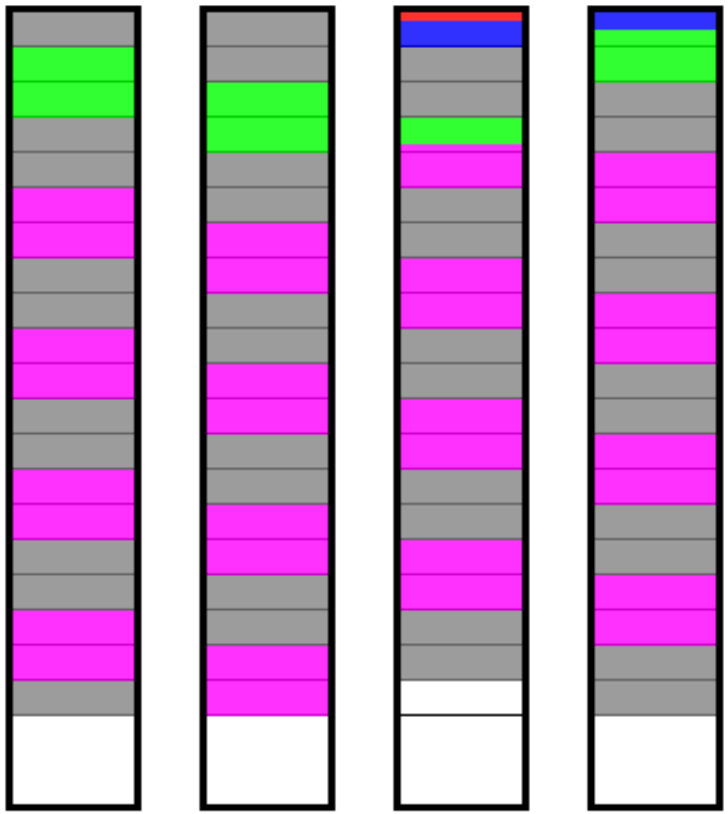
# RAID Levels (cont.)




RAID 6: P + Q redundancy



# RAID Levels (cont.)



This illustration shows how files of different sizes are distributed between the drives on a four-disk RAID 6 array using a 16 kB stripe size. As with the RAID 0 illustration, the red file is 4 kB in size; the blue is 20 kB; the green is 100 kB; and the magenta is 500 kB, with each vertical pixel representing 1 kB of space. This diagram is the same as the RAID 5 one, except that you'll notice that there is now twice as much gray parity information, and as a result, more space taken up on the four drives to contain the same data than the other levels that use striping.



# 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 almost all applications
- So competition is between 1 and 5 only.



# 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



# Additional RAID Levels

- The single RAID levels have distinct advantages and disadvantages, which is why most of them are used in various parts of the market to address different application requirements.
- It wasn't long after RAID began to be implemented that engineers looked at these RAID levels and began to wonder if it might be possible to get some of the advantages of more than one RAID level by designing arrays that use a combination of techniques.
- These RAID levels are called variously *multiple*, *nested*, or *multi*-RAID levels. They are also sometimes called *two-dimensional*, in reference to the two-dimensional schematics that are used to represent the application of two RAID levels to a set of disks.
- Multiple RAID levels are most commonly used to improve performance, and they do this well. Nested RAID levels typically provide better performance characteristics than either of the single RAID levels that comprise them. The most commonly combined level is RAID 0, which is often mixed with redundant RAID levels such as 1, 3 or 5 to provide fault tolerance while exploiting the performance advantages of RAID 0. There is never a "free lunch", and so with multiple RAID levels what you pay is a cost in complexity: many drives are required, management and maintenance are more involved, and for some implementations a high-end RAID controller is required.
- Not all combinations of RAID levels exist. Typically, the most popular multiple RAID levels are those that combine single RAID levels that complement each other with different strengths and weaknesses. Making a multiple RAID array marrying RAID 4 to RAID 5 wouldn't be the best idea, since they are so similar to begin with.



# Stable-Storage Implementation

- Write-ahead log scheme requires stable storage.
- To implement stable storage:
  - Replicate information on more than one nonvolatile storage media with independent failure modes.
  - Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.



# Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage.
- Generally, tertiary storage is built using *removable media*
- Common examples of removable media are floppy disks and CD-ROMs; other types are available.



# Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.
  - Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.
  - Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.



# Removable Disks

- A magneto-optic disk records data on a rigid platter coated with magnetic material.
  - Laser heat is used to amplify a large, weak magnetic field to record a bit.
  - Laser light is also used to read data (Kerr effect).
  - The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.
- Optical disks do not use magnetism; they employ special materials that are altered by laser light.



# WORM Disks

- The data on read-write disks can be modified over and over.
- WORM ("Write Once, Read Many Times") disks can be written only once.
- Thin aluminum film sandwiched between two glass or plastic platters.
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.
- Very durable and reliable.
- *Read Only* disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded.



# Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
  - stacker – library that holds a few tapes
  - silo – library that holds thousands of tapes
- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.

# Operating System Issues

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications

- For hard disks, the OS provides two types of abstractions:
  - Raw device – an array of data blocks.
  - File system – the OS queues and schedules the interleaved requests from several applications.



# Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device.
- Usually the tape drive is reserved for the exclusive use of that application.
- Since the OS does not provide file system services, the application must decide how to use the array of blocks.
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.



# Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- **locate** positions the tape to a specific logical block, not an entire track (corresponds to **seek**).
- The **read position** operation returns the logical block number where the tape head is.
- The **space** operation enables relative motion.
- Tape drives are "append-only" devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written.



# File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.



# Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system.
  - Small and frequently used files remain on disk.
  - Large, old, inactive files are archived to the jukebox.
- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.



# Speed

- Two aspects of speed in tertiary storage are bandwidth and latency.

- Bandwidth is measured in bytes per second.
  - Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time.
    Data rate when the data stream is actually flowing.
  - Effective bandwidth – average over the entire I/O time, including **seek** or **locate**, and cartridge switching.
    Drive's overall data rate.

# Speed

- Access latency – amount of time needed to locate data.
  - Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds.
  - Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds.
  - Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk.
- The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives.
- A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.



# Reliability

- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
- An optical cartridge is likely to be more reliable than a magnetic disk or tape.
- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

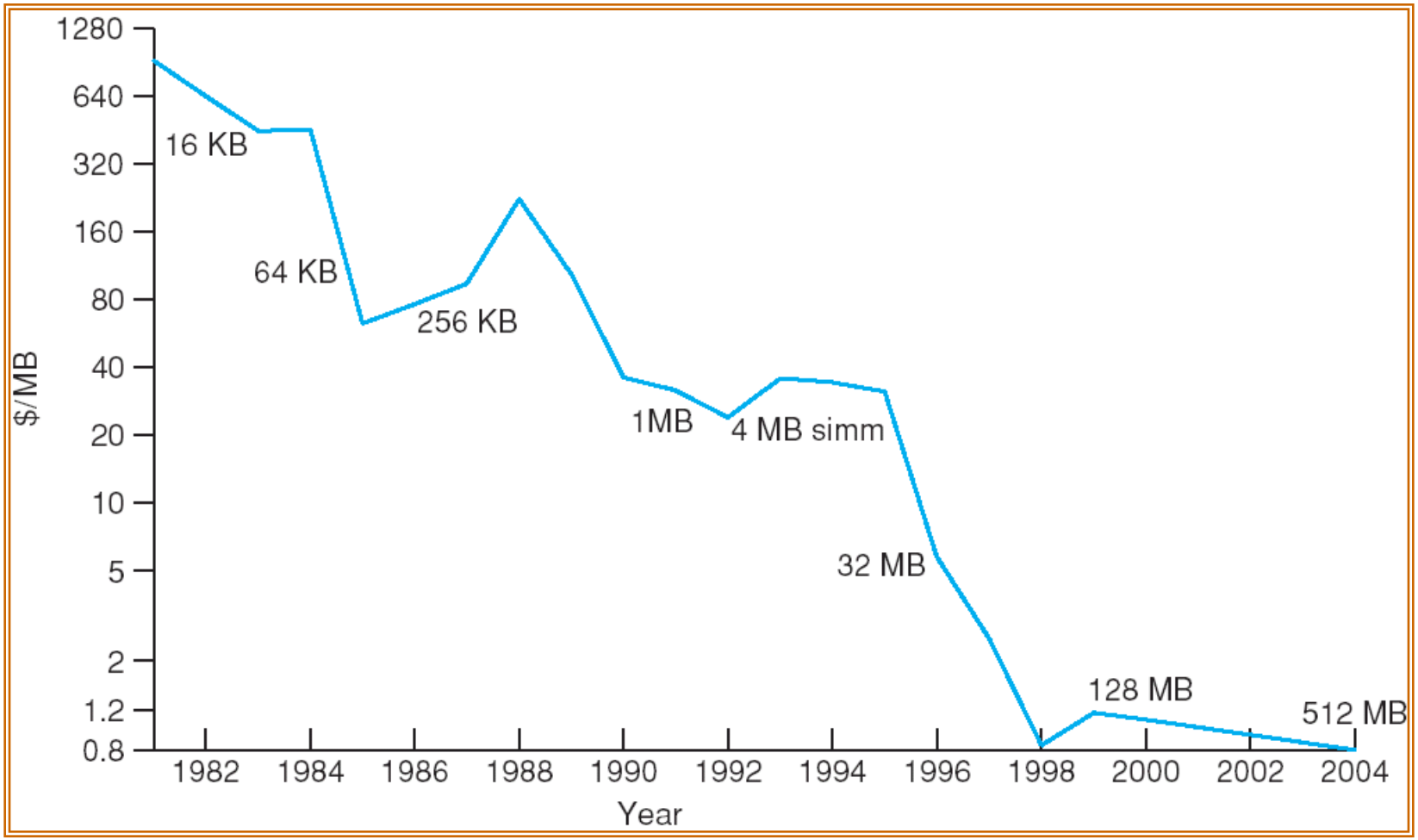


# Cost

- Main memory is much more expensive than disk storage
- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.
- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.

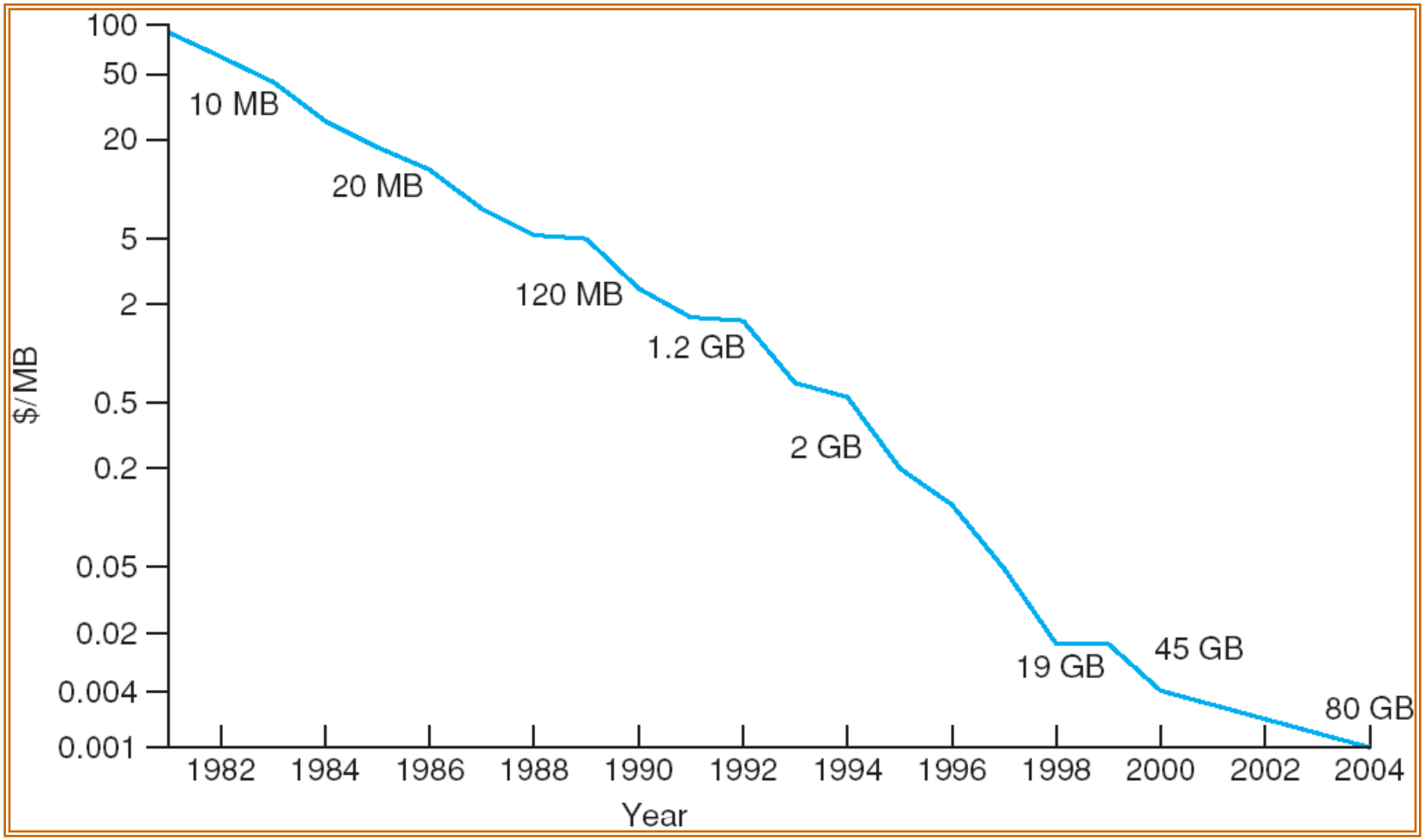


# Price per Megabyte of DRAM, From 1981 to 2004



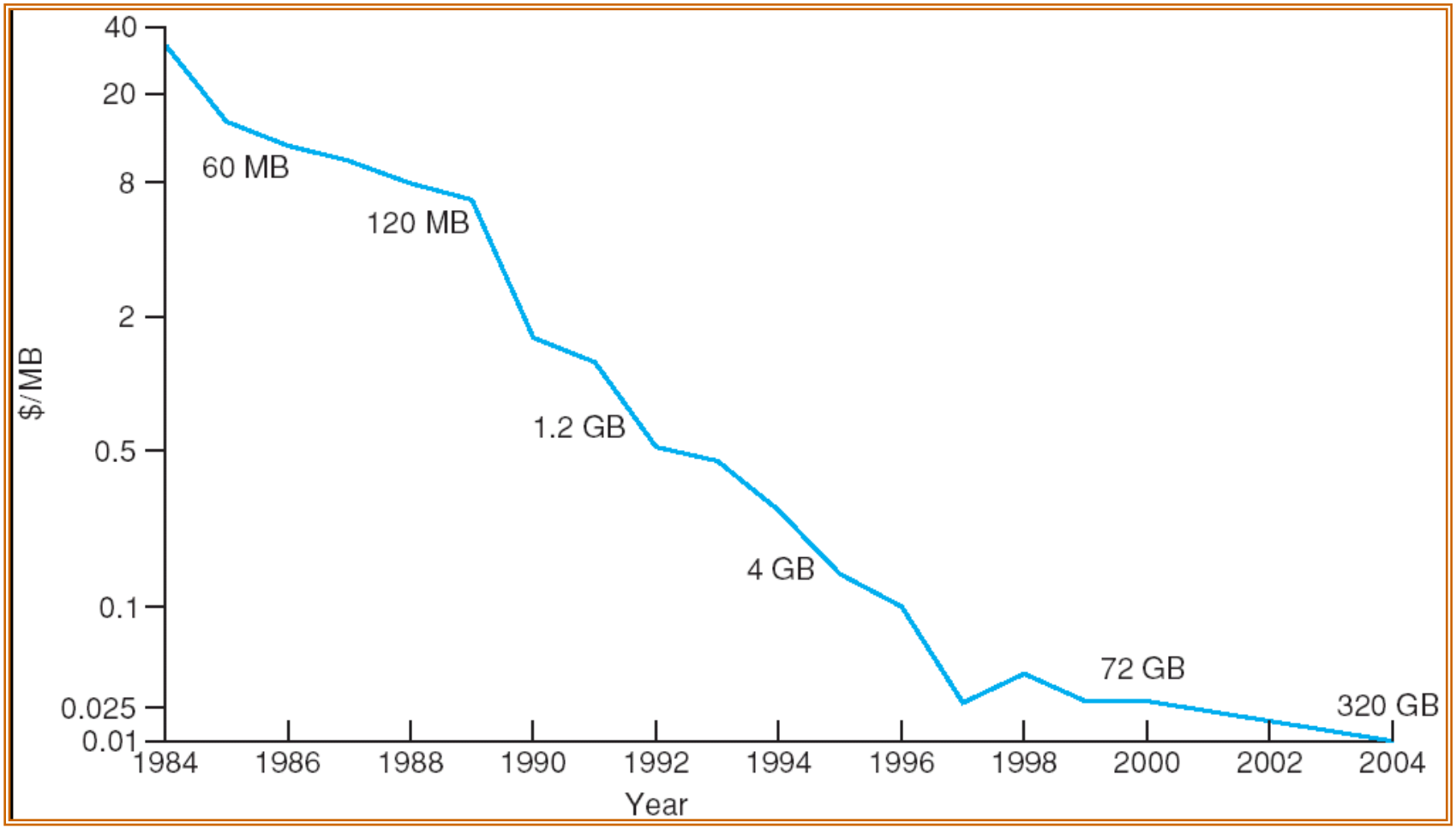



# Price per Megabyte of Magnetic Hard Disk, From 1981 to 2004






# Price per Megabyte of a Tape Drive, From 1984-2000






# File Organizations

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations
    - This case is easiest to implement; will consider variable length records later.



# Fixed-Length Records

- Simple approach:
  - Store record $i$ starting from byte $n * (i - 1)$, where $n$ is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record $i$: alternatives:
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record $n$ to $i$
  - do not move records, but link all free records on a *free list*

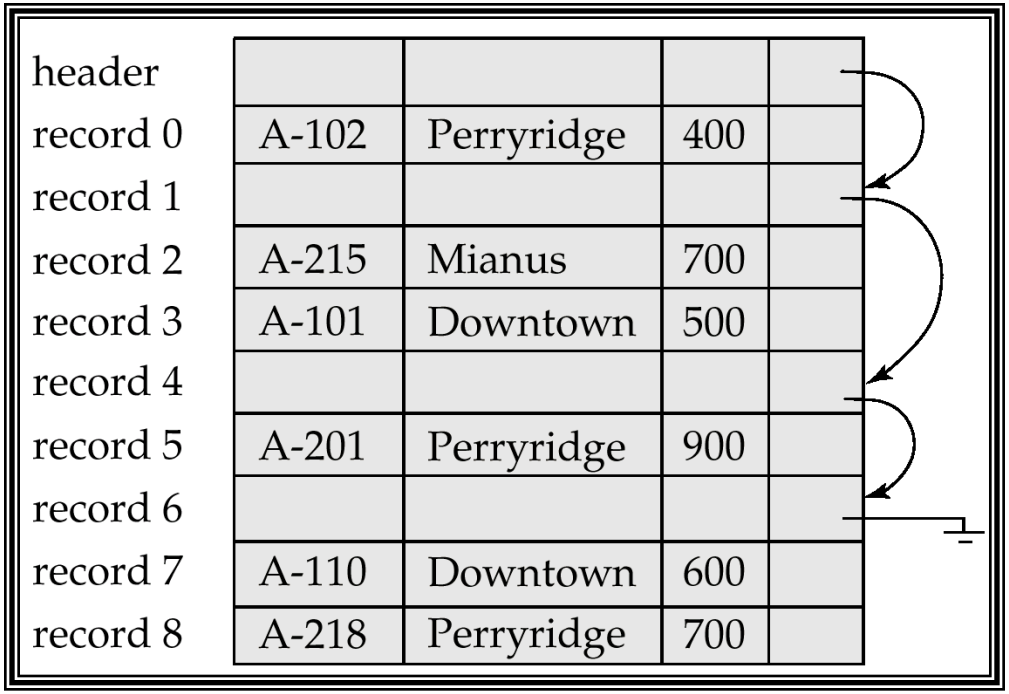
| | | | |
|---|---|---|---|
| record 0 | A-102 | Perryridge | 400 |
| record 1 | A-305 | Round Hill | 350 |
| record 2 | A-215 | Mianus | 700 |
| record 3 | A-101 | Downtown | 500 |
| record 4 | A-222 | Redwood | 700 |
| record 5 | A-201 | Perryridge | 900 |
| record 6 | A-217 | Brighton | 750 |
| record 7 | A-110 | Downtown | 600 |
| record 8 | A-218 | Perryridge | 700 |



# Free-Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.

- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)






# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).
- Byte string representation
  - Attach an *end-of-record* (⊥) control character to the end of each record
  - Difficulty with deletion
  - Difficulty with growth

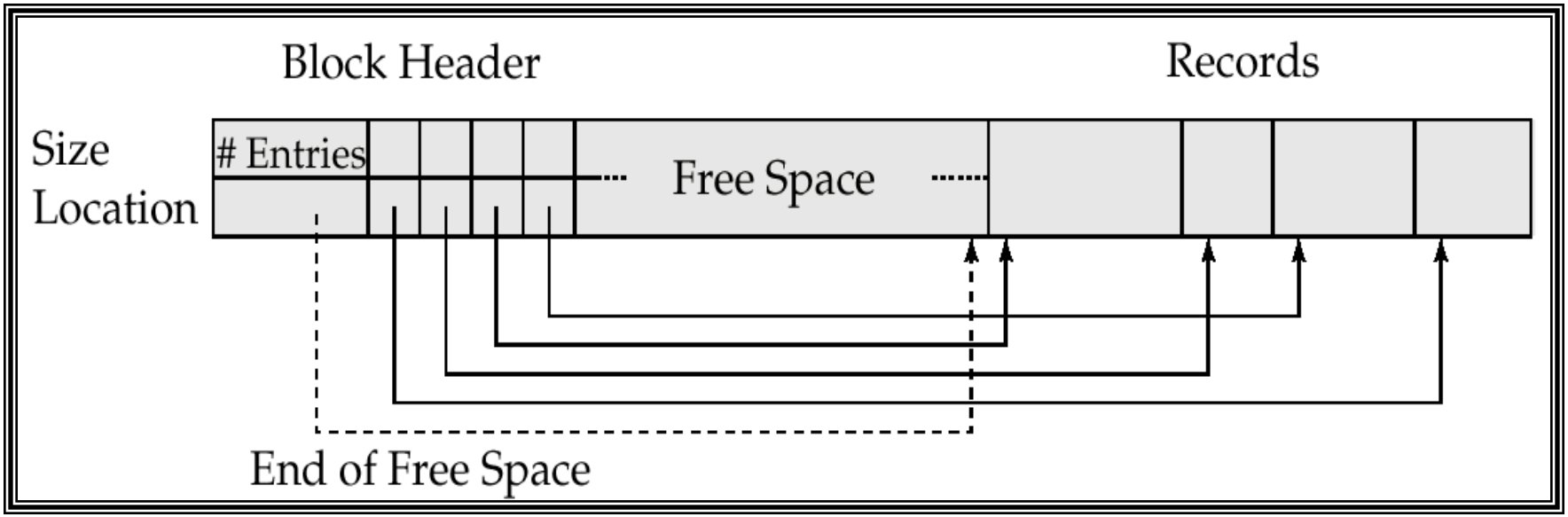


# Variable-Length Records – Slotted Page

- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.



# Variable-Length Records – Slotted Page (cont.)






# Variable-Length Records (cont.)

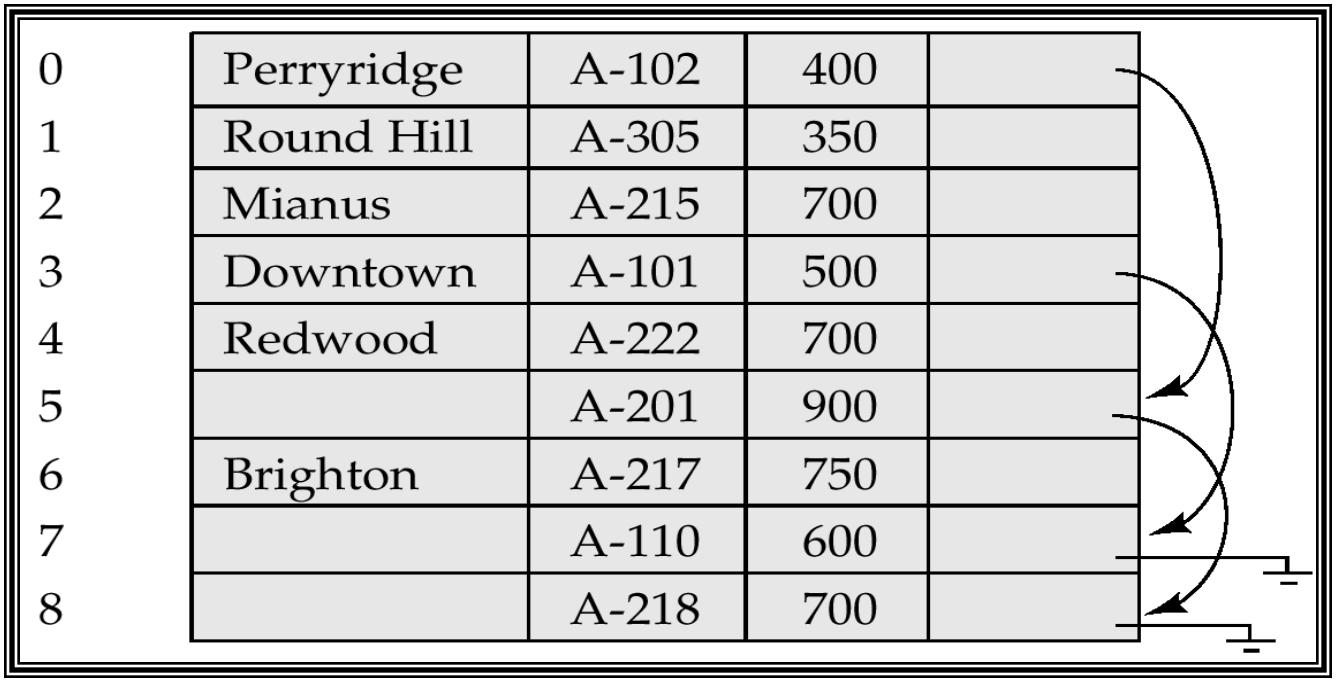
- Fixed-length representation:
  - reserved space
  - pointers
- Reserved space – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Perryridge | A-102 | 400 | A-201 | 900 | A-218 | 700 |
| 1 | Round Hill | A-305 | 350 | ⊥ | ⊥ | ⊥ | ⊥ |
| 2 | Mianus | A-215 | 700 | ⊥ | ⊥ | ⊥ | ⊥ |
| 3 | Downtown | A-101 | 500 | A-110 | 600 | ⊥ | ⊥ |
| 4 | Redwood | A-222 | 700 | ⊥ | ⊥ | ⊥ | ⊥ |
| 5 | Brighton | A-217 | 750 | ⊥ | ⊥ | ⊥ | ⊥ |



# Variable-Length Records – Pointer Method

- Pointer method
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known

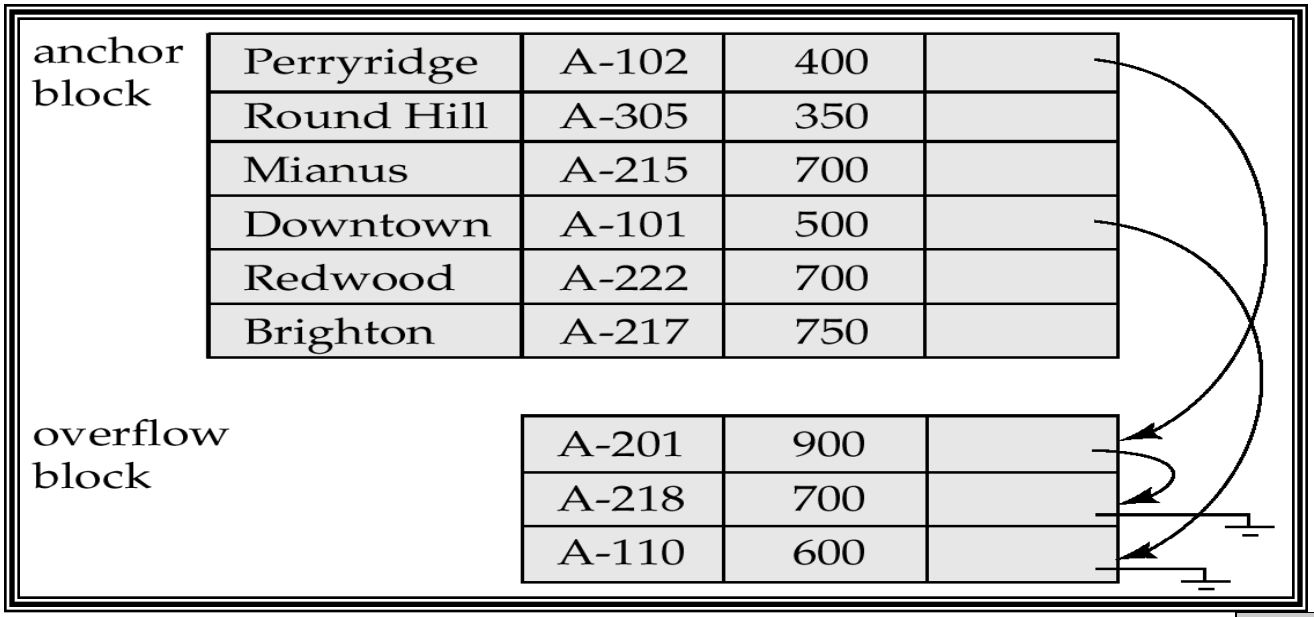


| | | | | |
|---|---|---|---|---|
| 0 | Perryridge | A-102 | 400 | |
| 1 | Round Hill | A-305 | 350 | |
| 2 | Mianus | A-215 | 700 | |
| 3 | Downtown | A-101 | 500 | |
| 4 | Redwood | A-222 | 700 | |
| 5 | | A-201 | 900 | |
| 6 | Brighton | A-217 | 750 | |
| 7 | | A-110 | 600 | |
| 8 | | A-218 | 700 | |



# Variable-Length Records – Pointer Method
(cont.)

- Disadvantage to pointer structure; space is wasted in all records except the first in a a chain.
- Solution is to allow two kinds of block in file:
  - Anchor block – contains the first records of chain
  - Overflow block – contains records other than those that are the first records of chairs.






# Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

# Sequential File Organization

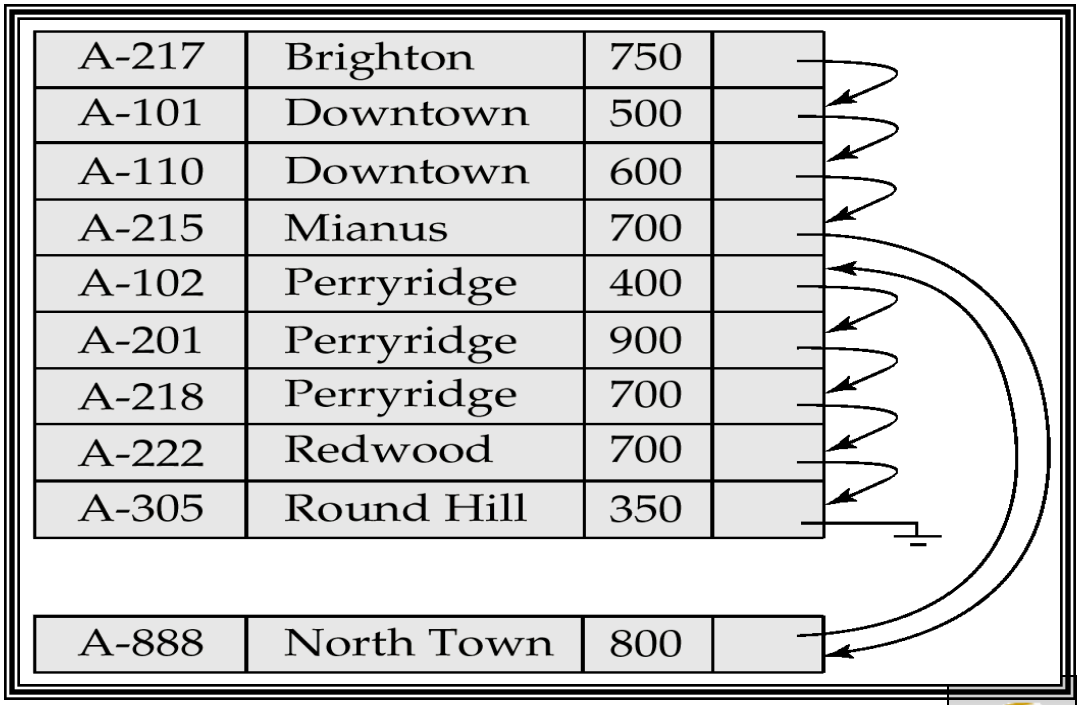
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

| | | | |
|---|---|---|---|
| A-217 | Brighton | 750 | |
| A-101 | Downtown | 500 | |
| A-110 | Downtown | 600 | |
| A-215 | Mianus | 700 | |
| A-102 | Perryridge | 400 | |
| A-201 | Perryridge | 900 | |
| A-218 | Perryridge | 700 | |
| A-222 | Redwood | 700 | |
| A-305 | Round Hill | 350 | |



# Sequential File Organization (cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order.






# Clustering File Organization

- Simple file structure stores each relation in a separate file
- Can instead store several relations in one file using a **clustering** file organization
- E.g., clustering organization of *customer* and *depositor:*

- Good for queries involving depositor customer, and for queries involving one single customer and his accounts
- Bad for queries involving only customer
- Results in variable size records

| | | |
|---|---|---|
| Hayes | Main | Brooklyn |
| Hayes | A-102 | |
| Hayes | A-220 | |
| Hayes | A-503 | |
| Turner | Putnam | Stamford |
| Turner | A-305 | |



# Basic Concepts Behind Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form


- Index files are typically much smaller than the original file.
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".



# Index Evaluation Metrics

- Access types: The types of access that are efficiently supported.
  - Finding records with a specified attribute value.
  - Finding records with an attribute value falling in a specified range of values.
- Access time: The time required to find a particular data item, or set of items.
- Insertion time: The time it takes to insert a new data item. This value includes the time required to find the correct place to insert, as well as the time required to update the index structure.
- Deletion time: The time it takes to delete a data item. This value includes the time required to find the item, as well as the time required to update the index structure.
- Space overhead: The additional space required by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.



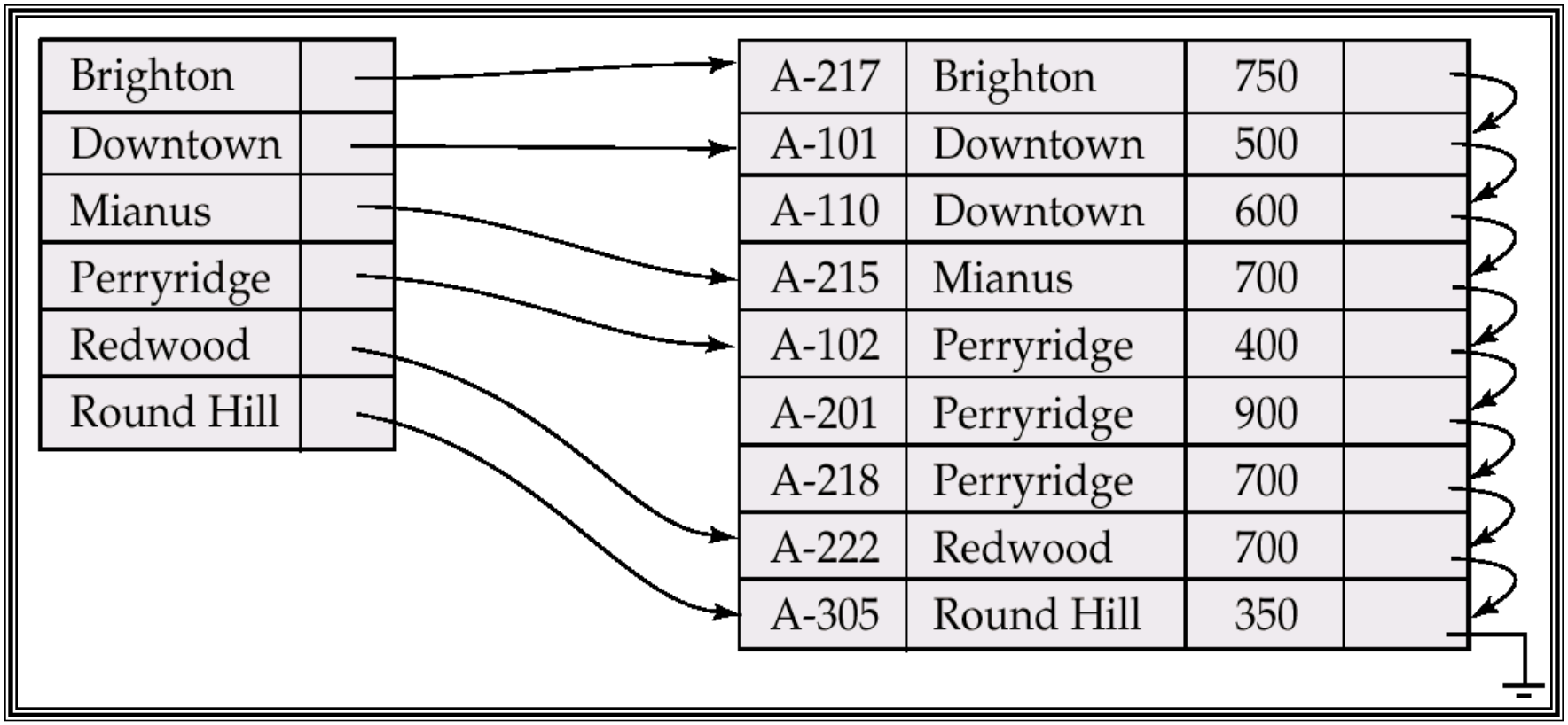
# Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called clustering index
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.
- One of the oldest index schemes used in database systems. Designed for applications that require both sequential processing of entire files as well as random access to individual records.



# Dense Index Files

- Dense index — Index record appears for every search-key value in the file.



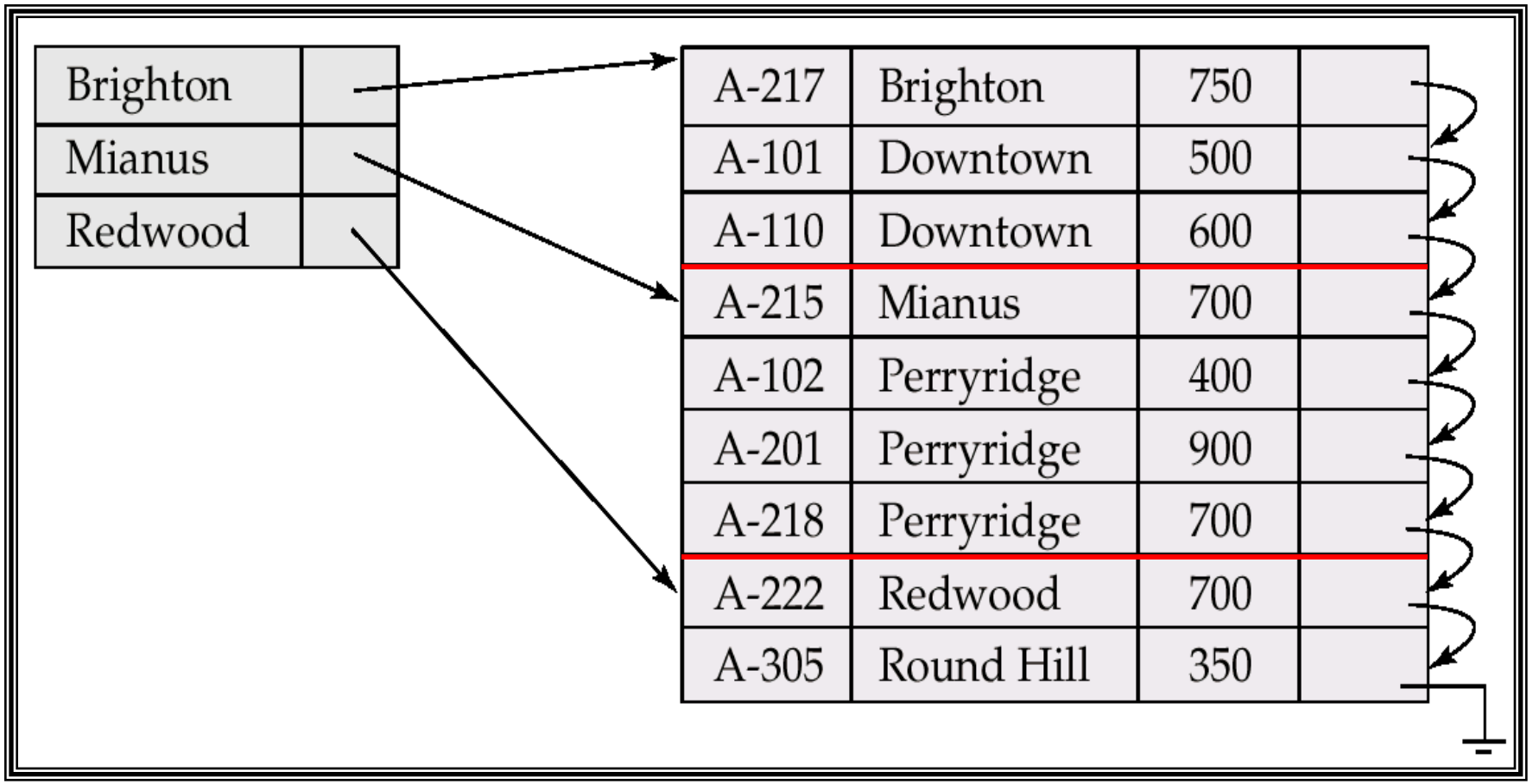



# Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



# Example of Sparse Index Files



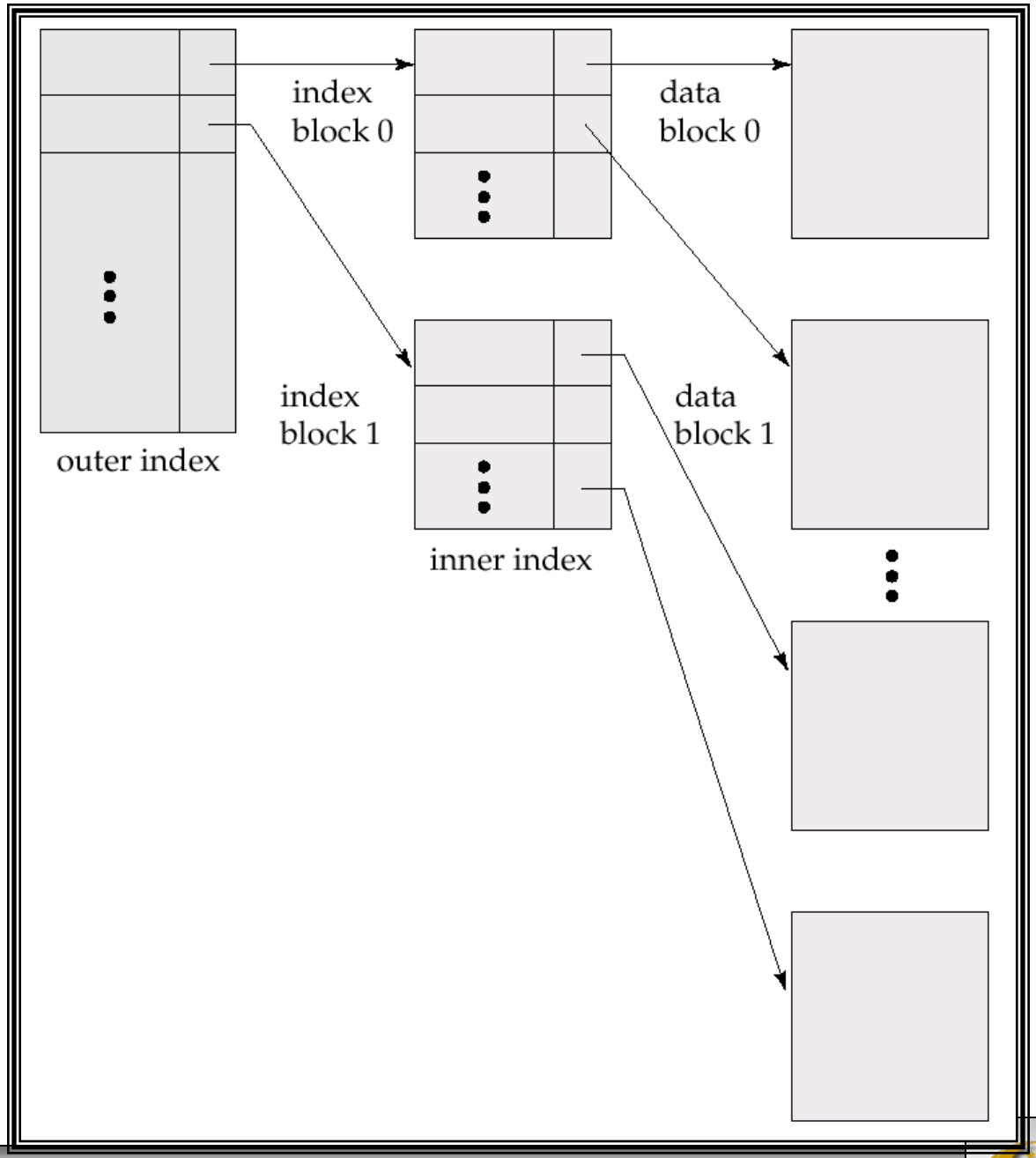



# Multi-level Indexing

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



# Example of Multi-level Indexing






# Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



# Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices – if the search-key value does not appear in the index, insert it.
  - Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

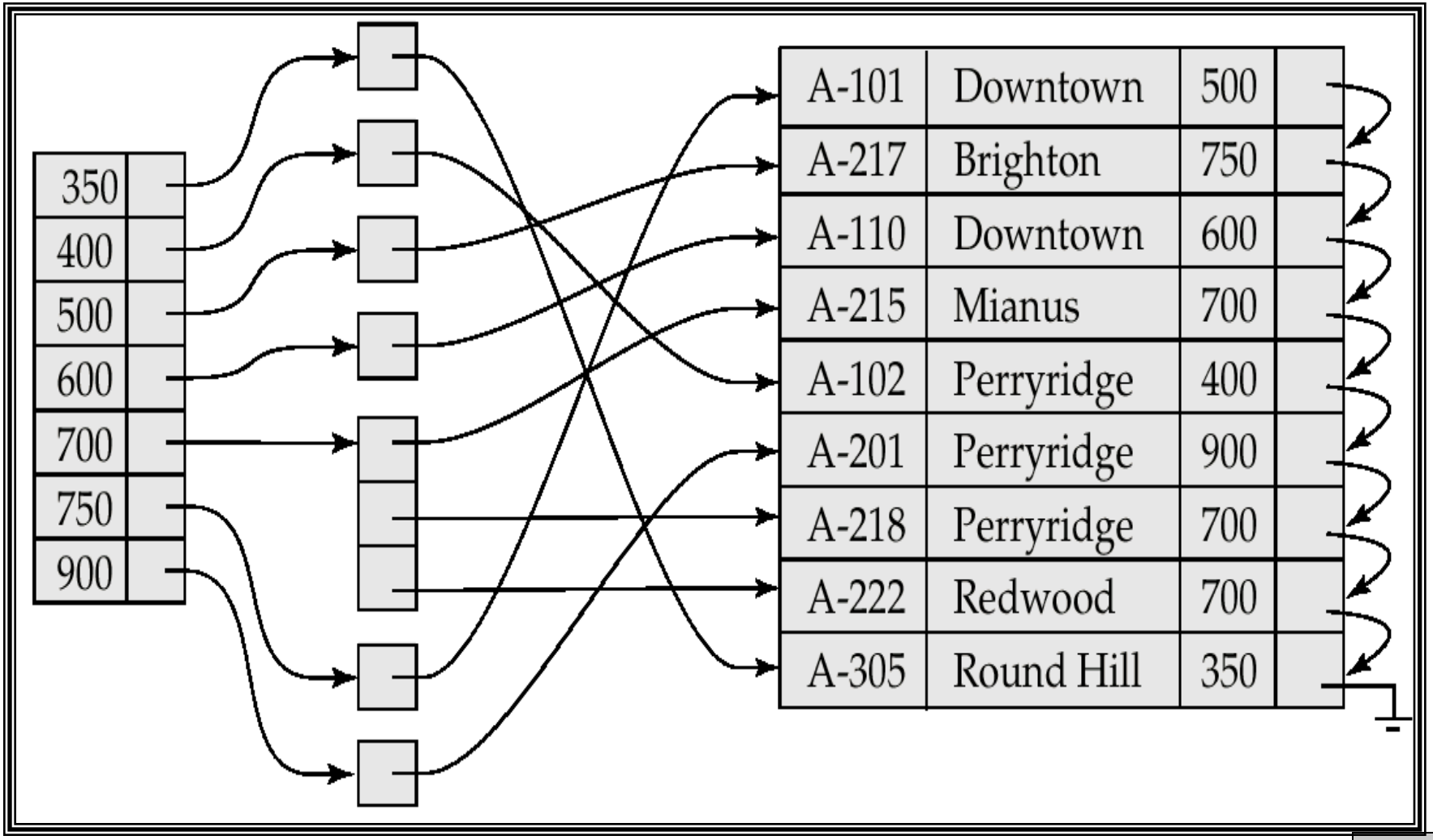


# Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) that satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch.
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances.
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.



# Secondary Index on ***balance*** field of ***account***






# Primary and Secondary Indices

- Secondary indices must be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk