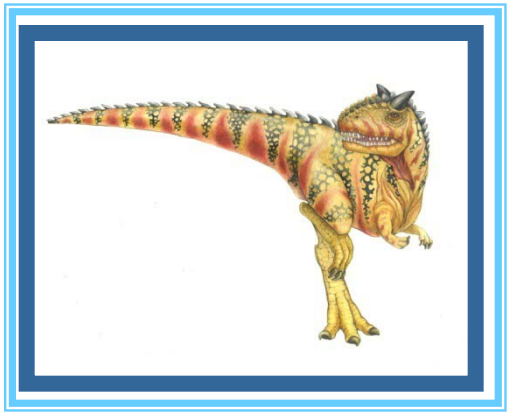
File Systems

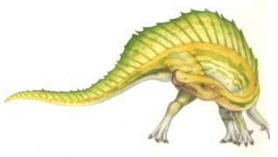


Sources:

- Kubiatowicz ©2010
- Walpole ©2010
- Silberschatz, Galvin and Gagne ©2009

Chapter 10, Sec. 10.1, Silberschatz et al., 8th ed.,

Intro

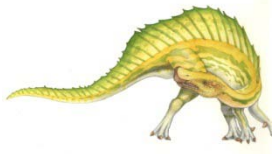


Why do we need a file system?

- Files are just a sequence of bits stored in a given device
  - Data must survive the termination of the process that created it (or even after the power down)
    - Called *persistence*
  - Can store large amounts of data
  - Users need to interchange information
  - Multiple processes must be able to access the information concurrently







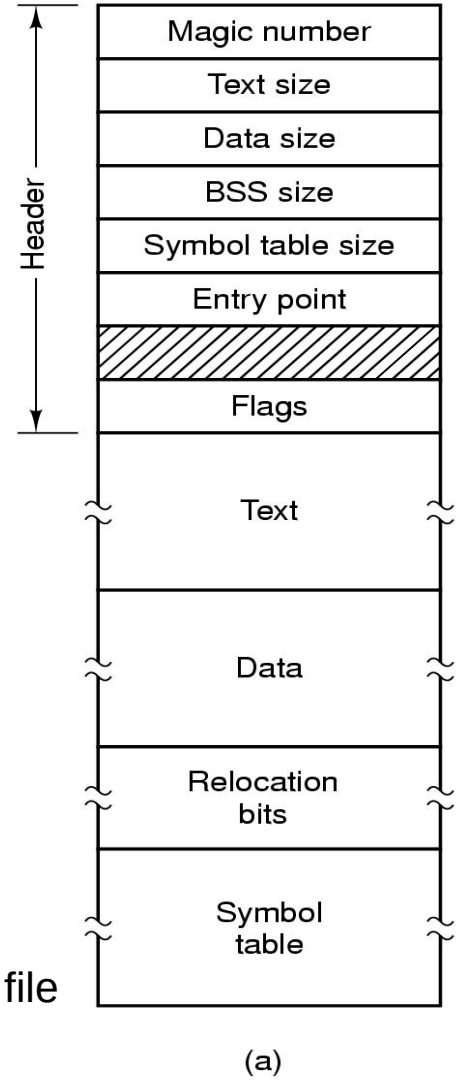
File types

- Even though files are just a sequence of bytes, programs can impose structure on them, by convention
  - Files with a certain standard structure imposed can be identified using an extension to their name
  - Application programs may look for specific file extensions to indicate the file's type
  - But as far as the operating system is concerned its just a sequence of bytes

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Which file types does the OS understand?

- Executable files
  - The OS must understand the format of executable files in order to execute programs
    - Create process (`fork`)
    - Put program and data in process address space (`exec`)



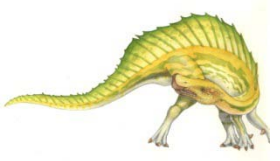
An executable unix file

File attributes

- Various meta-data needs to be associated with files

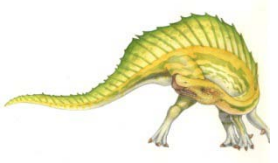
Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

- This meta-data is called the file attributes
  - Maintained in file system data structures for each file



File access

- Sequential Access
  - read all bytes/records from the beginning
  - cannot jump around (but could rewind or back up) convenient when medium was magnetic tape
- Random Access (or Direct Access)
  - can read bytes (or records) in any order
  - essential for database systems
  - option 1:
    - move position, then read
  - option 2:
    - perform read, then update current position



Some file-related system calls

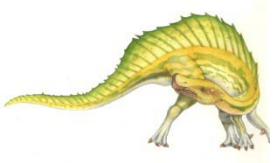
- Create a file
- Delete a file
- Open
- Close
- Read (n bytes from current position)
- Write (n bytes to current position)
- Append (n bytes to end of file)
- Seek (move to new position)
- Get attributes
- Set/modify attributes
- Rename file

- Example:

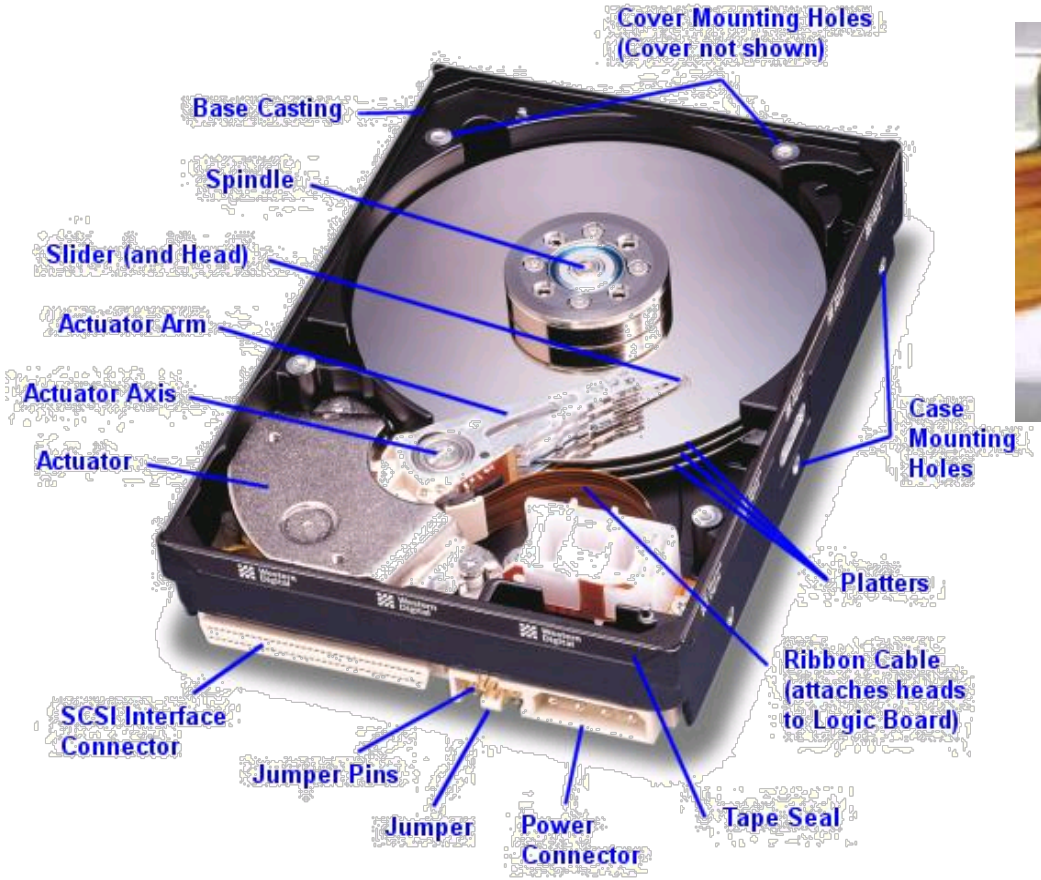
```
fd = open (name, mode)
byte_count = read (fd, buffer, buffer_size)
byte_count = write (fd, buffer, num_bytes)
close (fd)
```

Chapter 12, Sec. 12.1, Silberschatz et al., 8th ed.; Kubiatowicz, lecture 17

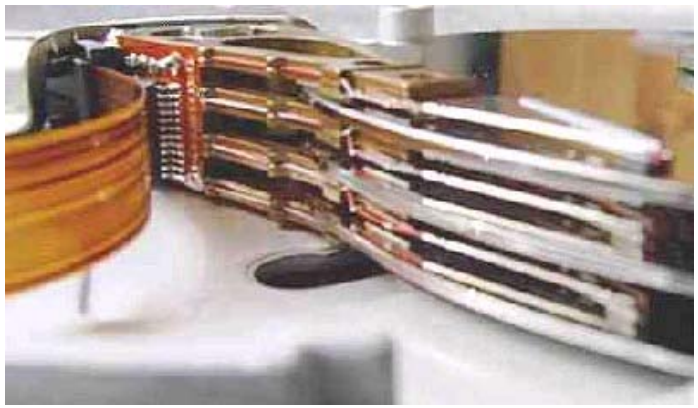
Overview of hard disks



Hard Disk Drives



Western Digital Drive
http://www.storagereview.com/guide/

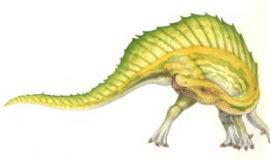


Read/Write Head
Side View

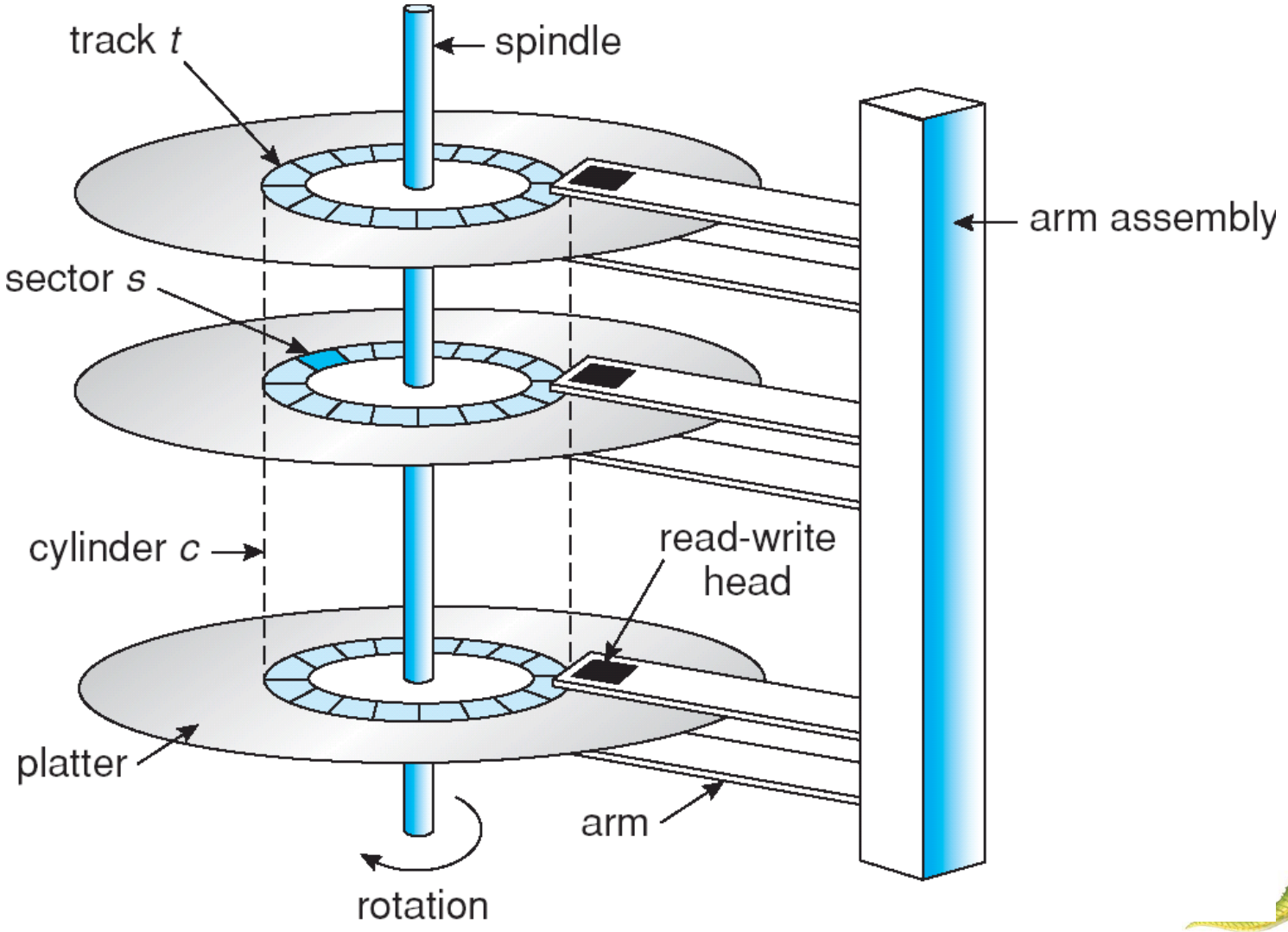
IBM/Hitachi Microdrive

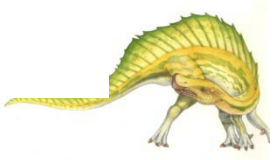
Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 250 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



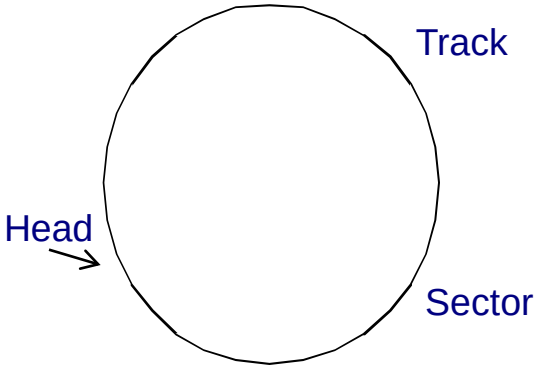
Moving-head Disk Mechanism

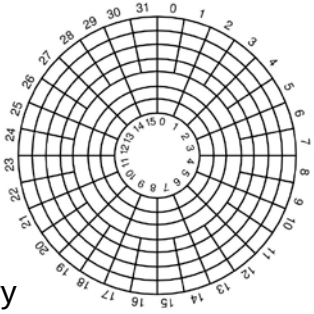




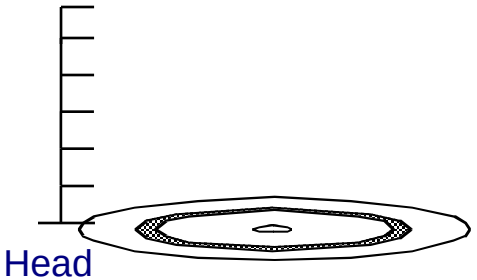
Disk Geometry

- Properties
  - Head moves in to address circular **track** of information
  - Independently addressable element: **sector**
    - OS always transfers groups of sectors together—**blocks** or **clusters**
  - Items addressable without moving head: **cylinder**
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location





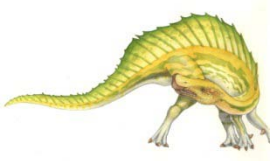
physical geometry



Disk formatting

- A new magnetic disk is just a blank platter of magnetic recorder material
- Formatting a disk: Divide a disk into sectors that the disk controller can read and write
- A disk sector

- The header and trailer have meta-data used by the disk controller, such as a sector number and error-correcting code (ECC).
- Typically
  - 512 bytes data area of the sector
  - ECC = 16 bytes

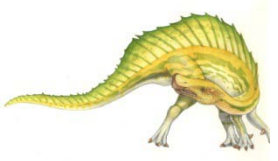


Disk Structure

- Disk can be subdivided into **partitions**
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- If formatted, each partition can hold its own file system
  - Unix file system (UFS)
  - Linux file systems (extended file system (ext2, ext3, ext4) and others)
  - Microsoft file systems: FAT (FAT12, FAT16, vFAT, FAT32), NTFS (Windows NT File system)
- Disks or partitions can be **RAID** protected against failure (RAID = Redundant Array of Independent Disks)
- Entity containing a file system (disk or partition) is known as a **volume**
- As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer

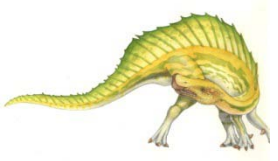
Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks** (also known as **clusters**)
- The logical block is the smallest unit of transfer (a file, at least, requires one block)
- 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.



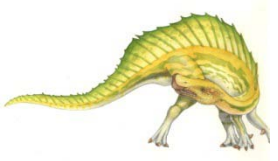
Typical Numbers of a Magnetic Disk

- Specs for modern drives
  - Space: 2TB in 3½ form factor (several manufacturers)
  - Area Density: up around 340 GB/square Inch for 2TB
- Average seek time as reported by the industry:
  - Typically in the range of 5 ms to 12 ms
  - Locality of reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - *Most* disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 512B – 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: 2 to 50 MB per second
- Controller time depends on controller hardware



Kubiatowicz, lecture 17

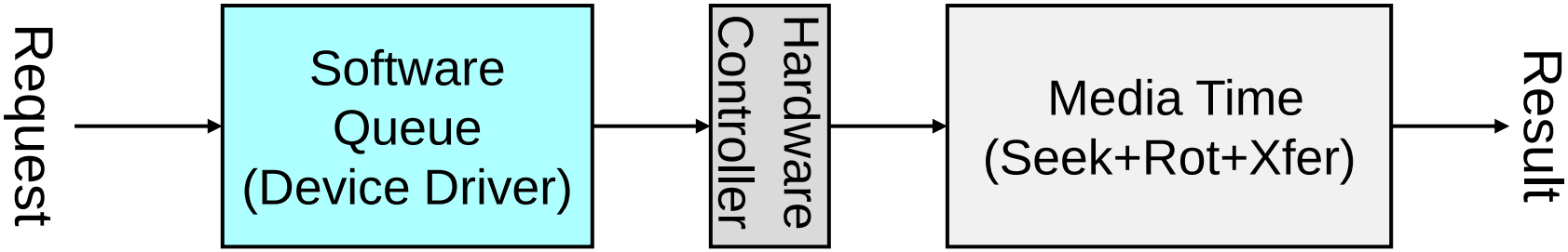
Hard disk performance



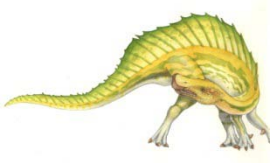
Performance Model

- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head

- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time

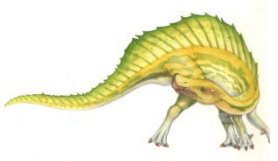


- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track



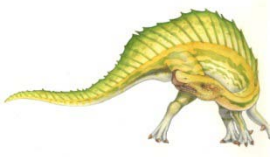
Disk Performance

- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms, avg rotational delay of 4ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 10ms to fetch/put data: 100 KByte/sec
- Random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 5ms to fetch/put data: 200 KByte/sec
- Next sector on same track:
  - Transfer (0.25ms): 4 MByte/sec
- Key to using disk effectively (esp. for file systems) is to minimize seek and rotational delays



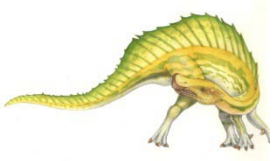
Disk Tradeoffs

- How do manufacturers choose disk sector sizes?
  - Need 100-1000 bits between each sector to allow system to measure how fast disk is spinning and to tolerate small (thermal) changes in track length
- What if sector was 1 byte?
  - Space efficiency – only 1% of disk has useful space
  - Time efficiency – each seek takes 10 ms, transfer rate of 50 – 100 Bytes/sec
- What if sector was 1 KByte?
  - Space efficiency – only 90% of disk has useful space
  - Time efficiency – transfer rate of 100 KByte/sec
- What if sector was 1 MByte?
  - Space efficiency – almost all of disk has useful space
  - Time efficiency – transfer rate of 4 MByte/sec



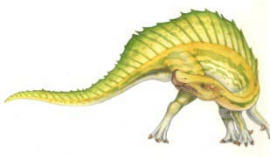
Secs. 11.1 – 11.4, Sec. 12.5, Silberschatz et al., 8th ed.,

File system implementation

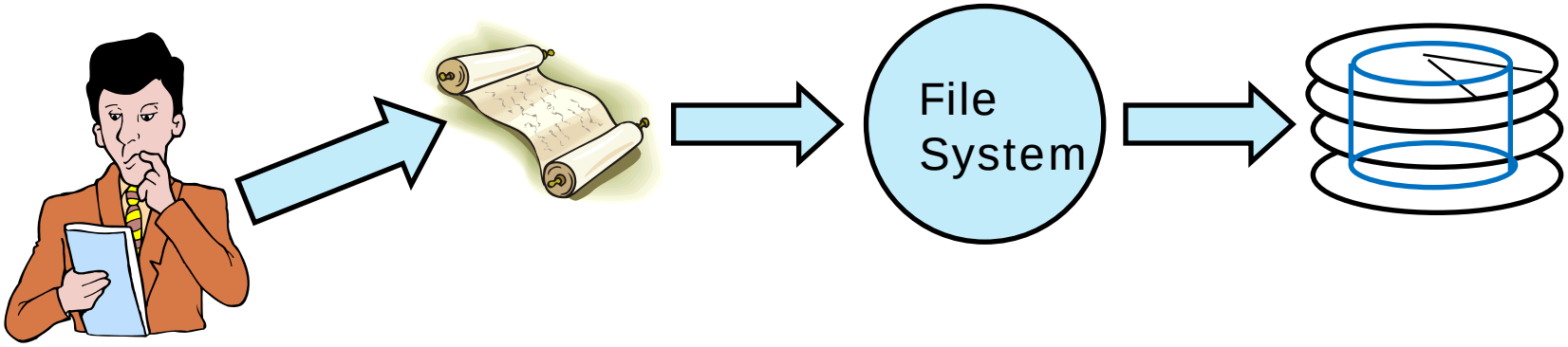


Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc.
- User vs. System View of a File
  - User's view:
    - Durable Data Structures
  - System's view (system call interface):
    - Collection of Bytes (UNIX)
    - Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - Block size ≥ sector size; in UNIX, block size is 4KB

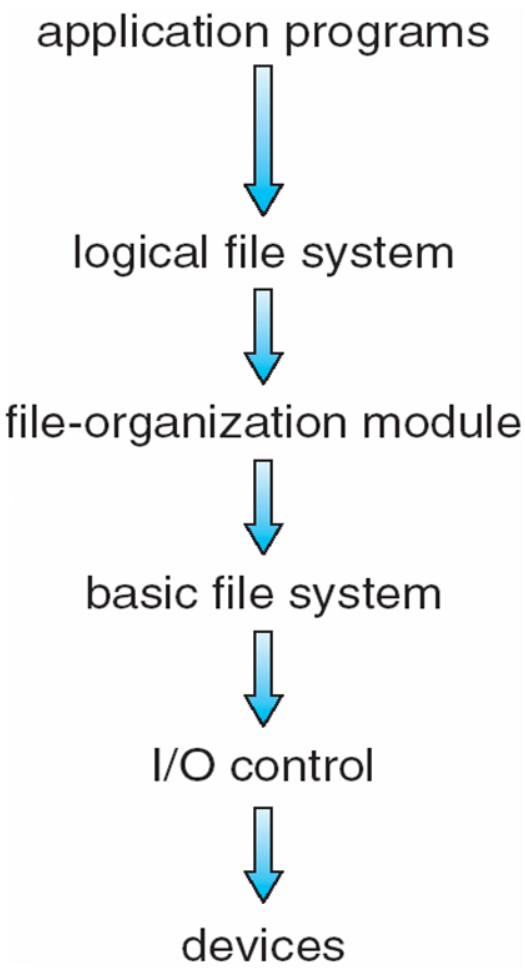


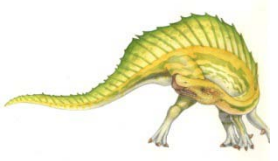
Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` $\Rightarrow$ buffers something like 4096 bytes, even if interface is one byte at a time
- **From now on, file is a collection of blocks**

Layered File System





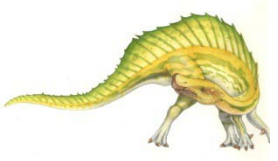
File system implementation

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files

- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - First case: OS/BIOS must deal with bad sectors
    - Second case: hardware shields OS from structure of disk

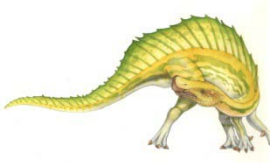
- Need way to structure files: File header or file-control block (FCB)
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk



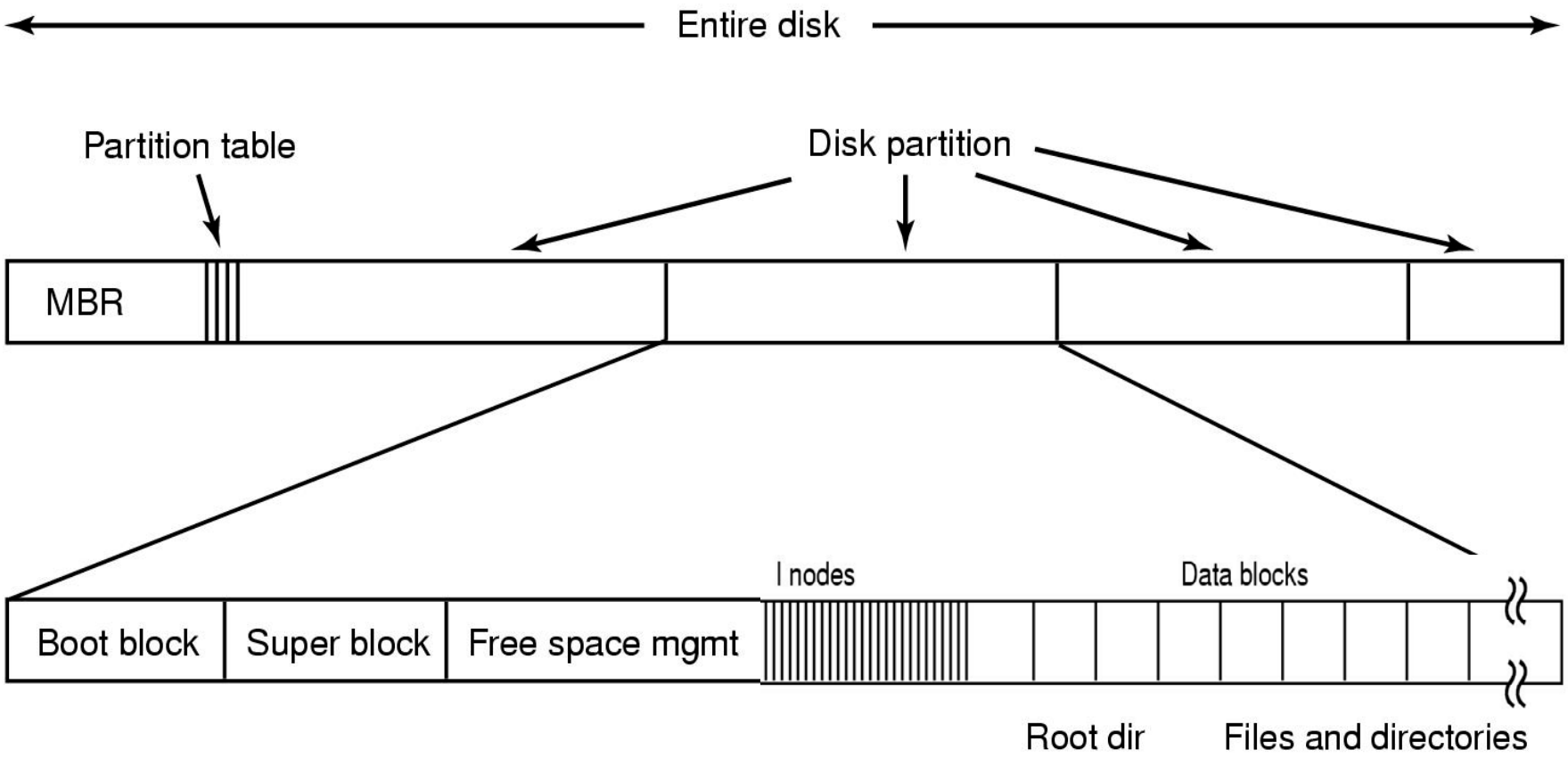
On-disk structures

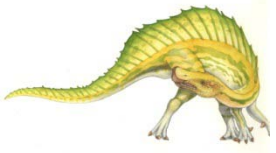
- Boot control block (per volume) containing information needed by the system to boot an operating system from than volume. It can be empty if the disk hasn't an OS.
  - In UFS (Unix File System): the Boot Block
  - In NTFS: the Partition Boot Sector
- Volume control block (per volume) with the volume metadata, such as the number of blocks in the partition, size of blocks, free-block info, FCB pointers, …
  - In UFS: the superblock
  - In NTFS: the Master File Table (MFT)
- Directory structure (per file system) used to organized data
- File control block (FCB) (per file) with details about the file. Include a unique identifier to allow association with a directory entry.
  - In UFS: the i-node
  - In NTFS: the Master File Table
  - In FAT: the FAT itself



The UNIX file system

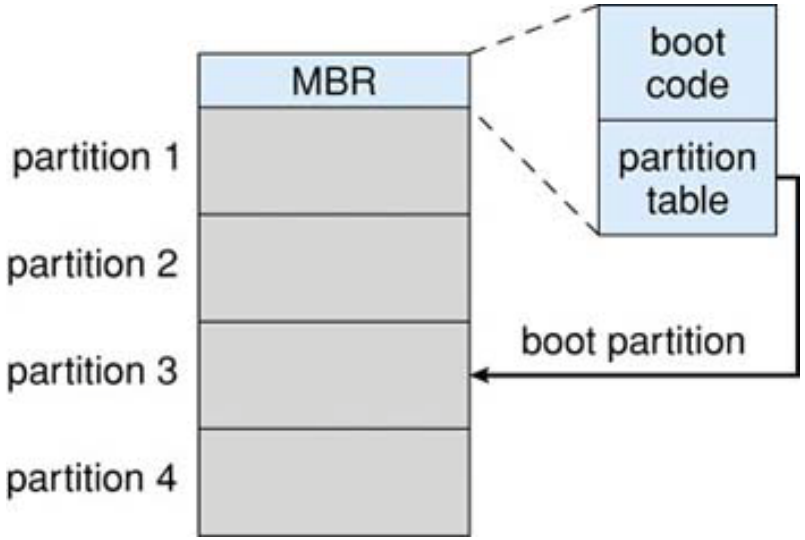
- The layout of the disk (for early Unix systems):



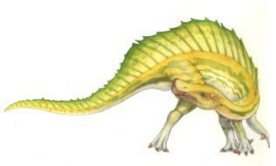


On-disk structures

- Example:
  - Sector 0: “Master Boot Record” (MBR)
    - Contains the partition map
  - Rest of disk divided into “partitions”
    - Partition: sequence of consecutive sectors.

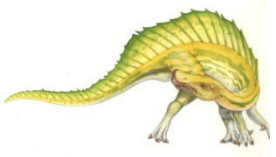


  - Every partition may start with a “boot block”
    - Contains a small program
    - This “boot program” reads in an OS from the file system in that partition
  - OS Startup
    - Bios reads MBR , then reads & execs a boot block



In-memory structures

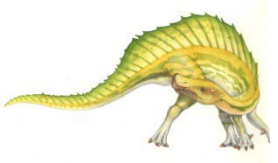
- In-memory information used for both file-system management and performance improvement via caching
- Data are loaded at mount time, updated during file-system operations and discarded at dismount
- Several type of structures may include:
  - Mount table: with information about each mounted volume
  - Directory structure cache: holding the directory information of recently accessed directories
  - System-wide open-file table: contains a copy of the FCB of each open file (as well as other information)
  - Per-process open-file table: contains a pointer to the appropriate entry in the system-wide open-file table
  - Buffers holding file-system blocks when they are being read from/write to disk



Allocation Methods

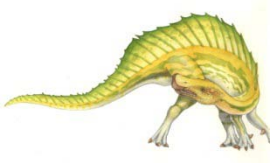
- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (deletion, growth, truncation, etc)

- An allocation method refers to how disk blocks are allocated for files:
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

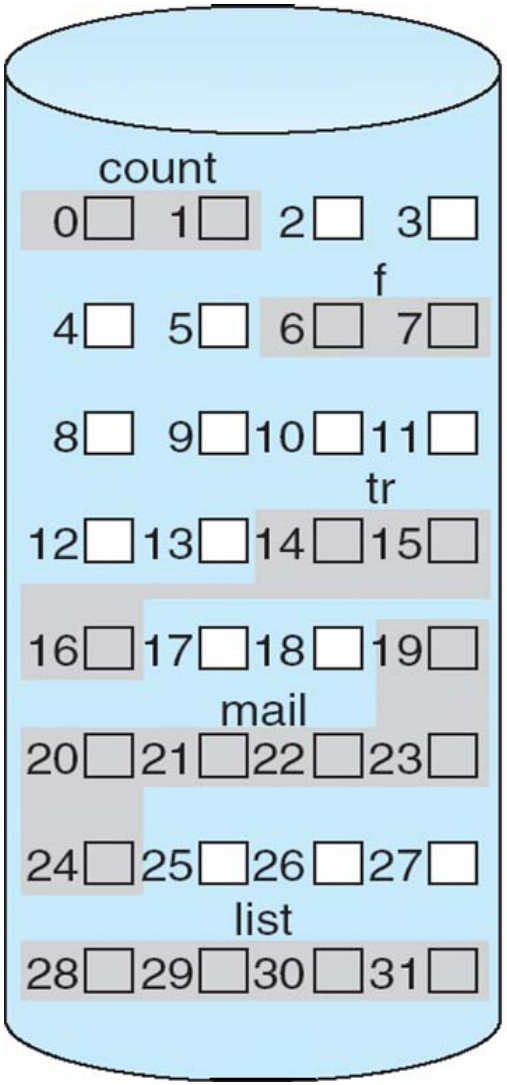


Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
  - Use continuous range of blocks in logical block space (analogous to *base + bounds* in virtual memory)
  - User says in advance how big file will be (disadvantage)
- Search bit-map for space using best fit/first fit (What if not enough contiguous space for new file?)
- Pros:
  - Fast Sequential access, Easy Random access
  - Simple: File header contains only the starting location (first block/LBA) and length (# of blocks) are required
- Cons:
  - External Fragmentation / Hard to grow files
  - Free holes get smaller and smaller
  - Could compact space, but that would be *really* expensive

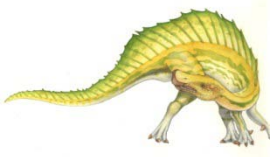


Contiguous Allocation



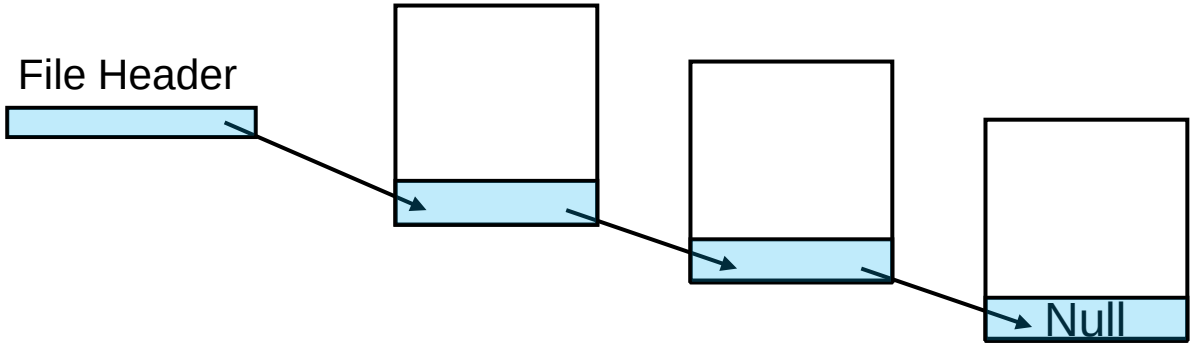
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

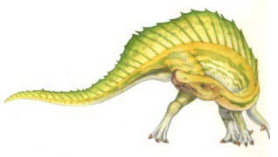


Linked Allocation

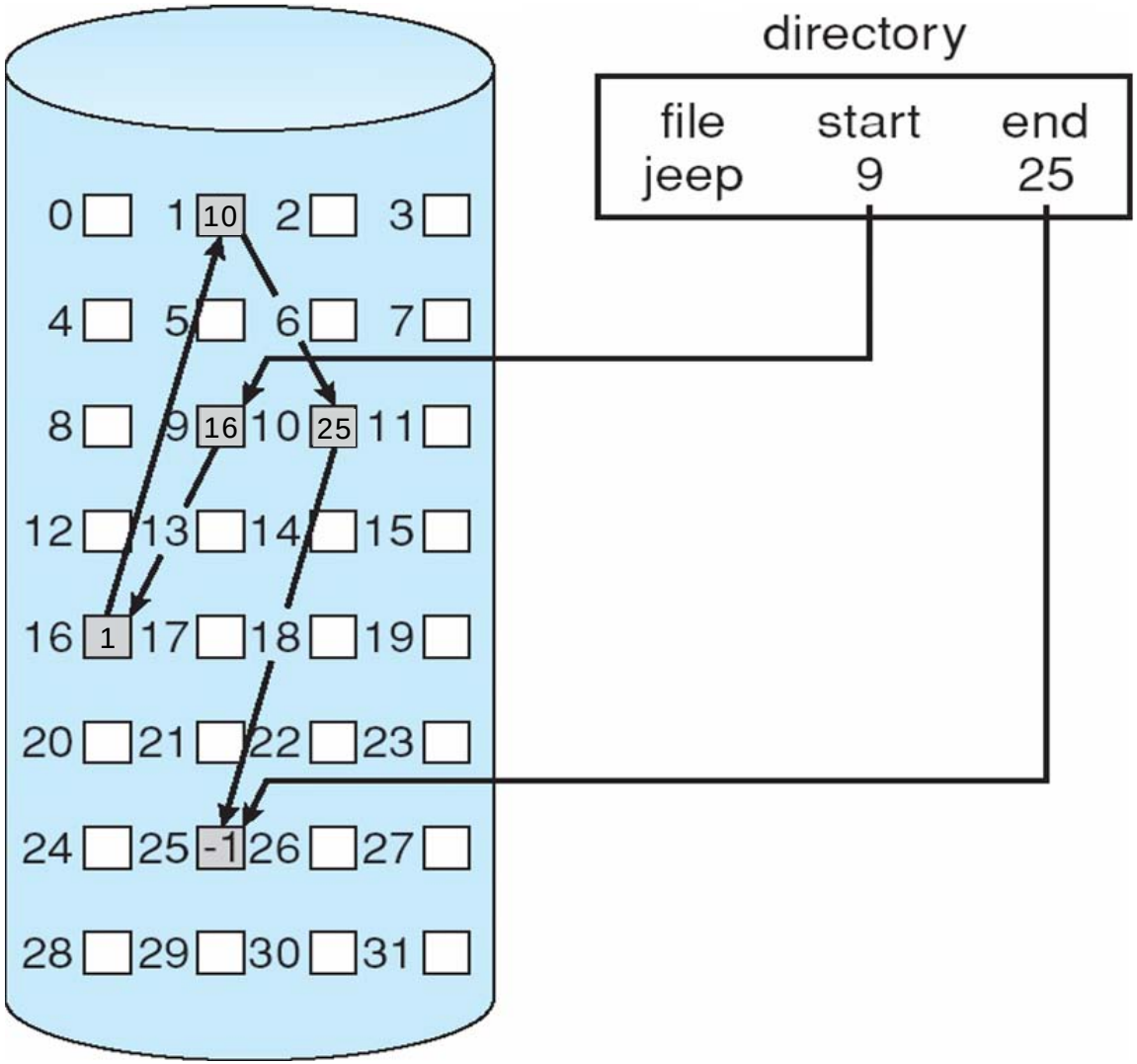
- Each file is a linked list of disk blocks
  - blocks may be scattered anywhere on the disk.



- Pros:
  - Can grow files dynamically,
  - Free list same as file
- Cons:
  - Unreliable (lose block, lose rest of file)
  - Serious con: Bad random access!!!! (seek between each block)
- Technique originally from Palo Alto (First PC, built at Xerox)

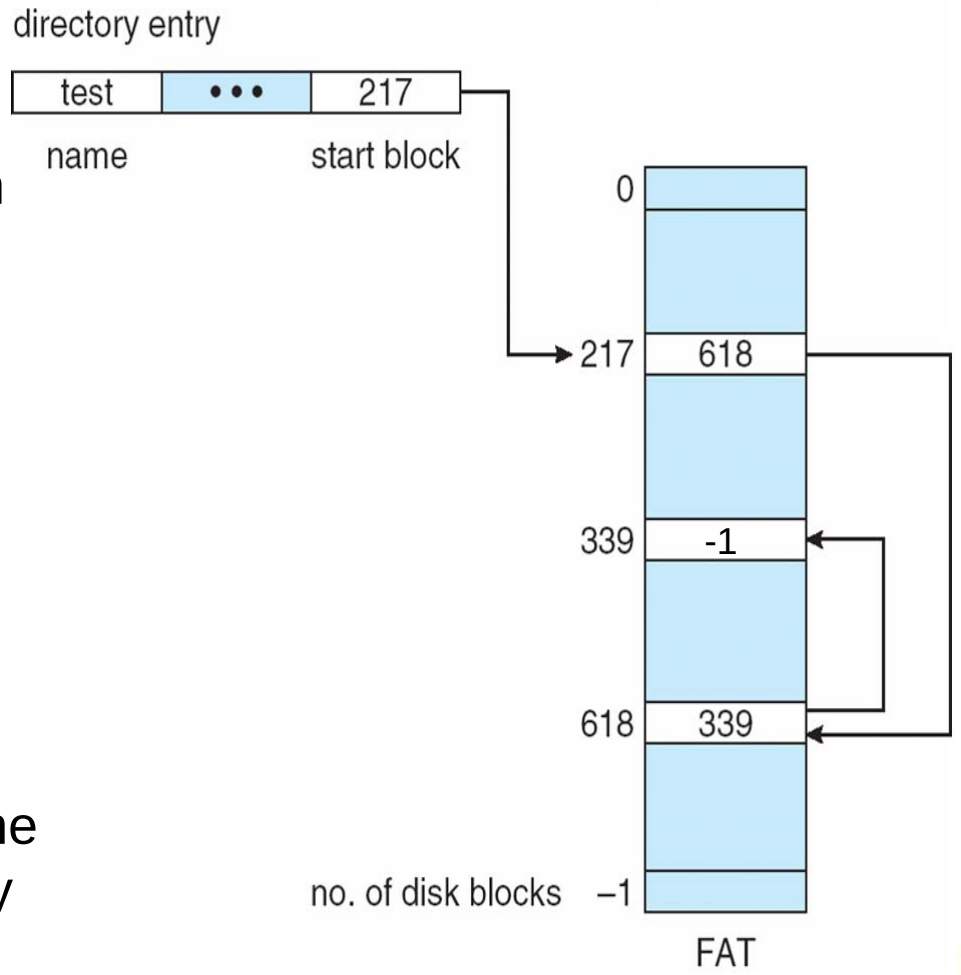


Linked Allocation



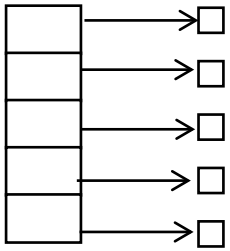
Linked Allocation (FAT)

- MSDOS links blocks/clusters together to create a file
  - Links not in blocks, but in the *File Allocation Table* (FAT)
    - FAT contains an entry for each block on the disk
    - FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - Sequential access expensive unless FAT cached in memory
    - Random access expensive always, but really expensive if FAT not cached in memory (The entire table must be in memory all at once!)

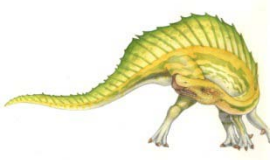


Indexed Allocation

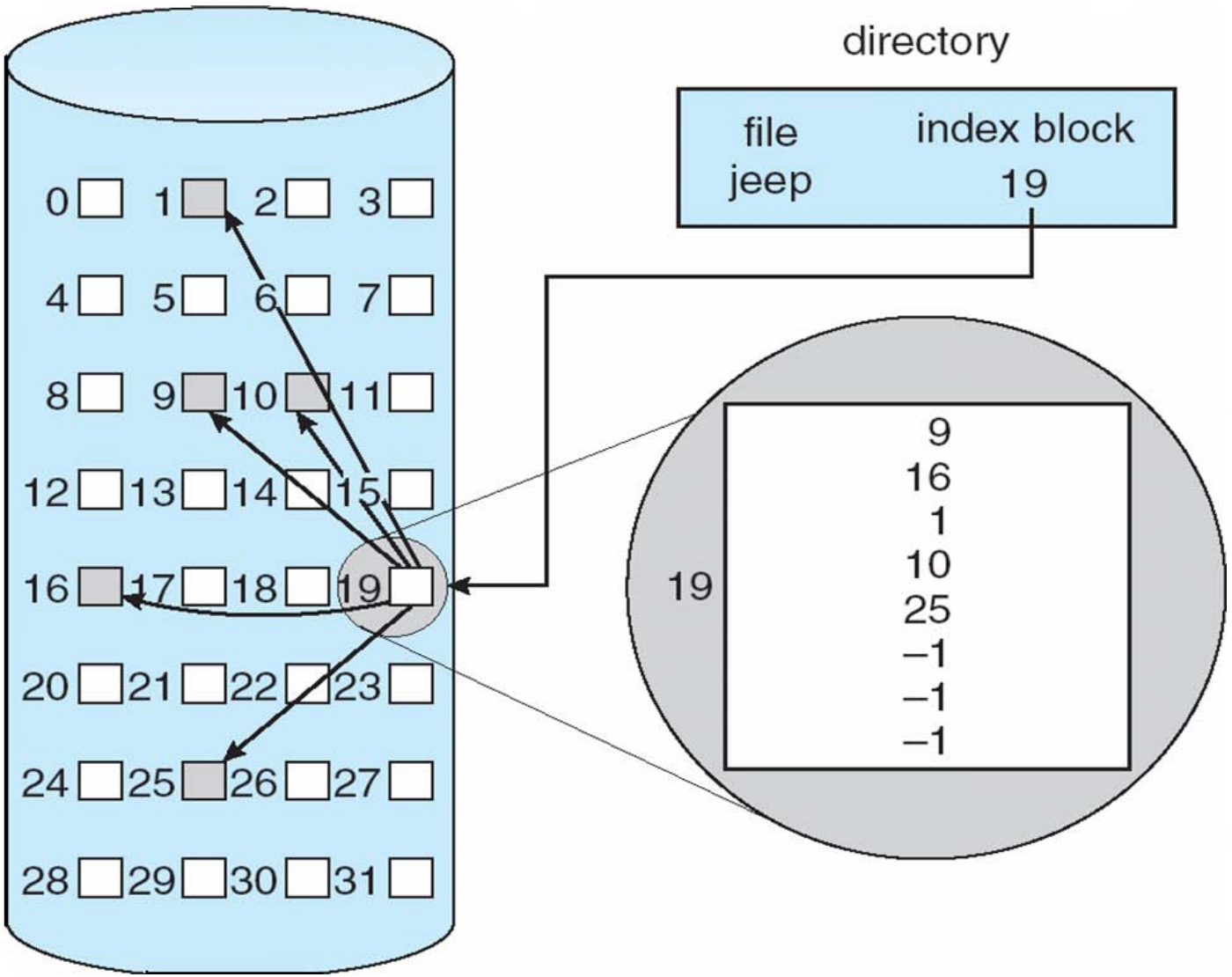
- System Allocates file header block to hold array of pointers big enough to point to all blocks
  - User pre-declares max file size
  - All pointers together into the *index block/table.*
  - Every file must have an index block
- Pros:
  - Random access is fast
  - Dynamic access without external fragmentation, but have overhead of index table
  - Can easily grow up to space allocated for index
- Cons:
  - Clumsy to grow file bigger than table size
  - Still lots of seeks: blocks may be spread over disk
- Logical view:



index table

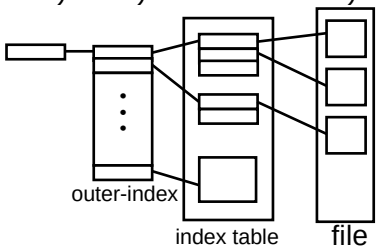


Indexed Allocation

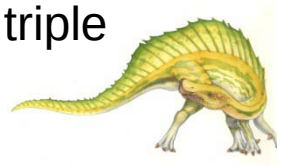


Indexed Allocation

- If the index block is too small, it will not be able to hold enough pointers for large files
- Mechanisms for this purpose include the following:
  - *Linked scheme*: link together *index blocks (*making one to point to another)
  - *Multilevel index:*
    - a first-level index block to point to a set of second-level index blocks, which in turn point to the file
    - *continue to a 3rd , 4th, … levels, depending on the desired maximum file size*

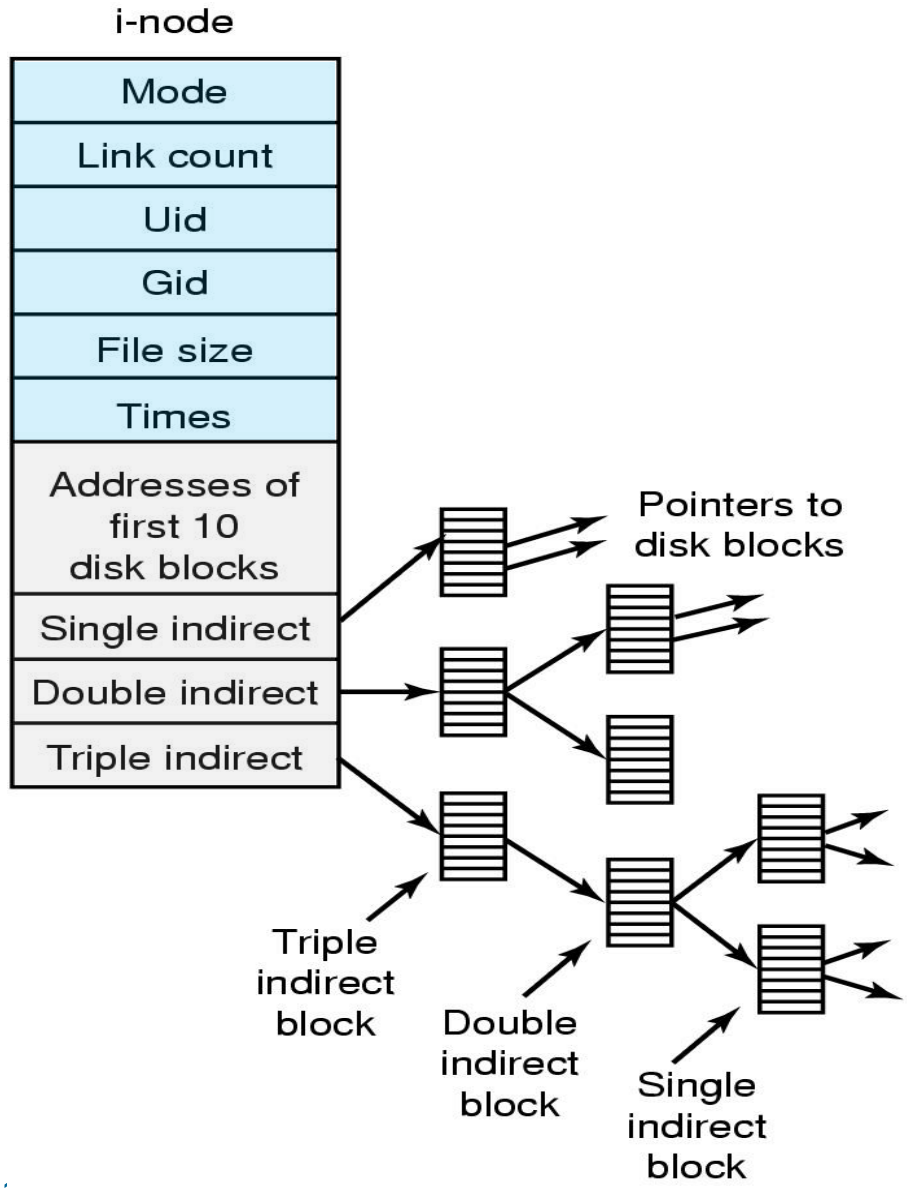


  - *Combined scheme:*
    - Index block contains: Direct blocks, and indirect blocks
    - Indirect blocks: single indirect blocks, double indirect block, triple indirect block
    - E.g. Unix *i-node*



Combined Indexed Files (UNIX 4.1)

- Multilevel Indexed Files (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big file
- File header contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an **"i-node"** in UNIX
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a "triply indirect block" (16M blocks)



Combined Indexed Files (UNIX 4.1)

- The Unix i-node entries

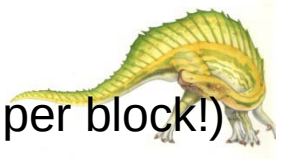
Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Combined Indexed Files (UNIX)

- Examples:
  - How many accesses for block #23? (assume file header accessed on open)? Two: One for indirect block, one for data
  - How about block #5? One: One for data
  - Block #340? Three: double indirect block, indirect block, and data

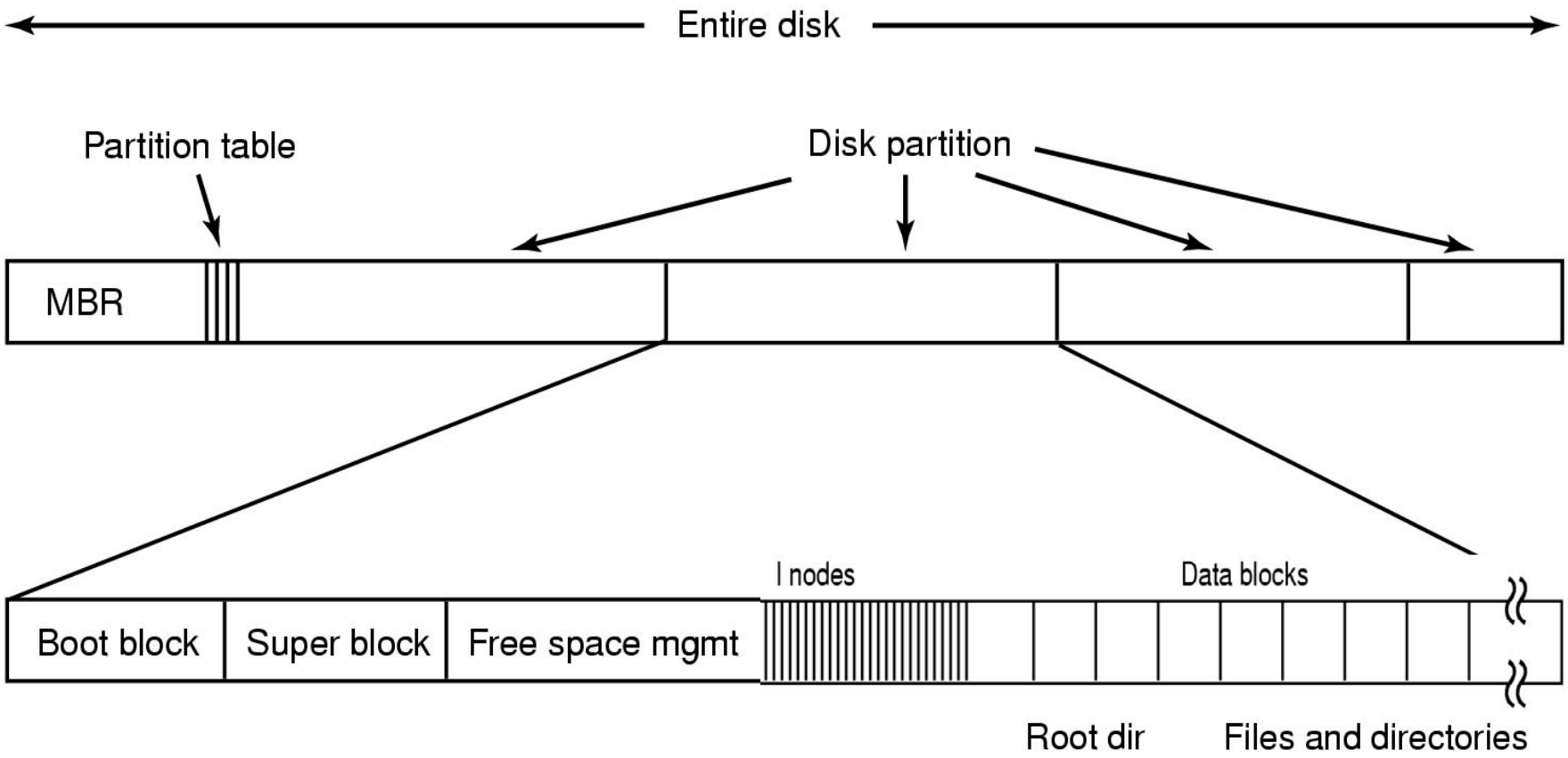
- Basic technique places an upper limit on file size (approx. 16GB)
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

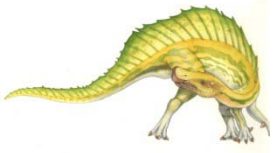
- UNIX i-nodes:
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    - Very large files must read many indirect blocks (four I/Os per block!)



The UNIX file system

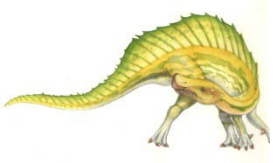
- The layout of the disk (for early Unix systems):





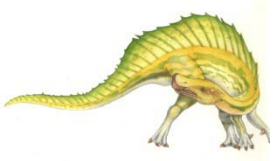
Chapter 10, Sec. 10.3, Silberschatz et al., 8th ed., Walpole, lecture 17

Directories



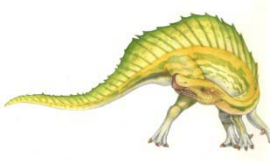
Directories

- Organize files into directories (logically) to obtain:
  - Efficiency – locating a file quickly
  - Naming – convenient to users
    - Two users can have same name for different files
    - The same file can have several different names
  - Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

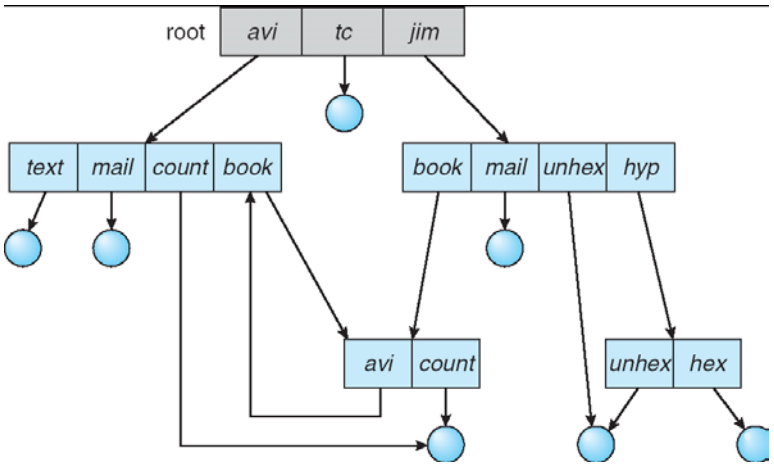


Directories

- Directory or Folder: a relation used for naming
  - Just a table of (file name, inumber) pairs
- Directories organized into a hierarchical structure
  - Seems standard, but in early 70's it wasn't
  - Permits much easier organization of data structures
- Entries in directory can be either files or directories
- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
- Files named by ordered set (e.g., "/programs/p/list")

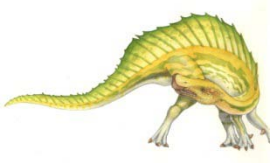


Directory Structure



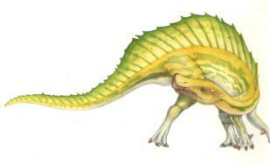
- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - Multiple directory entries point at the same file
  - Soft Links: “shortcut” pointers to other files
    - Implemented by storing the logical name of actual file

- Name Resolution: The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network



Implementing directories

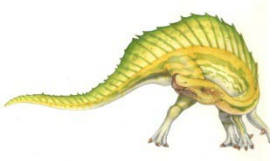
- How are directories constructed?
  - List of files
    - File name
    - File Attributes
- Simple Approach (e.g. FAT):
  - Put all attributes in the directory
- Unix Approach (i-nodes):
  - Directory contains
    - File name
    - I-Node number
  - I-Node contains
    - File Attributes



Implementing directories

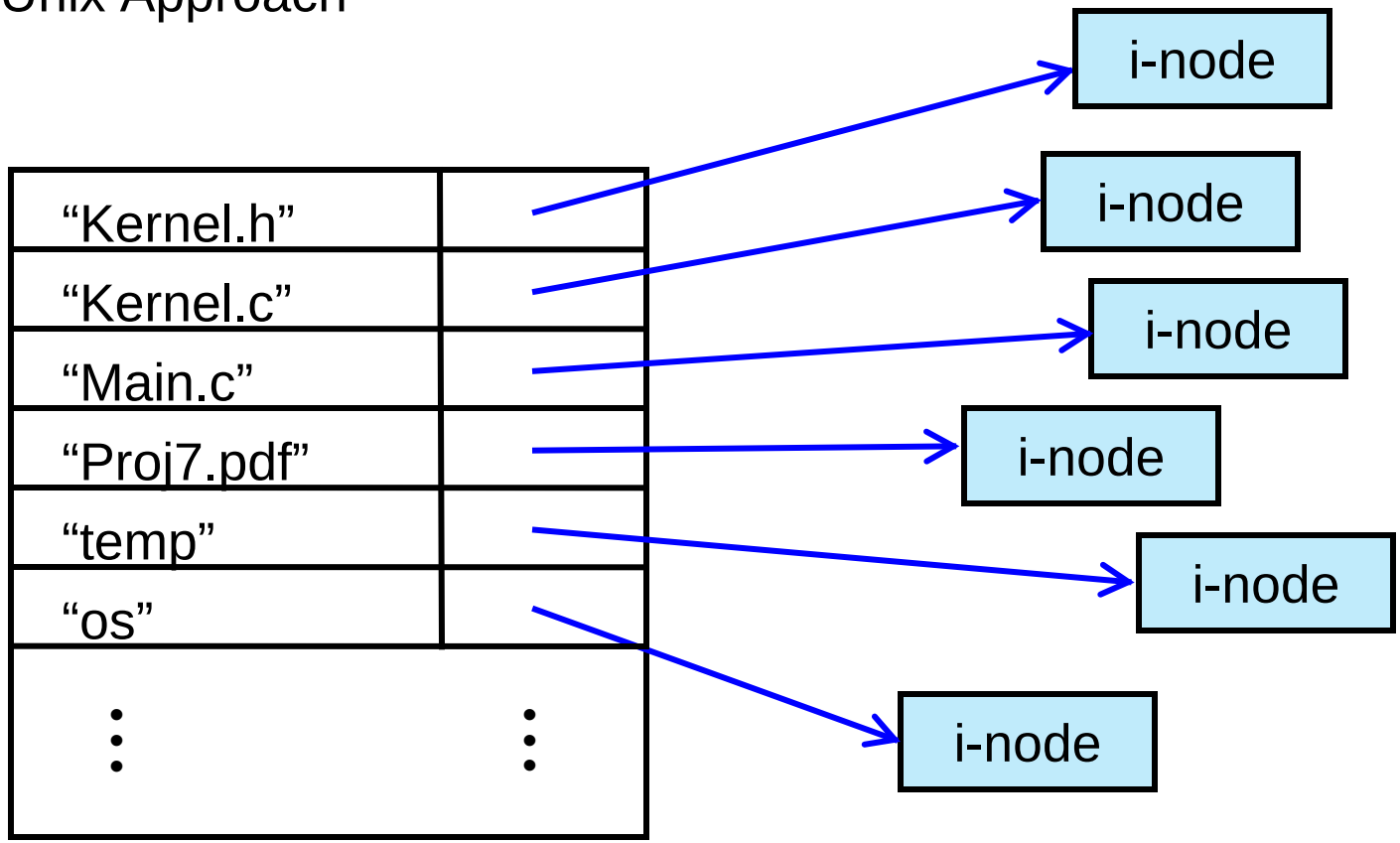
- Simple Approach

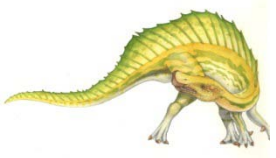
“Kernel.h”	attributes
“Kernel.c”	attributes
“Main.c”	attributes
“Proj7.pdf”	attributes
“temp”	attributes
“os”	attributes
⋮	⋮



Implementing directories

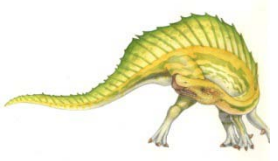
- Unix Approach





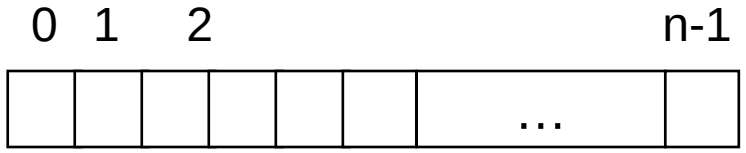
Chapter 11, Sec. 11.5, Silberschatz et al., 8th ed.,

Free space management



Free-Space Management

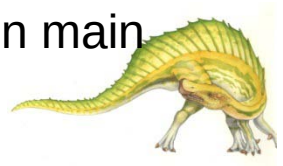
- Bit vector (*n* blocks)



$$bit[i] = \begin{cases} 0 \Rightarrow block[i] \text{ free} \\ 1 \Rightarrow block[i] \text{ occupied} \end{cases}$$

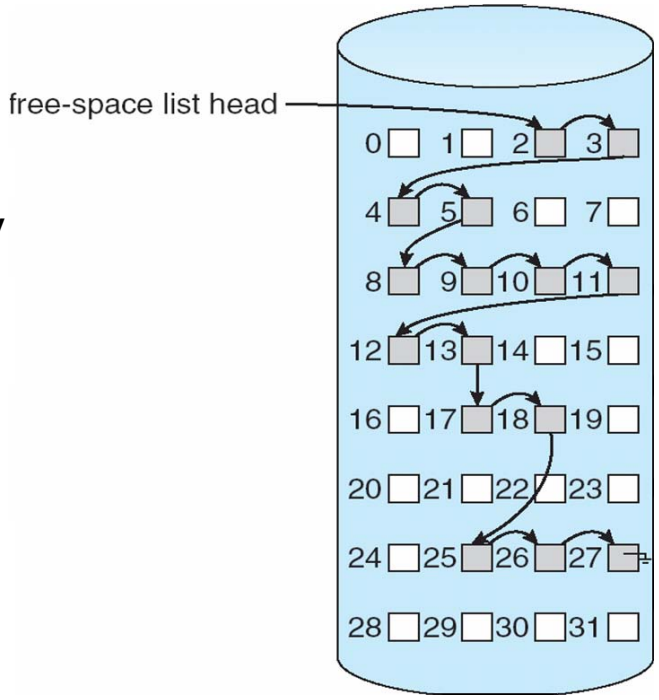
  - Block number calculation = (number of bits per word) * (number of 0-value words) + offset of first 1 bit
  - Bit map requires extra space
    - Example: block size = 2^{12} bytes; disk size = 2^{30} bytes (1 gigabyte)

      $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
  - Easy to get contiguous files
  - Many CPUs provide bit-manipulation instructions that can be used effectively
  - But, bit vectors are inefficient unless the whole vector is kept in main memory (may be a problem with very large disk)



Free-Space Management

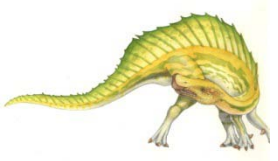
- Linked list (free list)
  - No waste of space
  - Not very efficient: traversing the list may require substantial I/O time
  - Cannot get contiguous space easily
  - FAT free-block accounting into the allocation data structure itself



- Other linked-list based approaches:
  - Grouping: store the addresses of *n* free blocks in the first block. The first *(n-1)* blocks are free. The last block has the addresses of another *n* free blocks. And so on.
  - Counting: if several consecutive blocks are free, rather than keeping a list of *n* free blocks, keep only the address of the first and the counting *(n)* of free contiguous blocks following the first.

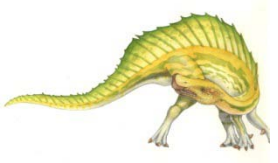
Free-Space Management

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk
  - Solution:
    - Set bit[*i*] = 0 in disk
    - Allocate block[*i*]
    - Set bit[*i*] = 0 in memory



Chapter 11, Sec. 11.6, Silberschatz et al., 8th ed., Walpole, lecture 18

Efficiency and Performance

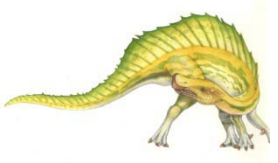


Disk space management

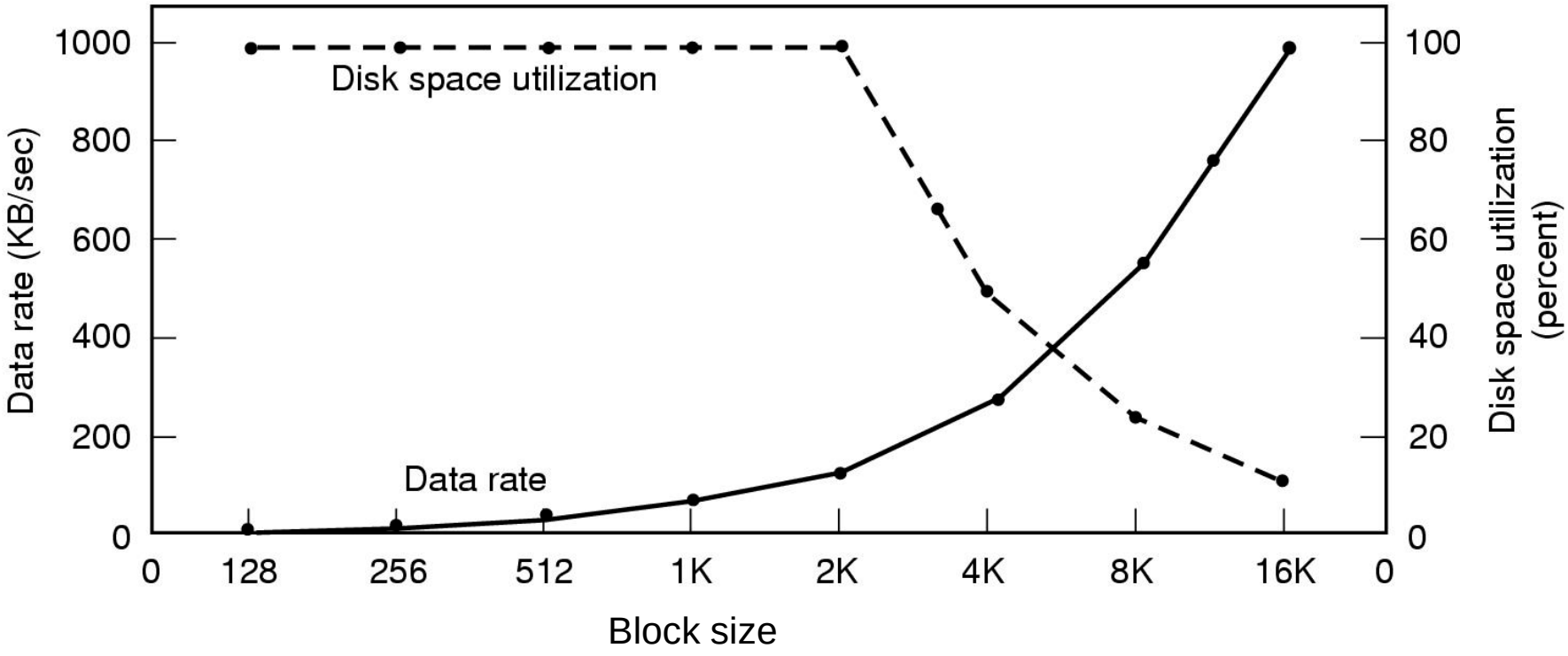
- A disk block size must be chosen (=page size?, =sector size?, =track size?)

- Large block sizes:
  - Internal fragmentation
  - If lots of very small files → waste is greater.
  - Lower disk space utilization
  - But, better performance

- Small block sizes:
  - More seeks → file access will be slower
  - Bigger data structures (e.g. FAT)
  - Poor performance
  - But, better disk utilization



Block size tradeoff



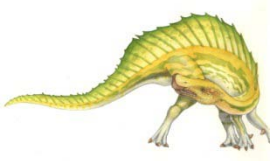
Assumption: All files are 2K bytes

Given: Physical disk properties

Seek time=10 msec

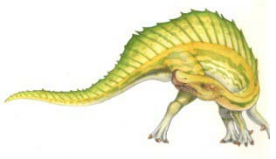
Transfer rate=15 Mbytes/sec

Rotational Delay=8.33 msec * 1/2



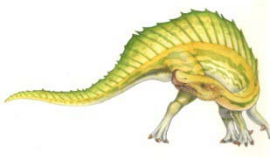
Improving file system performance

- Memory mapped files
  - Avoid system call overhead
- Buffer cache
  - Avoid disk I/O overhead
- Careful data placement on disk
  - Avoid seek overhead
- Log structured file systems
  - Consistency checking and recovery on crash
  - Avoid seek overhead for disk writes (reads hit in buffer cache)



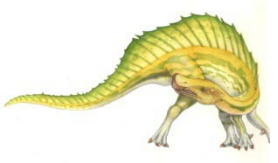
Memory-mapped files

- Conventional file I/O
  - Use system calls (e.g., open, read, write, ...) to move data from disk to memory
- Observation
  - Data gets moved between disk and memory all the time without system calls
    - Pages moved to/from PAGEFILE by VM system
  - Do we really need to incur system call overhead for file I/O?



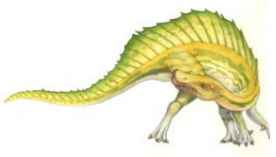
Memory-mapped files

- Why not “map” files into the virtual address space
  - Place the file in the “virtual” address space
  - Each byte in a file has a virtual address
- To read the value of a byte in the file:
  - Just load that byte’s virtual address
    - Calculated from the starting virtual address of the file and the offset of the byte in the file
  - Kernel will fault in pages from disk when needed
- To write values to the file:
  - Just store bytes to the right memory locations
- Open & Close syscalls → Map & Unmap syscalls

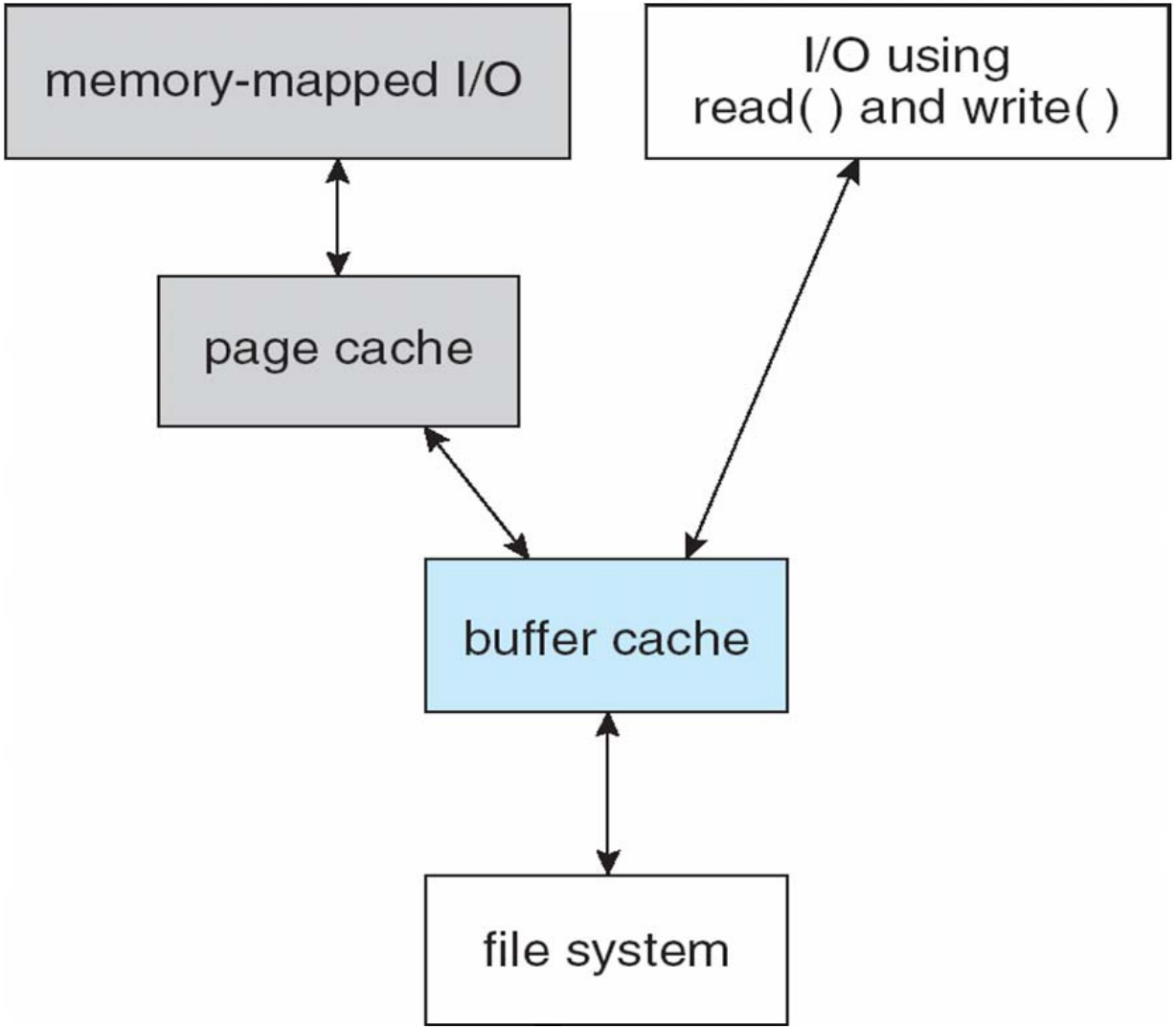


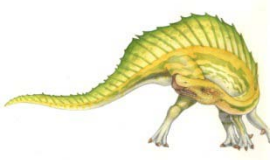
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure



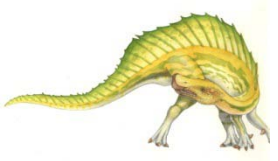
I/O Without a Unified Buffer Cache



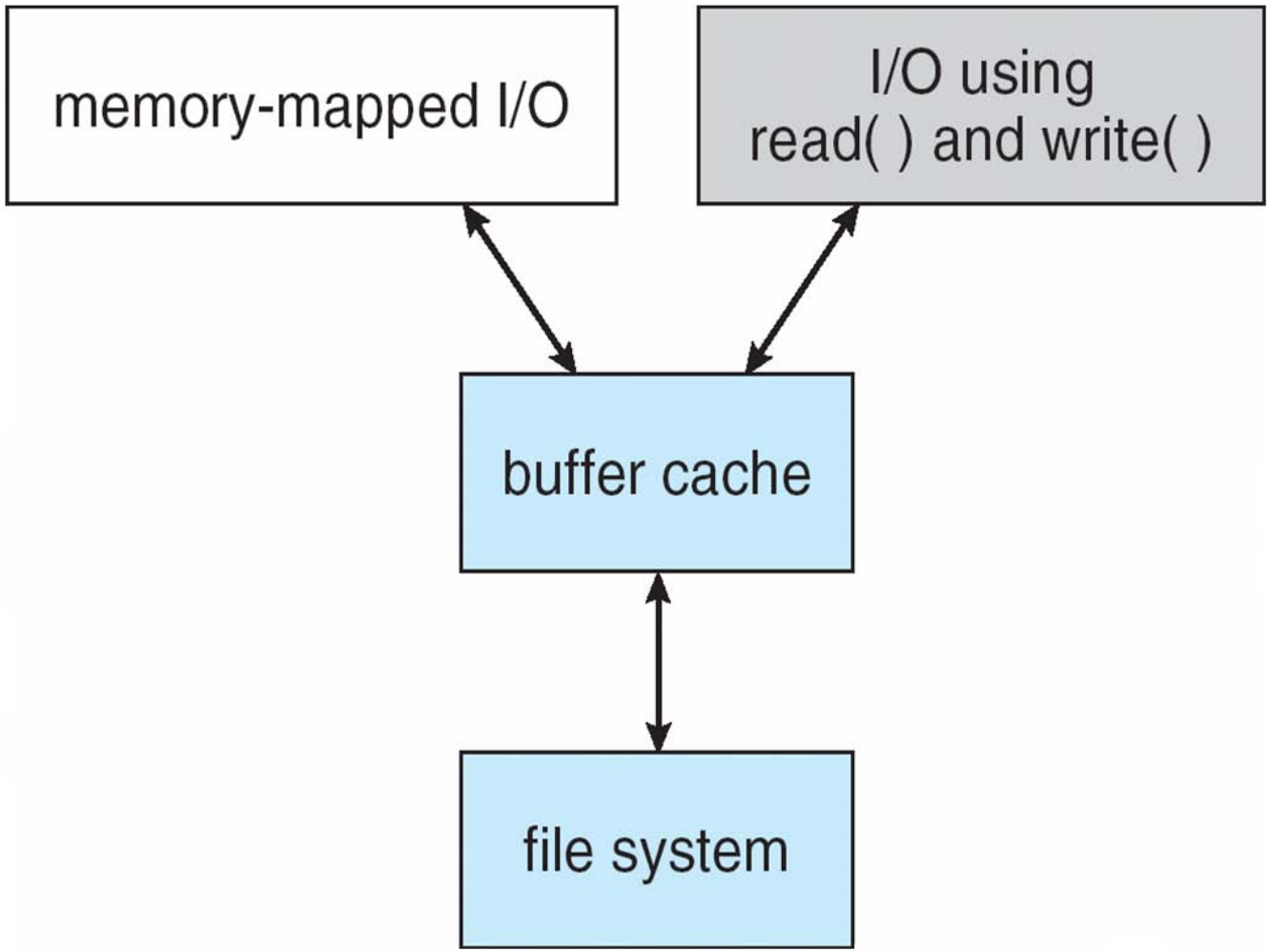


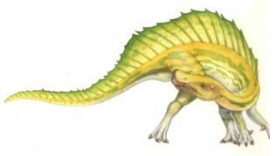
Buffer cache

- Observations:
  - Once a block has been read into memory it can be used to service subsequent read/write requests without going to disk
  - Multiple file operations from one process may hit the same file block
  - File operations of multiple processes may hit the same file block
- Idea: maintain a "*block cache*" (or "*buffer cache*") in memory
  - When a process tries to read a block check the cache first



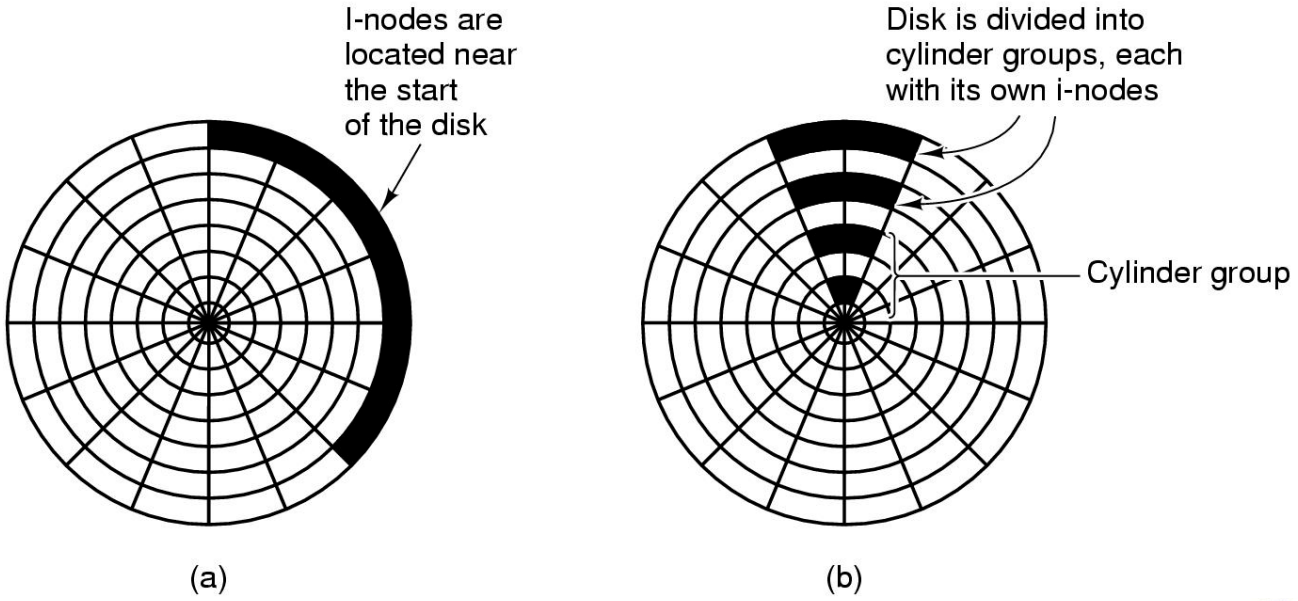
I/O Using a Unified Buffer Cache

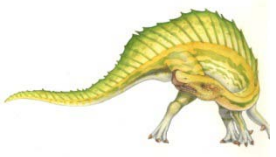




Careful data placement

- Idea
  - Break disk into regions
    - “Cylinder Groups”
  - Put blocks that are “close together” in the same cylinder group
    - Try to allocate i-node and blocks in the file within the same cylinder group





Log Structured File Systems

- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**
- All transactions are written to a **log**
  - A transaction is considered **committed** once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed