file System

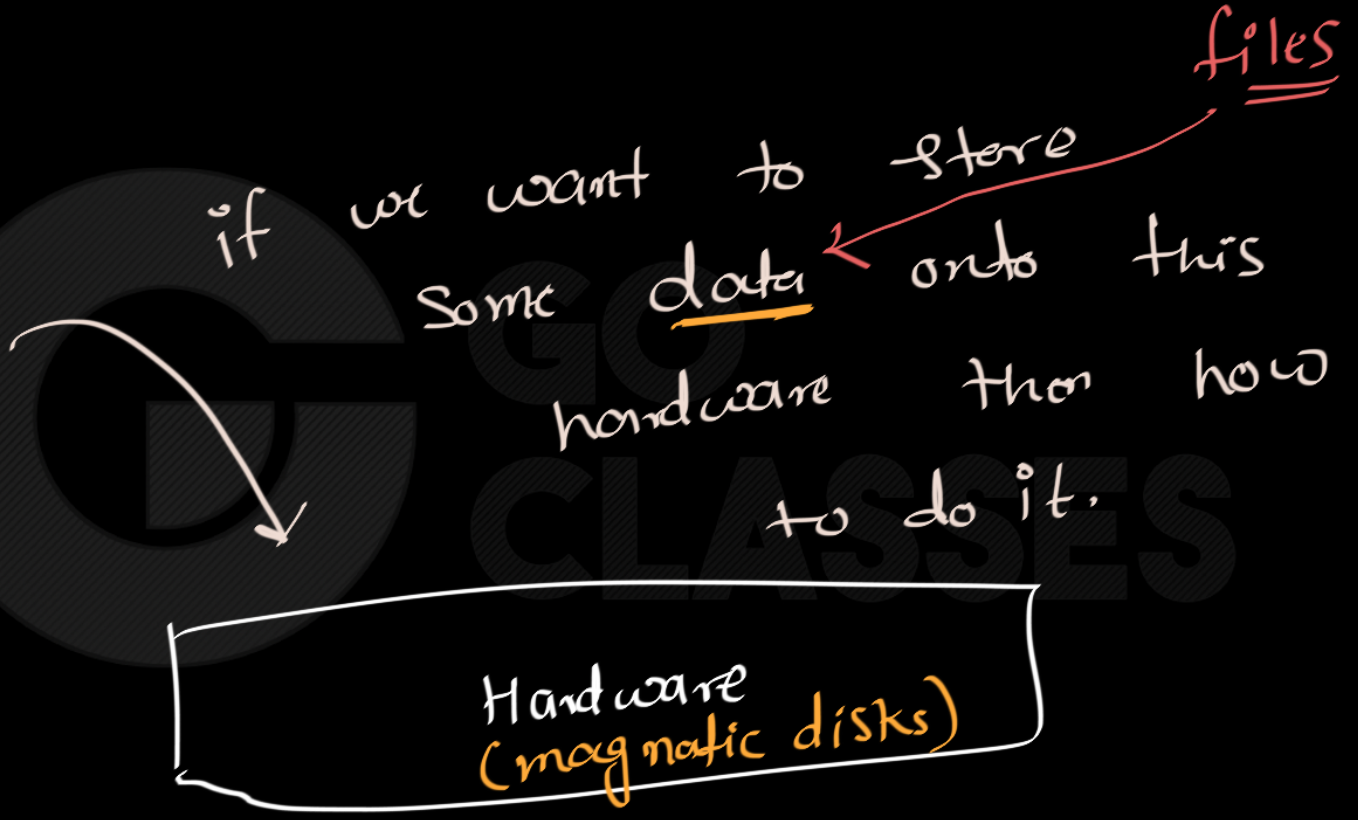
implementation

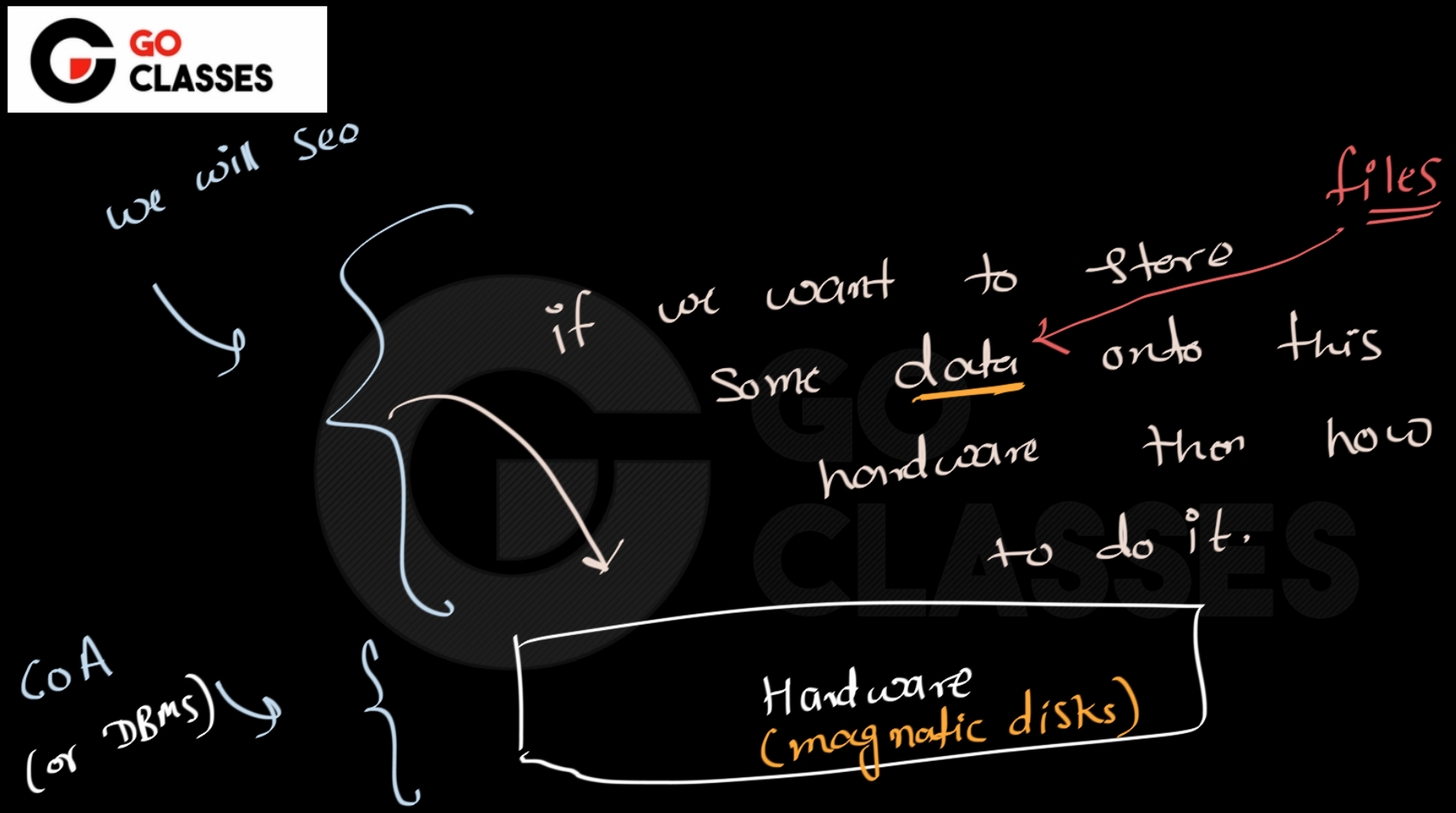
how does secondary memory looks like

Hardware

magnetic tapes

implementation


files
if we want to store
some data onto this
hardware then how
to do it.
Hardware
(magnatic disks)


GO CLASSES
We will see
if we want to store some data onto this hardware then how to do it.
files
CoA (or DBMS)
Hardware (magnatic disks)

File System (implementation)

What is File ?

Data in some format

- pdf
- mp3
- mp4

What is File System ?

- Provides a way to store files
- Provides a way to name files
- Provides a way to store metadata of file

....

How to manage the files on hardware.

File types – name, extension

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

Figure 4-1. Some typical file extensions.

Source: Tanenbaum

File Operations

- Create
- Rename
- Open
- Read
- Write

..etc

GO CLASSES

Store files on disk

what should be key features for file system

→ easy to fetch (efficient fetching)

you should not take too much time to just calculate where is your file.

→ file should not get mixed up.

→

→

GO CLASSES

Plan

We will see few common ways to implemend file system.

→ method 1 : this is how you should store Contigious / linked / indexed files

→ method 2 : ''

→ method 3 : ''

How morden file systems are implemented

→ windows

→ Mac

→ linux

→ unix

(90% method 2 + 10% method 3)

Major component of any file system:

"How to store files"

Sector or Block = unit of atomicity

modify one byte in a file

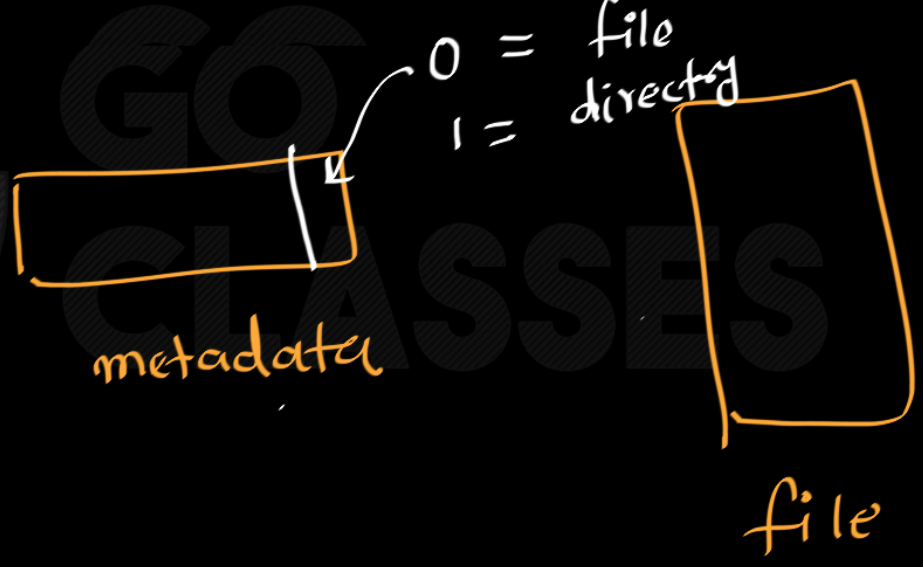
- **Disk reads/writes in terms of sectors, not bytes**
  - Read/write single sector

Size of block : 4KB

- **How to write a single byte? “Read-modify-write”**
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk

Directory

A File (in UNIX, everything is file) with special bit set in metadata to indicate directory.



Directories

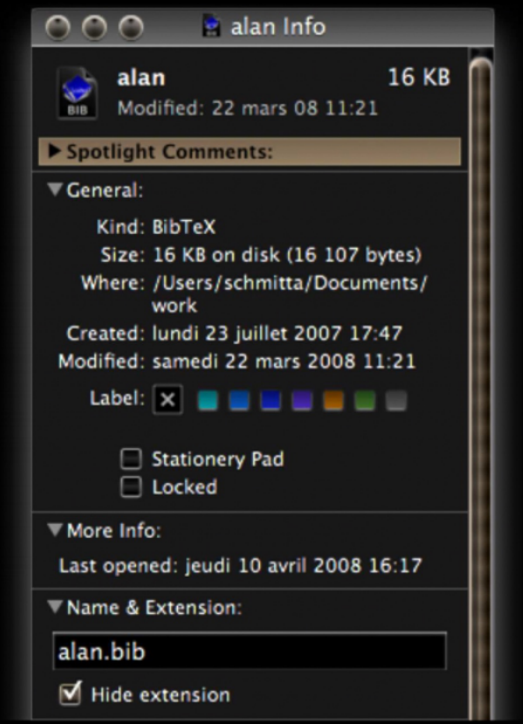
File Metadata

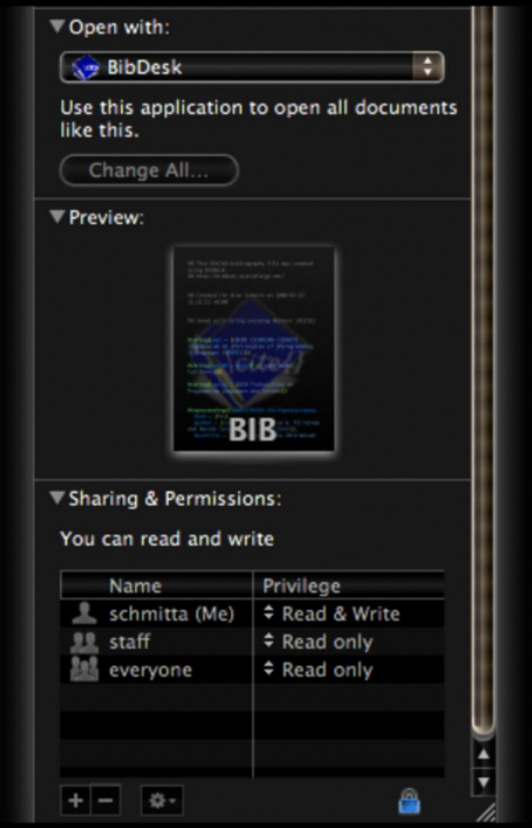
- Name – only information kept in human-readable form
- Identifier – unique tag (number) identifies file within file system (inode number in UNIX)
- Location – pointer to file location on device
- Size – current file size
- Protection – controls who can do reading, writing, executing
- Time, date, and user identification – data for protection, security, and usage monitoring

- How is metadata stored? (inode in UNIX)

File Metadata

Stored in some data structure = inode.





Inodes

Inodes

- Store the metadata for a file (which data blocks belong to the file, file size, owner, access rights, ...)
- Inode stands for index node

- **Inodes** - An inode is the data structure that describes the meta data of files (regular files and directories). One inode represents one file or one directory. An inode is composed of several fields such as ownership, size, modification time, mode (permissions), reference count, and data block pointers (which point to the data blocks that store the content of the file). Note that the inode does not include a file/directory name. A file's inode number can be found using the `ls -i` command.

- for every file or directory ⟺ one inode
- every inode has one number inode number

File(or Directory) ≡ inode (header) + data

↓

metadata

What does inode of file contain ?

What does data of file contain ?

What does inode of file contain ?

→ metadata

What does data of file contain ?

→ data

GO CLASSES

usr / Desktop / memory_mangement. pdf

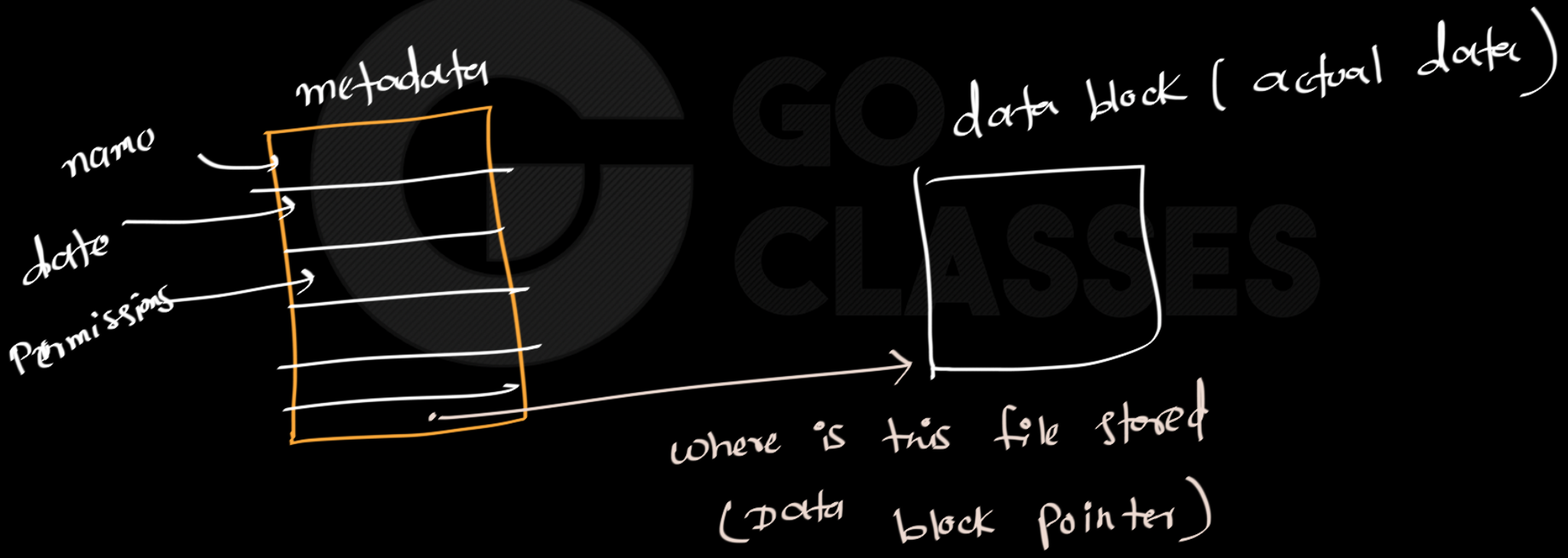
inode for Desktop

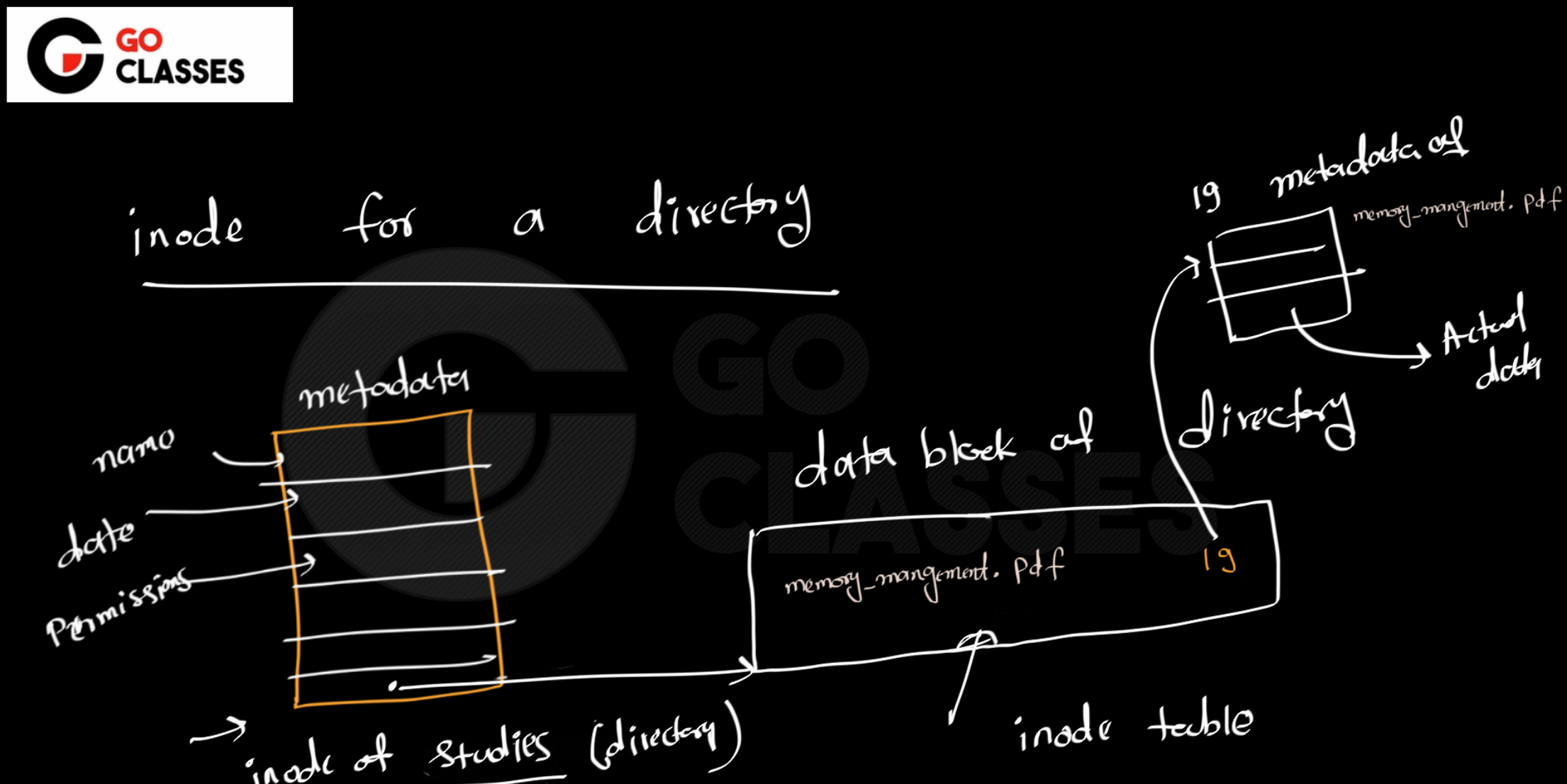
memory_mangement. pdf

19 ~ inode no. of file memory_mangement. pdf

How to get that inode number?

'inode for a file




inode for a directory
metadata
name
date
Permissions
inode of Studies (directory)
data block of directory
memory_mangement. Pdf
19
inode table
19 metadata of
memory_mangement. Pdf
Actual data

Inodes: How to index the content of a file?

Indexing inodes

- An inode is identified by an inumber
- Corresponds to its index in the inode table
- Computing in which sector an inode is stored is easy (inputs: inode table start address, inumber, size of inode, size of block, size of sector)

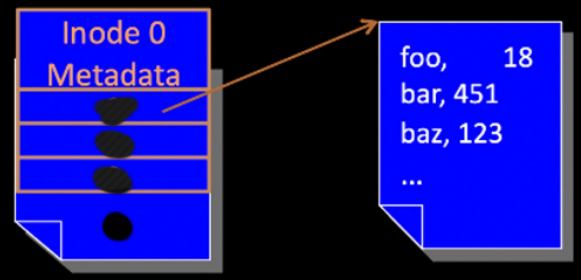
What does inode of directory contain ?

What does data of directory contain ?

What does inode of directory contain ?

What does data of directory contain ?

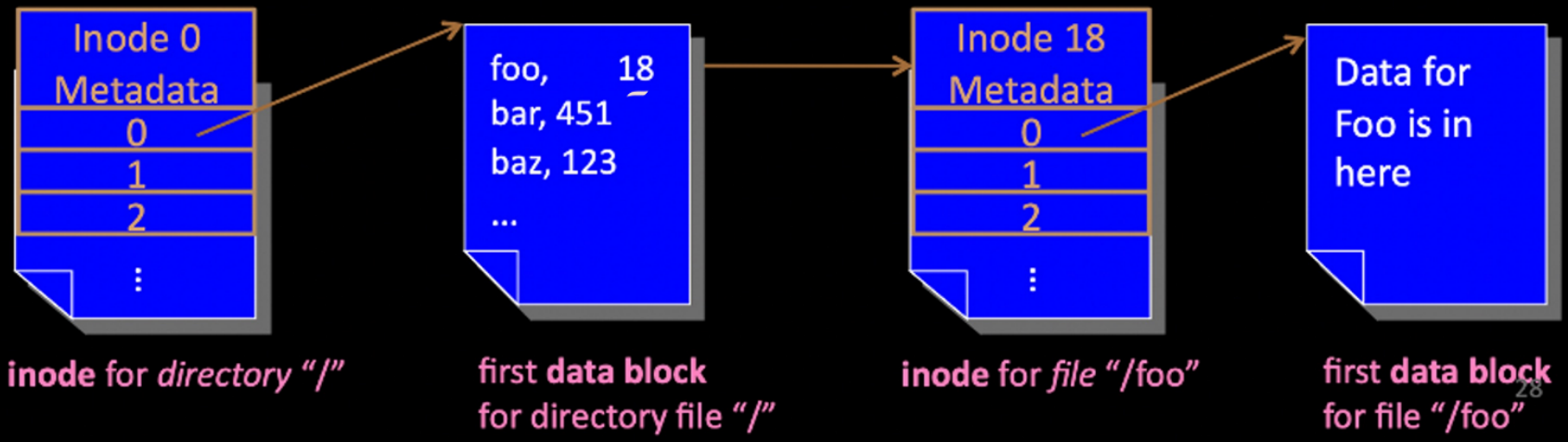
Inode table



inode for *directory "/"*

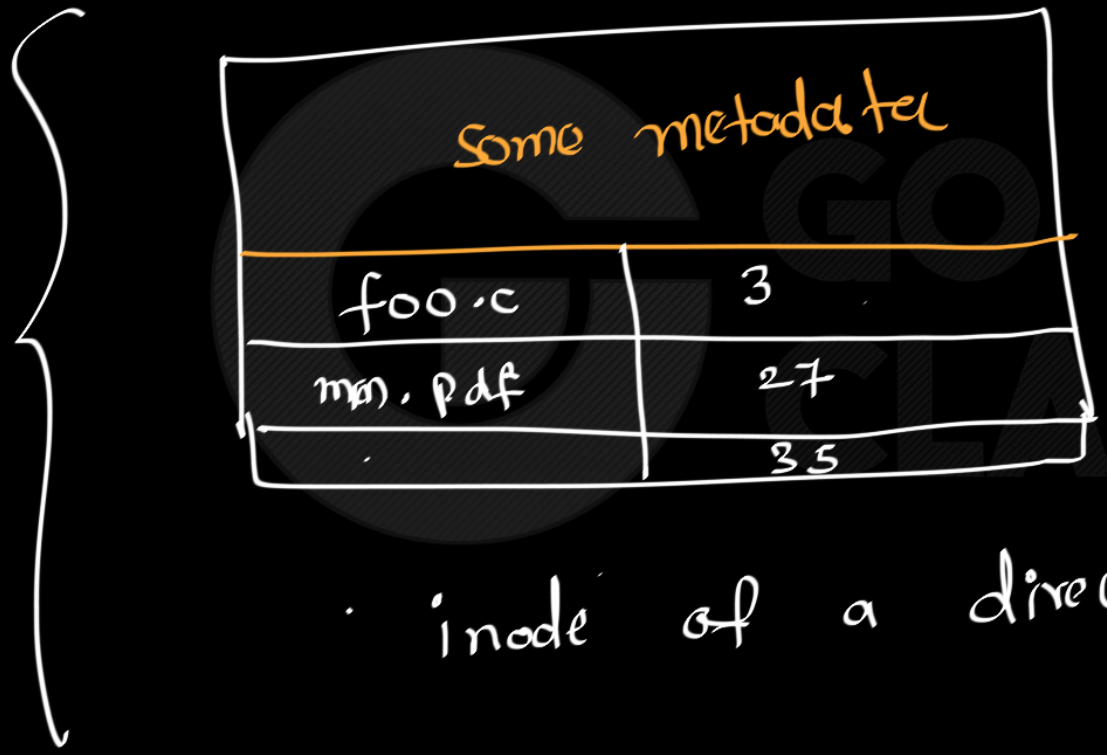
first **data block**
for directory file "/"

(Inode table)


Inode 0
Metadata
0
1
2
⋮
foo, 18
bar, 451
baz, 123
...
Inode 18
Metadata
0
1
2
⋮
Data for
Foo is in
here
inode for directory "/"
first data block
for directory file "/"
inode for file "/foo"
first data block
for file "/foo"

GO CLASSES

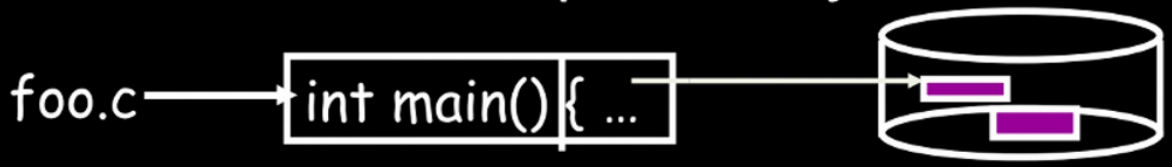
windows 95



← is this a Possible implementation?

Yes

- **File abstraction:**
  - User's view: named sequence of bytes



  - FS's view: collection of disk blocks
  - File system's job: translate name & offset to disk blocks:

(foo.c, 512)

https://www.scs.stanford.edu/17wi-cs140/notes/file_systems.pdf

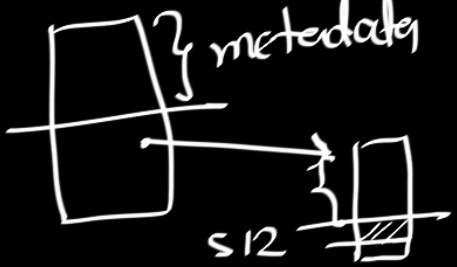
- **Like page tables, file system metadata are simply data structures used to construct mappings**
  - Page table: map virtual page # to physical page #

  - Directory: map name to disk address or file #

inode number of file

  - File metadata: map byte offset to disk block address

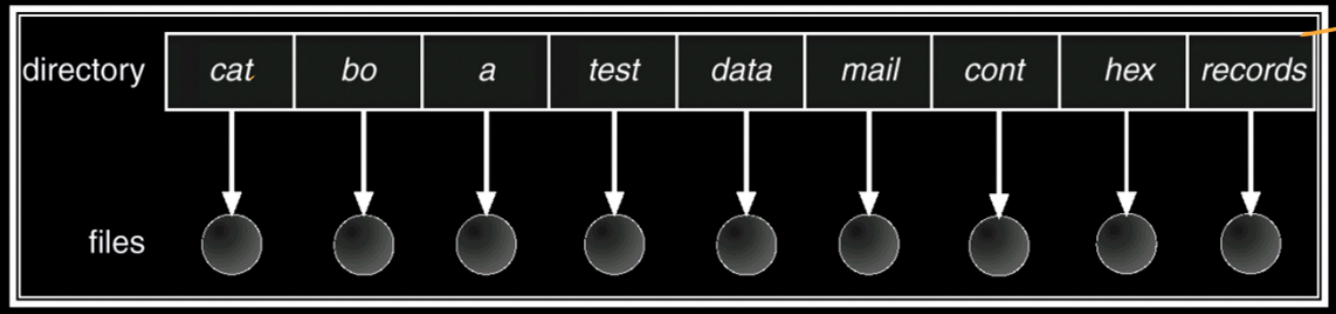
44th inode



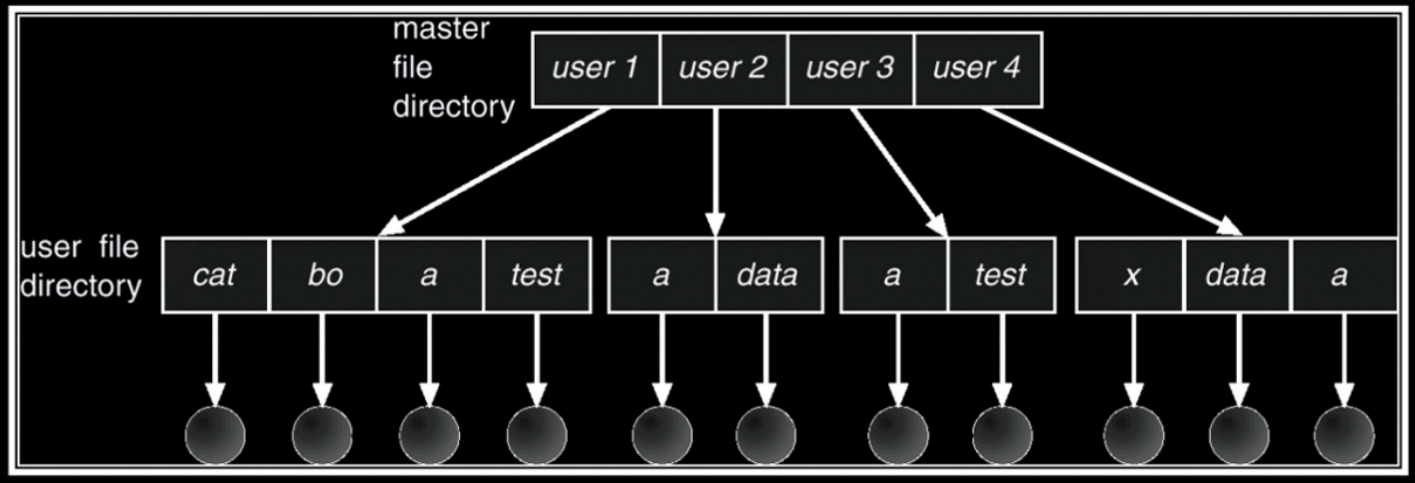
A short history of directories

- **Approach 1: Single directory for entire system**
  - Put directory at known location on disk
  - Directory contains $\langle \text{name}, \text{inumber} \rangle$ pairs
  - If one user uses a name, no one else can
  - Many ancient personal computers work this way
- **Approach 2: Single directory for each user**
  - Still clumsy, and `ls` on 10,000 files is a real pain
- **Approach 3: Hierarchical name spaces**
  - Allow directory to map names to files *or other dirs*
  - File system forms a tree (or graph, if links allowed)
  - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)

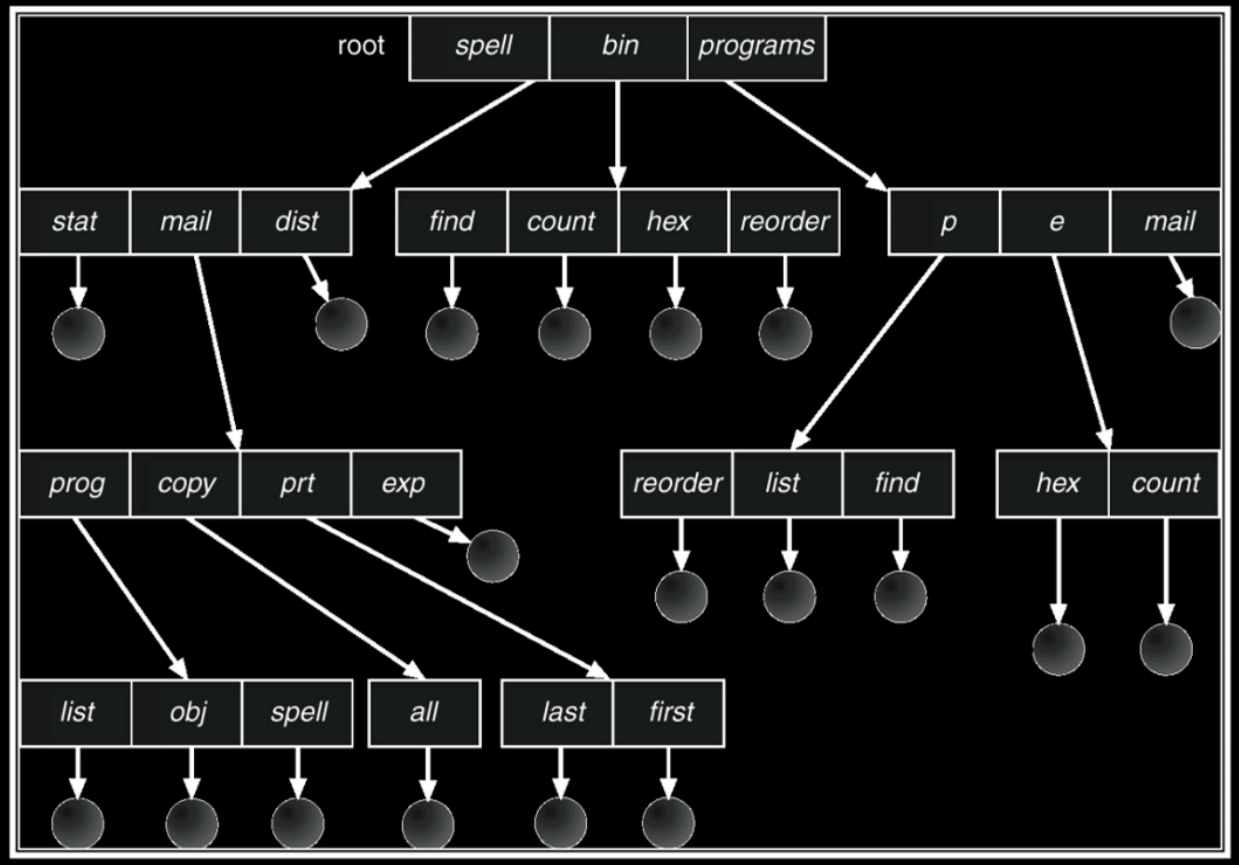
- Single-Level Directory



- Two-Level Directory



- Tree-Structured Directories

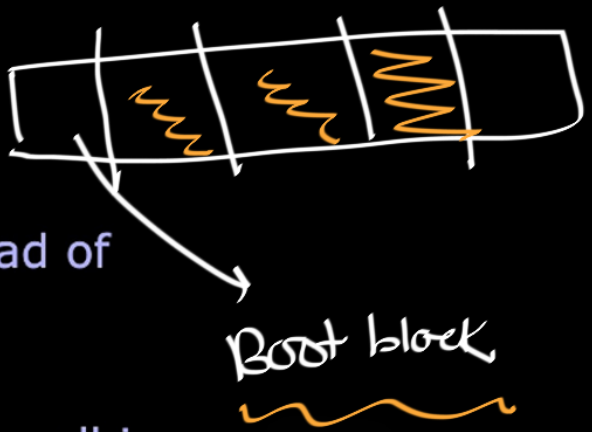


modern systems


Disk
all inodes at one place
actual data of file

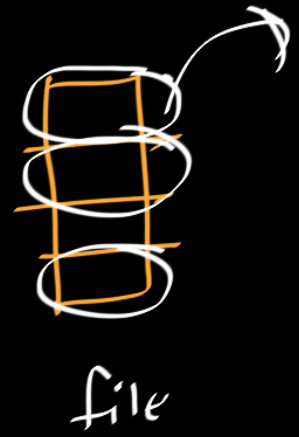
(Old) Unix disks are divided into five parts …

- Boot block
  - can boot the system by loading from this block
- Superblock
  - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
  - contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area ← HUGE
  - fixed-size blocks; head of freelist is in the superblock
- Swap area
  - holds processes that have been swapped out of memory



- Boot block: usually contains bootstrap code
- Super block: size, # files, free blocks, etc.
- Inode list
  - size fixed when configuring file system
  - contains *all* inodes
- Data blocks: file data (Huge area)
- Files, directories organized into tree-like structure

1. (16 points total) Short answer questions:
   a. (2 points) What UNIX structure is used to keep track of the sectors allocated to a given file?



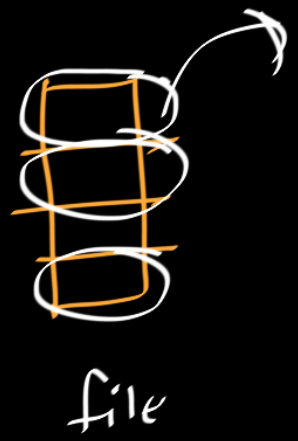
   b. (2 points) What is the smallest addressable piece of data on a disk drive?

   c. (2 points) What is a persistent, named collection of data?

https://hkn.eecs.berkeley.edu/examfiles/cs162_sp03_mt2_sol.pdf

1. (16 points total) Short answer questions:
   a. (2 points) What UNIX structure is used to keep track of the sectors allocated to a given file?

inode



   b. (2 points) What is the smallest addressable piece of data on a disk drive?

Sector

   c. (2 points) What is a persistent, named collection of data?

https://hkn.eecs.berkeley.edu/examfiles/cs162_sp03_mt2_sol.pdf

1. (16 points total) Short answer questions:
   a. (2 points) What UNIX structure is used to keep track of the sectors allocated to a given file?

i-node, file header, or index block

   b. (2 points) What is the smallest addressable piece of data on a disk drive?

Disk sector or block

   c. (2 points) What is a persistent, named collection of data?

File or directory

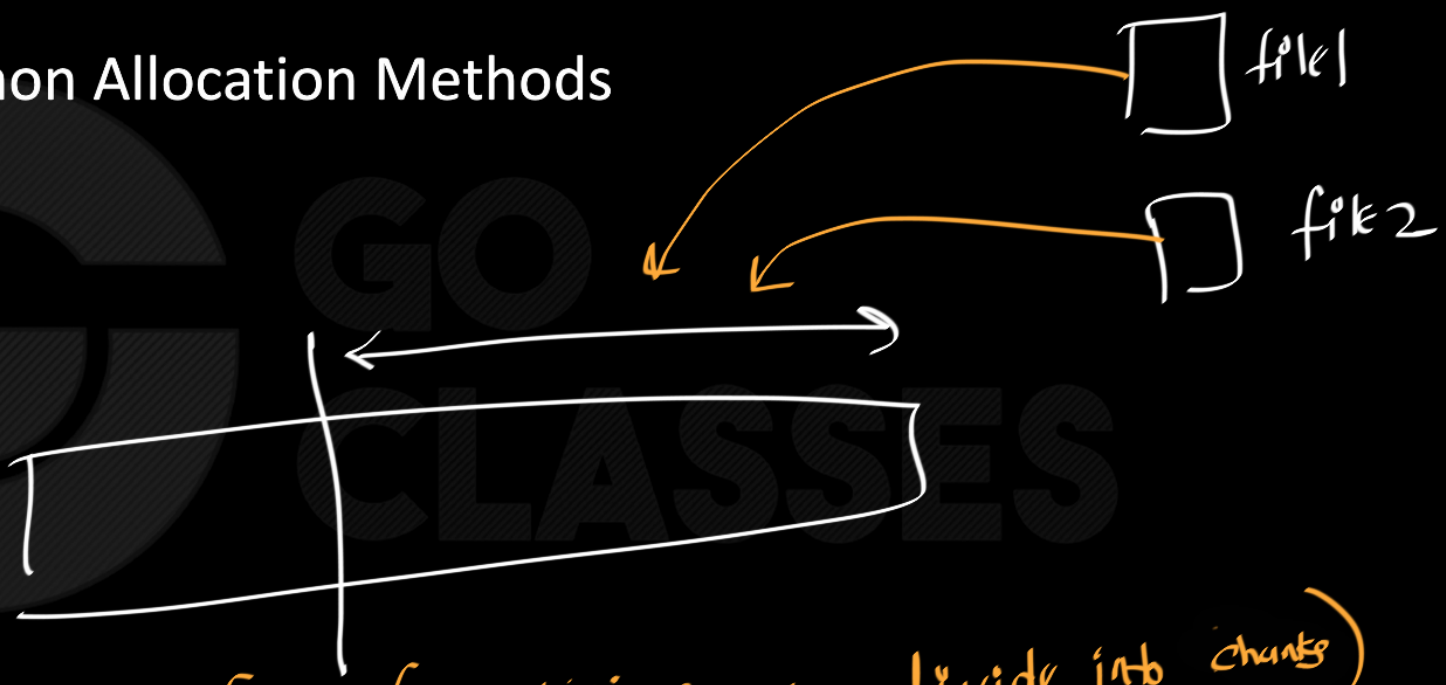
File Implementation

How to allocate disk space to files ?

How to allocate disk space to files ?

There are few common Allocation Methods

(see next slide)



→ How to store file (contigious or divide into chunks)

→ how inodes are organised.

Allocation strategies

- Various approaches (similar to memory allocation)
  - Contiguous
    - Extent-based (Variation to Contiguous)
  - Linked
    - FAT tables (Variation to Linked)
  - Indexed
    - Multi-Level Indexed (Variation to Indexed)
- Key metrics
  - Fragmentation (internal & external)?
  - Grow file over time after initial creation?
  - Fast to find data for sequential and random access?
  - Easy to implement?
  - Storage overhead?

Allocation strategies

- Various approaches (similar to memory allocation)
  - Contiguous
    - Extent-based (Variation to Contiguous)
  - Linked
    - FAT tables (Variation to Linked)
  - Indexed
    - Multi-Level Indexed (Variation to Indexed)

3+3 = 6 methods

windows
mac

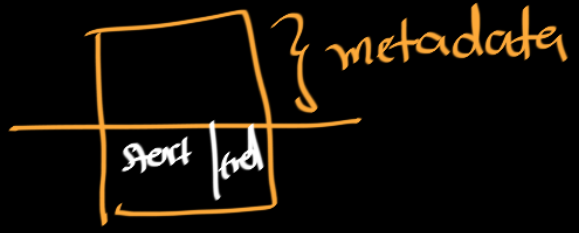
- Key metrics
  - Fragmentation (internal & external)?
  - Grow file over time after initial creation?
  - Fast to find data for sequential and random access?
  - Easy to implement?
  - Storage overhead?

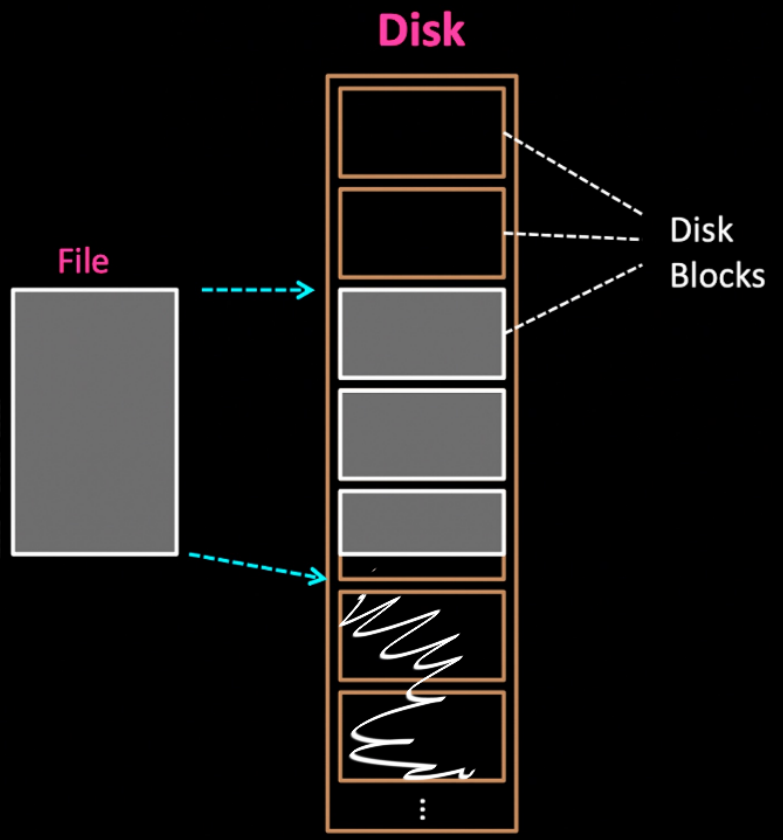
Contiguous allocation

- User specifies length, file system allocates space all at once

- Metadata (inode):

  Contains starting location and size of file

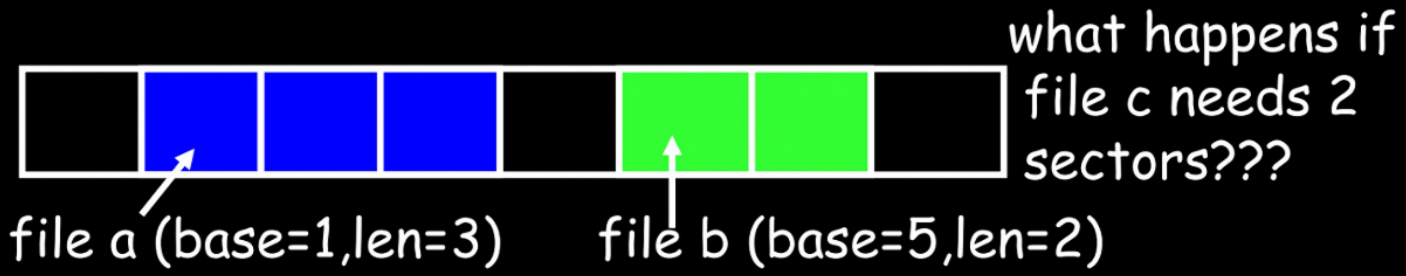




file a (base=1,len=3)

file b (base=5,len=2)

inode has this


what happens if
file c needs 2
sectors???
file a (base=1,len=3)
file b (base=5,len=2)

Pros and cons

- Pros
  - Easy to implement
  - Low storage overhead (two variables to specify disk area for file)
  - Fast sequential access since data stored in continuous blocks
  - Fast to compute data location for random addresses. Just an array index

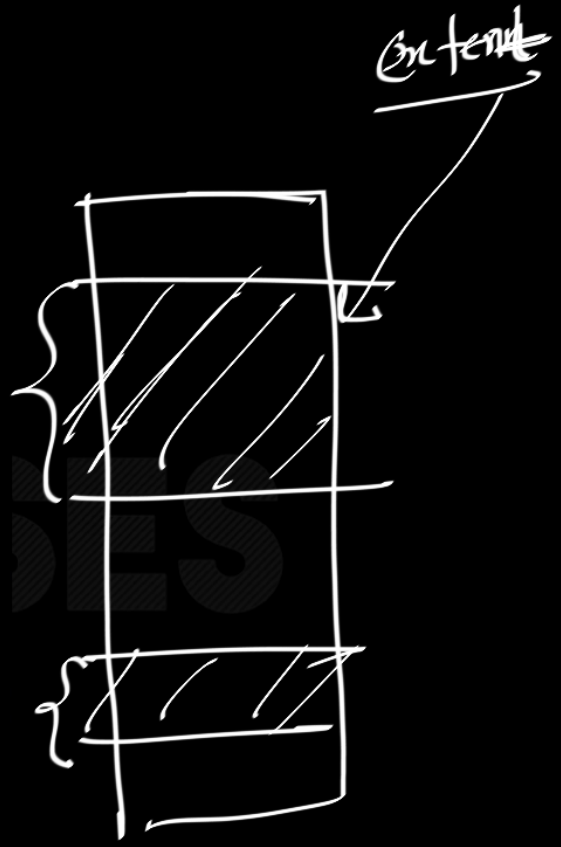
Start location and length

- Cons
  - Large external fragmentation
  - Difficult to grow file

Small internal fragmentation

Extent-based allocation

- Multiple contiguous regions per file (like segmentation)
  - Each region is an extent
  - Metadata: contains small array of entries designating extents
    - Each entry: start and size of extent



all extents info

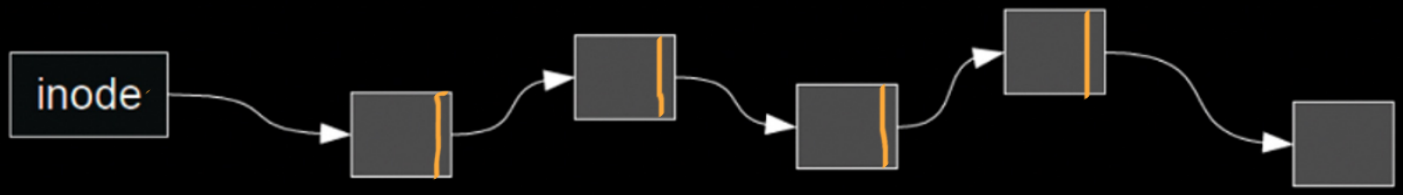
Pros and cons

- Pros
  - Easy to implement
  - Low storage overhead (a few entries to specify file blocks)
  - File can grow overtime (until run out of extents)
  - Fast sequential access
  - Simple to calculate random addresses
- Cons
  - Help with external fragmentation, but still a problem

Linked allocation

- All blocks of a file on linked list
  - Each block has a pointer to next

- Metadata (inode): pointer to the first block

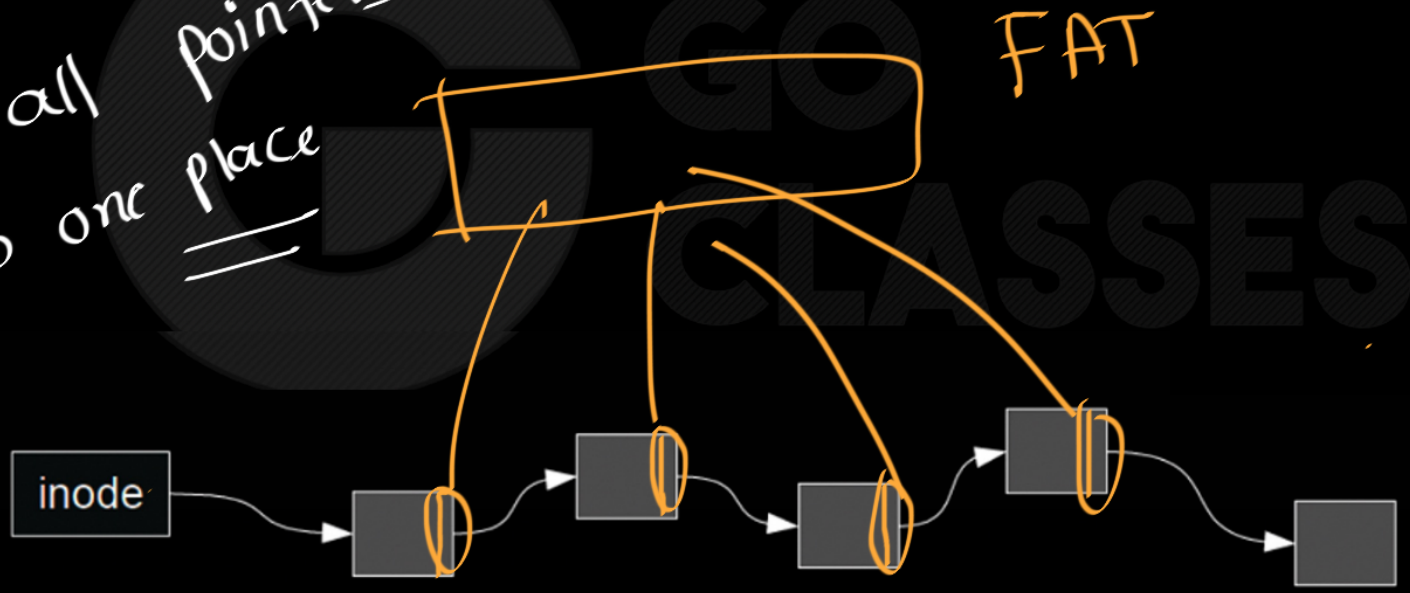


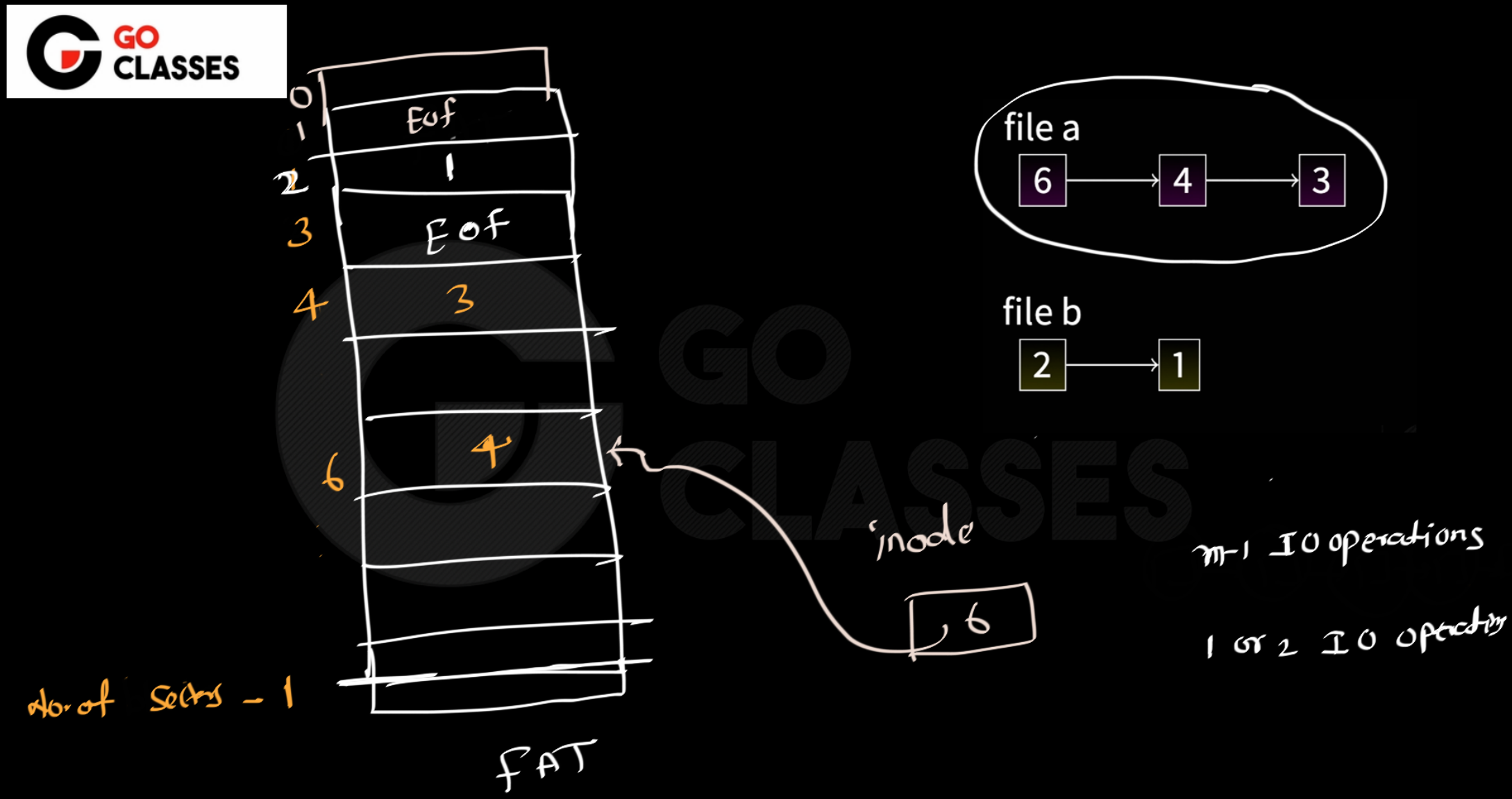
Pros and cons

- Pros
  - No external fragmentation
  - Files can be easily grown with no limit
  - Also easy to implement, though awkward to spare space for disk pointer per block
- Cons
  - Large storage overhead (one pointer per block)
  - Potentially slow sequential access
  - Difficult to compute random addresses

Variation: FAT table ← imp

bring all pointers to one place

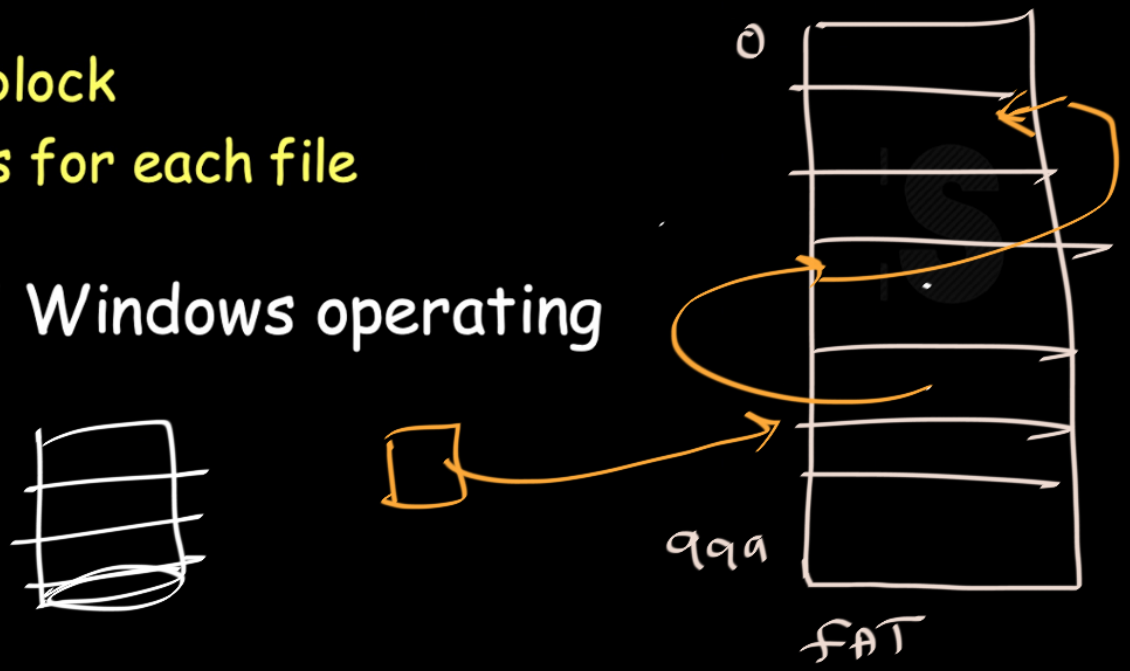


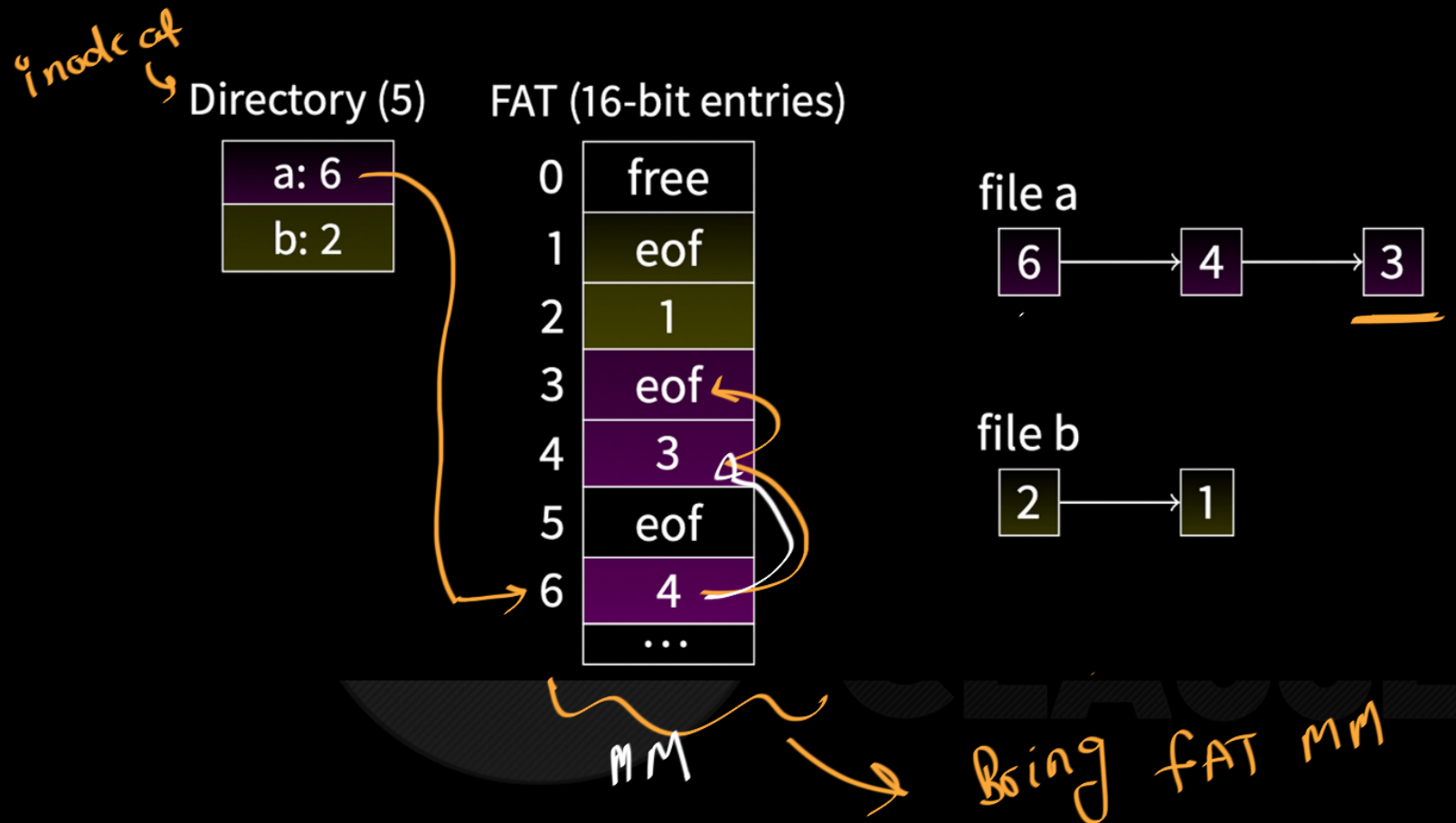

0
1
2
3
4
6
Eof
1
Eof
3
4
FAT
no. of Sectors - 1
file a
6
4
3
file b
2
1
inode
6
m-1 IO operations
1 or 2 IO operations

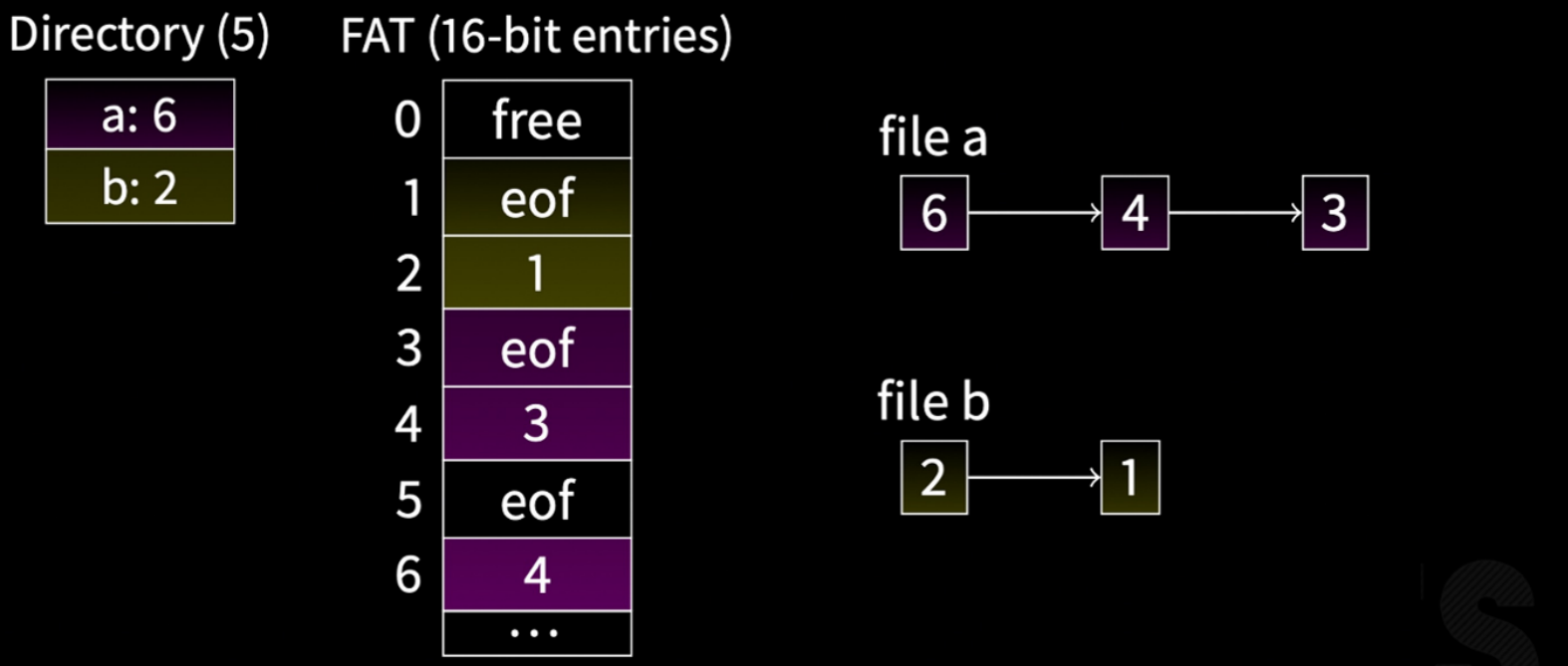
Variation: FAT table

imp

- Store linked-list pointers outside block in File-Allocation Table
  - One entry for each block
  - Linked-list of entries for each file
- Used in MSDOS and Windows operating systems




inode of
Directory (5)
a: 6
b: 2
FAT (16-bit entries)
0 free
1 eof
2 1
3 eof
4 3
5 eof
6 4
...
file a
6 → 4 → 3
file b
2 → 1
MM
Bring FAT MM
Still do pointer chasing



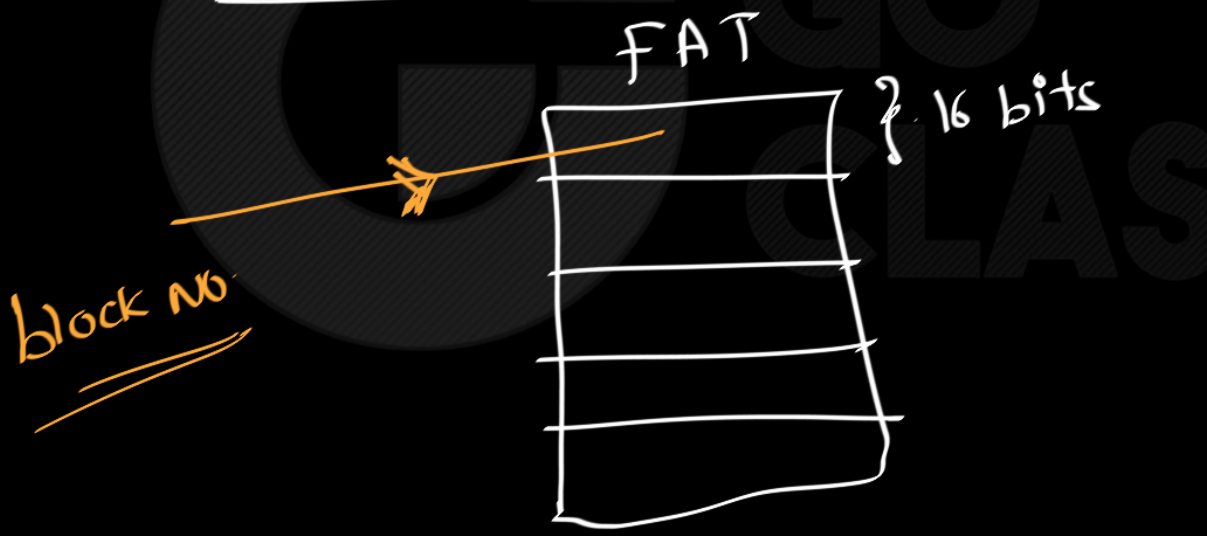
Still do pointer chasing, but can bring entire FAT in MM so can be cheap compared to disk access.

- **Entry size = 16 bits**
  - What's the maximum size of the FAT?
  - Given a 512 byte block, what's the maximum size of FS?

- **Entry size = 16 bits**
  - What's the maximum size of the FAT?
  - Given a 512 byte block, what's the maximum size of FS?

2^{16} blocks

$2^{16} \times 512$



- **Entry size = 16 bits**
  - What's the maximum size of the FAT? 65,536 entries
  - Given a 512 byte block, what's the maximum size of FS? 32 MB

2^{16} entries

$2^{16} \times 512$
$= 2^9 \times 2^{16}$
$= 2^5 \times 2^{20}$
$= 32$ MB

2^{16} blocks $\Rightarrow$ one entry for each block

An important variation on linked allocation is the use of a **file-allocation table** (**FAT**). This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 12.7 for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

GALVIN

Pros and cons

- Pros
  - Fast random access. Only search cached FAT
- Cons
  - Large storage overhead for FAT table
  - Potentially slow sequential access

12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

block

https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

block size = M bytes

No. of blocks = 2^N

https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

Ans: $2^N * M$ bytes

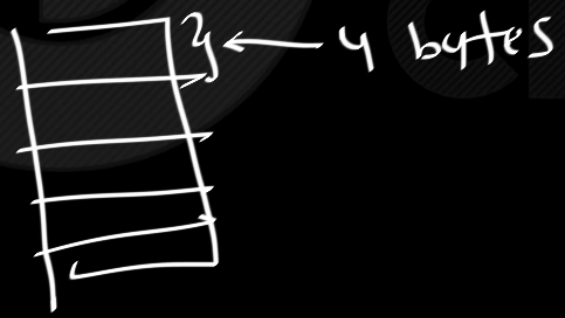
GATE CSE 2014 Set 2 | Question: 20

asked in **Operating System** Sep 28, 2014 • edited Jun 26, 2018 by **kenzou** 15,059 views

47

A FAT (file allocation table) based file system is being used and the total overhead of each entry in the FAT is 4 bytes in size. Given a 100×10^6 bytes disk on which the file system is stored and data block size is 10^3 bytes, the maximum size of a file that can be stored on this disk in units of 10^6 bytes is _____.

gatecse-2014-set2 operating-system disk numerical-answers normal file-system



https://gateoverflow.in/1977/gate-cse-2014-set-2-question-20

GATE CSE 2014 Set 2 | Question: 20

asked in **Operating System** Sep 28, 2014 • edited Jun 26, 2018 by **kenzou**

15,059 views

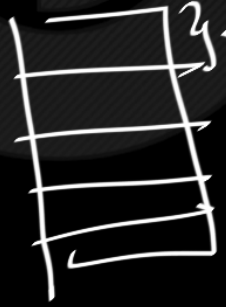
47

A FAT (file allocation table) based file system is being used and the total overhead of each entry in the FAT is 4 bytes in size. Given a 100×10^6 bytes disk on which the file system is stored and data block size is 10^3 bytes, the maximum size of a file that can be stored on this disk in units of 10^6 bytes is _____.

gatecse-2014-set2 | operating-system | disk | numerical-answers | normal | file-system

in bytes $\frac{4x}{10^6}$ } ← final answer

$\frac{100^8}{10^3}$



} ← 4 bytes

{ No. of blocks = 10^5

No. of entries

block size = $\frac{10^3}{}$ bytes

$4 \times 10^5 = 0.4 \times 10^6$

https://gateoverflow.in/1977/gate-cse-2014-set-2-question-20

GO CLASSES

No. of entries = 10^5

one entry size = 4 bytes

Size of FAT = 4×10^5 bytes

total disk size = 100×10^6 bytes = $10^3 \times 10^5$ bytes

Space left to store file = $10^3 \times 10^5 - 4 \times 10^5$

$= 996 \times 10^5$ bytes

in units of 10^6 ⇒ $\frac{}{10^6}$ $= 99.6$ MB

GO CLASSES

93

Each datablock will have its entry.

So, Total Number of entries in the FAT $= \frac{\text{Disk Capacity}}{\text{Block size}} = \frac{100MB}{1KB} = 100K$

Each entry takes up $4B$ as overhead

So, space occupied by overhead = $100K \times 4B = 400KB = 0.4MB$

Best answer

We have to give space to Overheads on the same file system and at the rest available space we can store data.

So, assuming that we use all available storage space to store a single file = Maximum file size =
$\text{Total File System size} - \text{Overhead} = 100MB - 0.4MB = 99.6MB$

answered Jan 18, 2015 • edited Jun 26, 2018 by **Milicevic3306**

comment Follow share this

kalpish

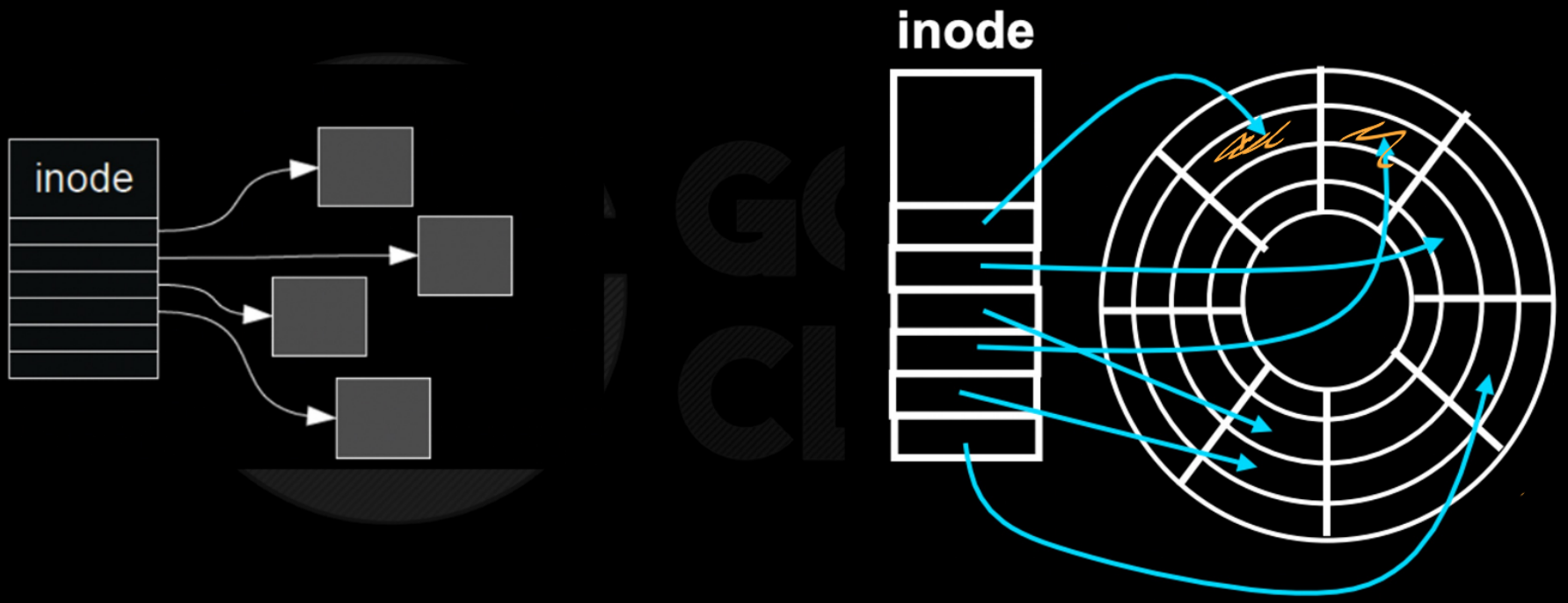
Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.

However, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

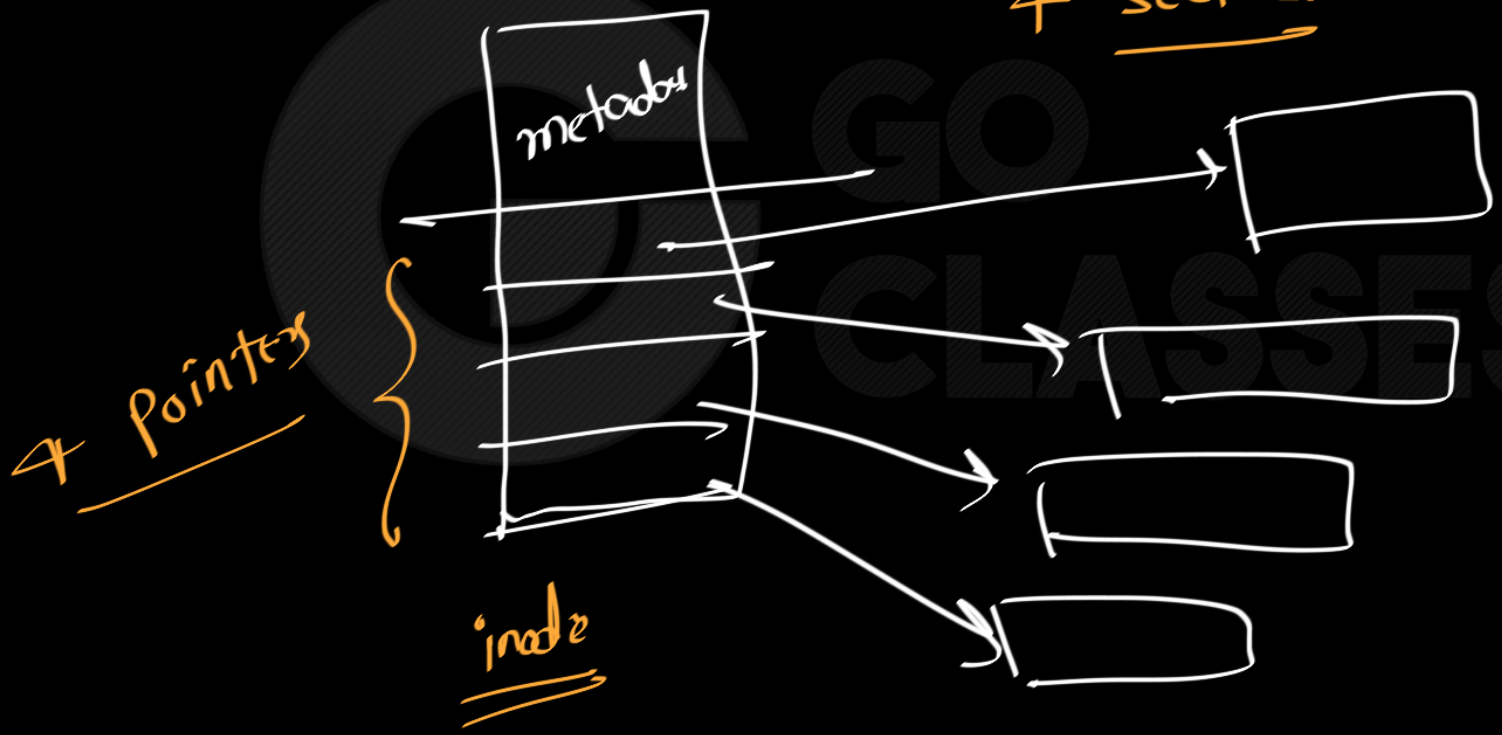
GAIVIN

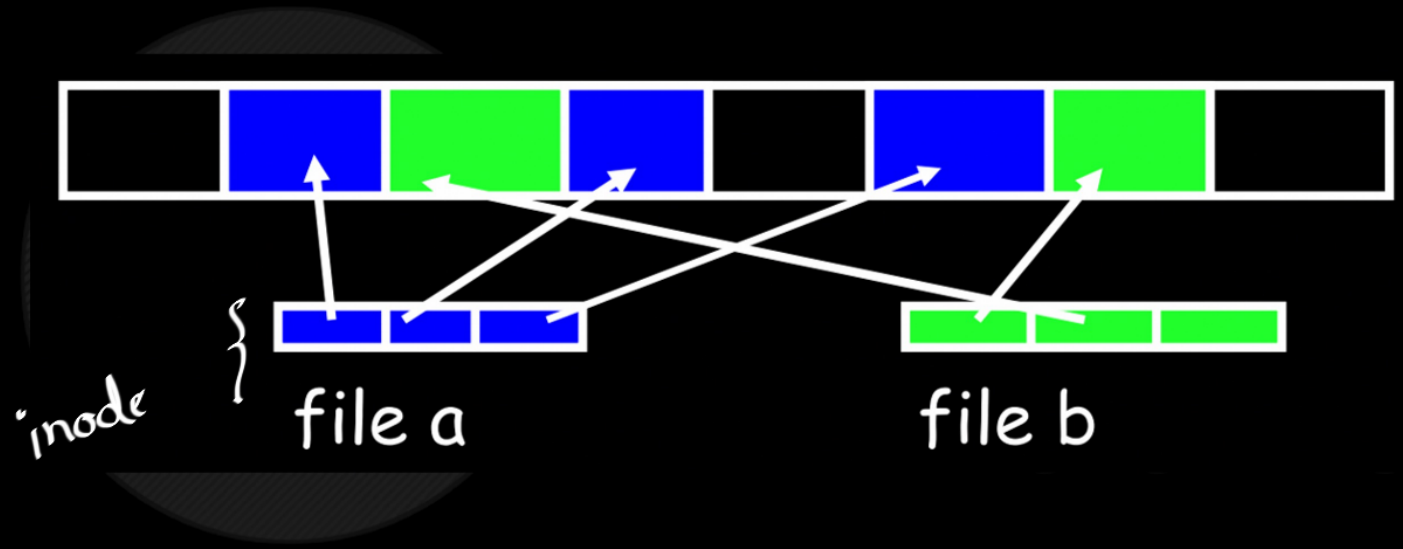
Indexed Allocation

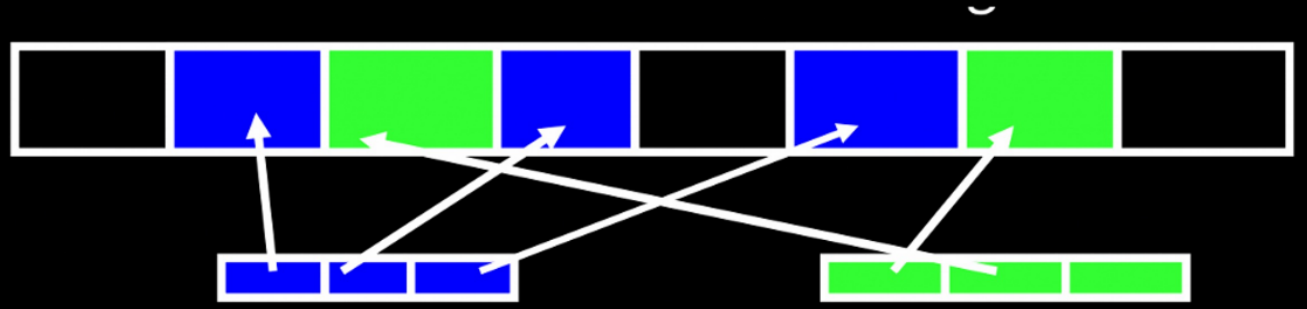


GO CLASSES

if you have space for 4 pointers in inode then you can use atmost 4 sectors.




inode
file a
file b



- **Pros?**
  - Both sequential and random access easy
- **Cons?**
  - Mapping table requires large chunk of contiguous space
    …Same problem we were trying to solve initially

inode size is Big.

GO CLASSES

Small inode

max. file size is less

Big inode

inode size itself is big

max file size is large enough

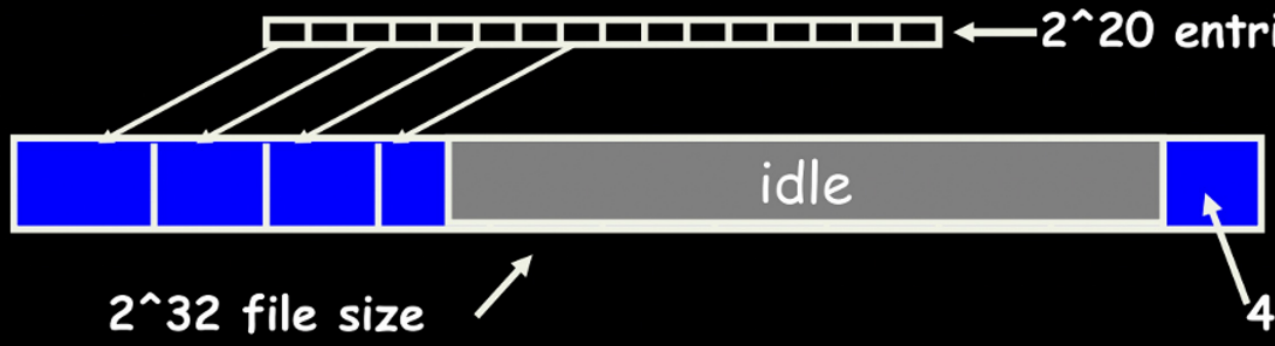

10
entries
10
entries
later file grows

GO CLASSES

Can we have a method by which we can have small inode size still large max file size?

→ Yes

- Issues same as in page tables



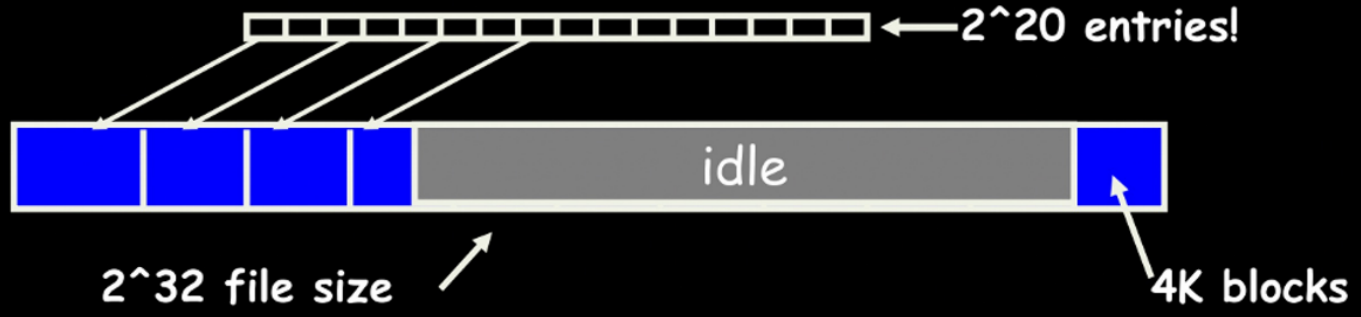
big inode

useless for most of the files

- Large possible file size = lots of unused entries
- Large actual size? table needs large contiguous disk chunk

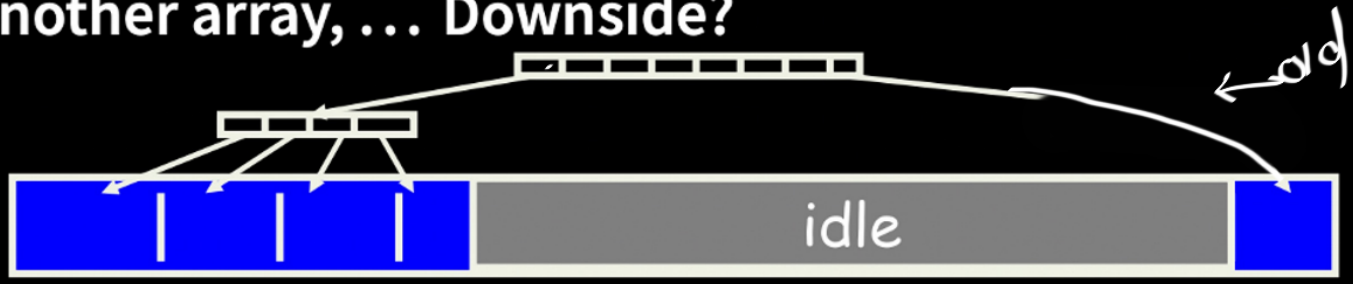
most of the files are small

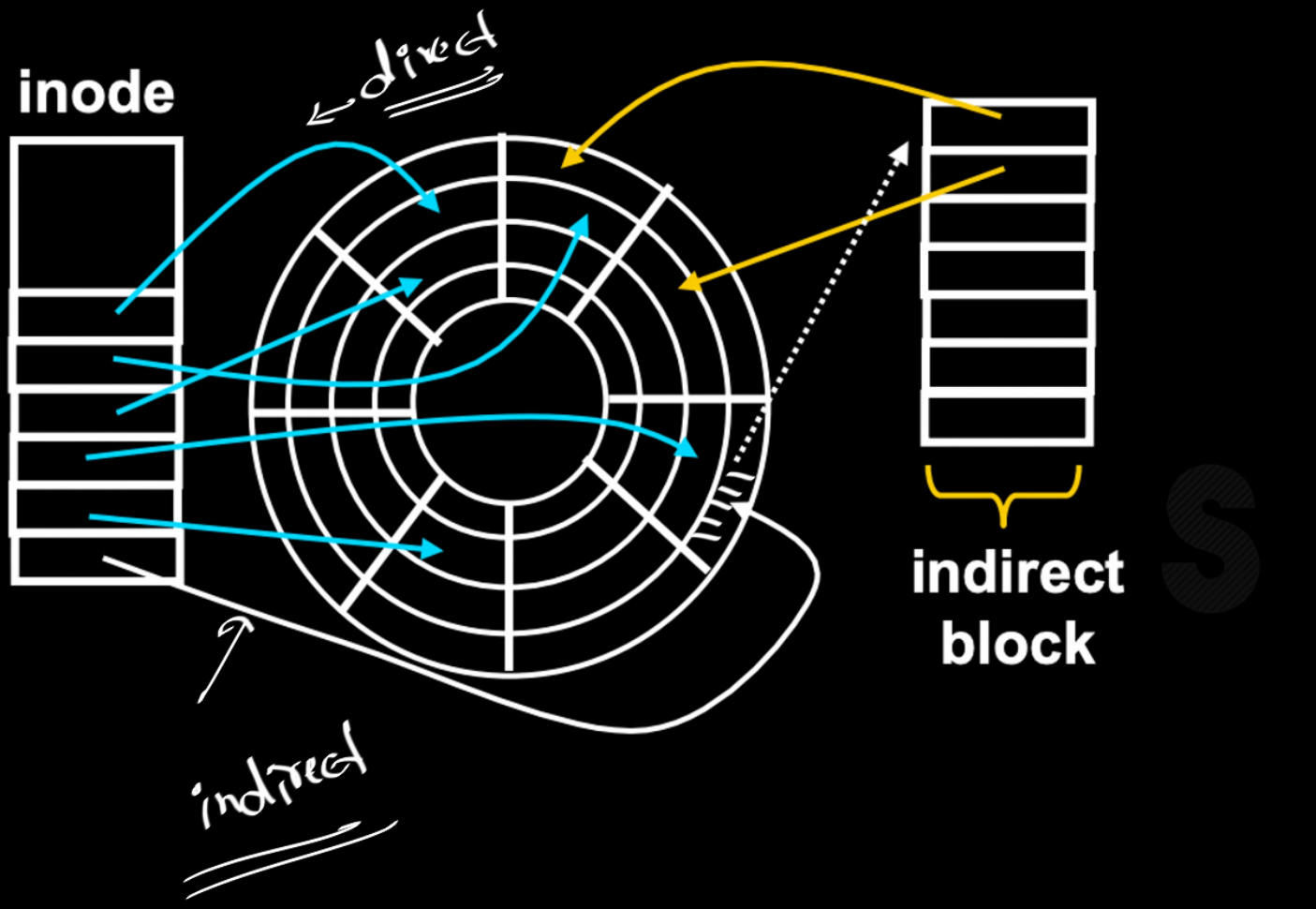
- **Issues same as in page tables**



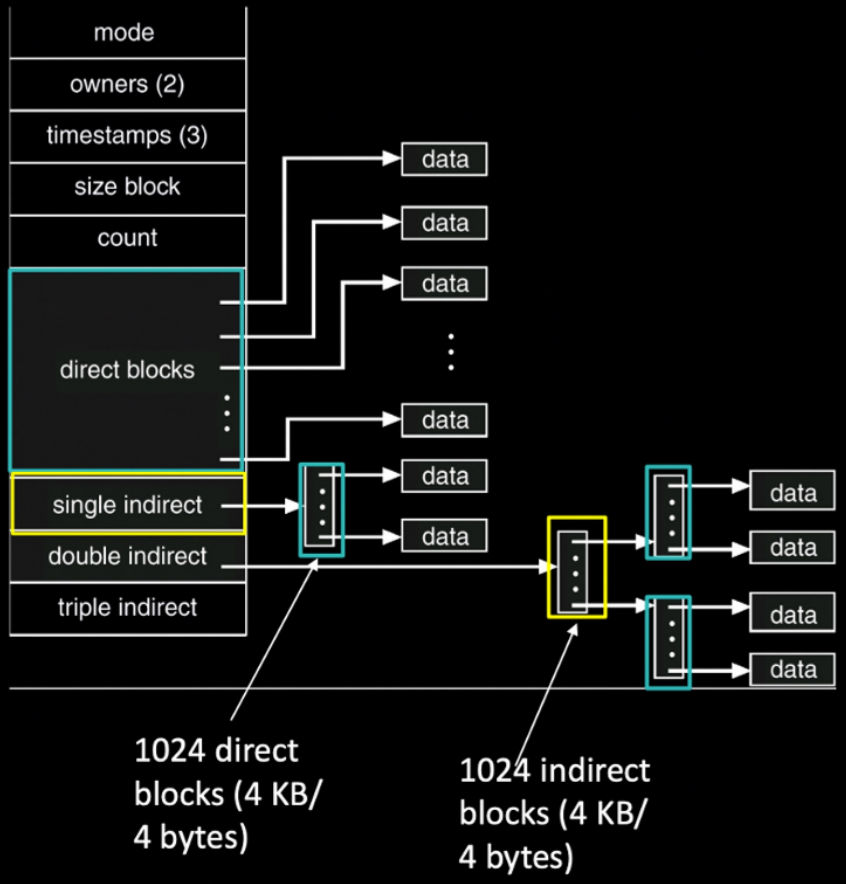
  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk

- **Solve identically: small regions with index array, this array with another array, . . . Downside?**

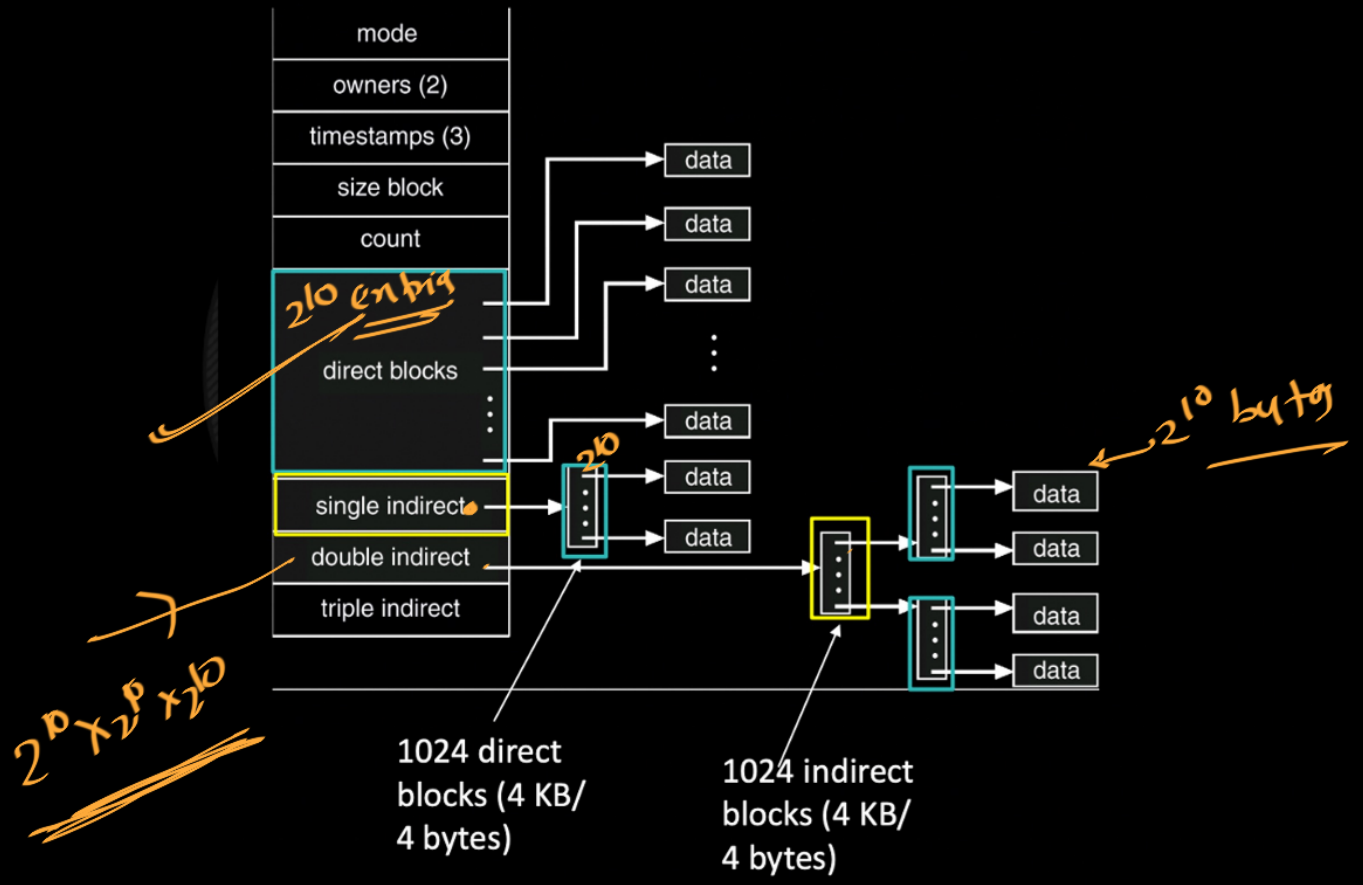



inode
direct
indirect
indirect block

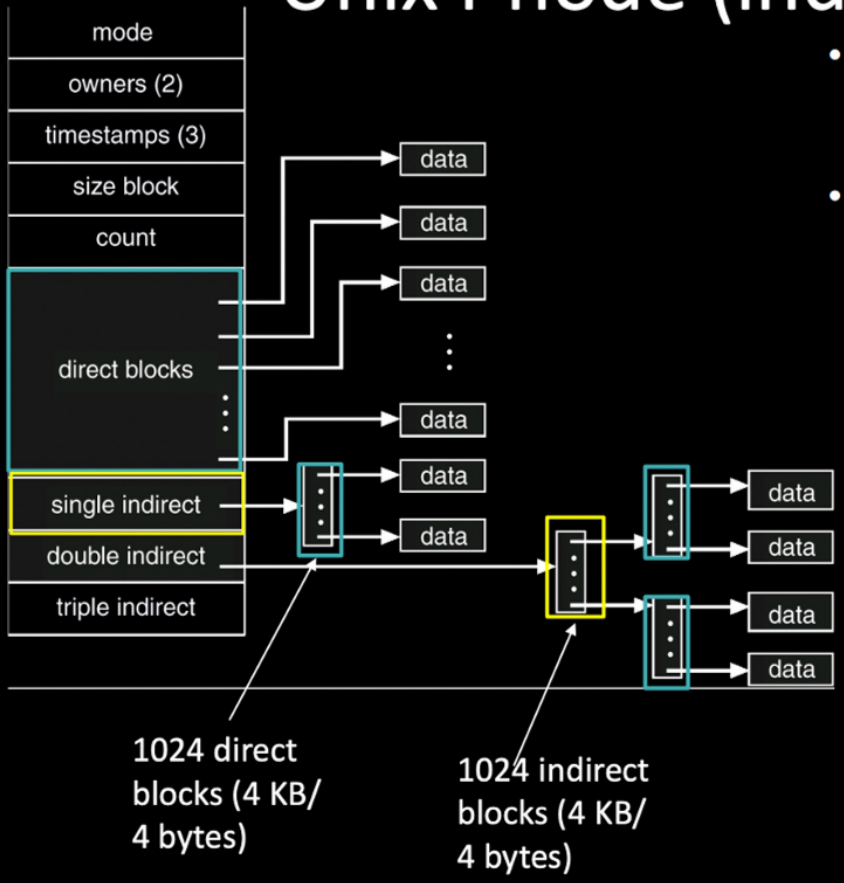
Unix i-node (index node)



Unix i-node (index node)



Unix i-node (index node)



- Small files can be accessed quickly
- If each block is 4KB, each block address is 4 bytes
  - First 48KB of file reachable from 12 direct blocks
  - Next 4MB available from single-indirect blocks
  - Next 4GB available from double-indirect blocks
  - Next 4TB available through the triple-indirect blocks
  - To access any one block of a 4TB file, how many disk accesses at most do we need?

4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

   (a) How large of a disk can this file system support?

   (b) What is the maximum file size?

https://inst.eecs.berkeley.edu/~cs162/fa19/static/sections/section10-sol.pdf

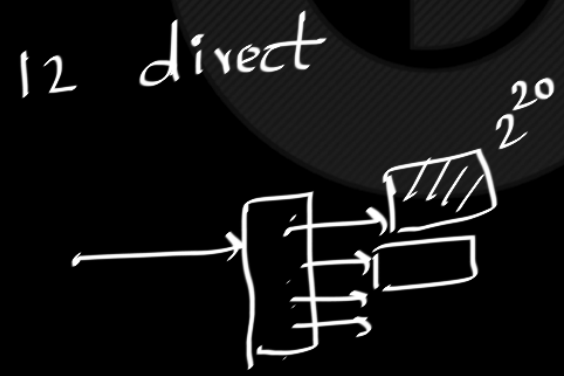
4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

   (a) How large of a disk can this file system support?

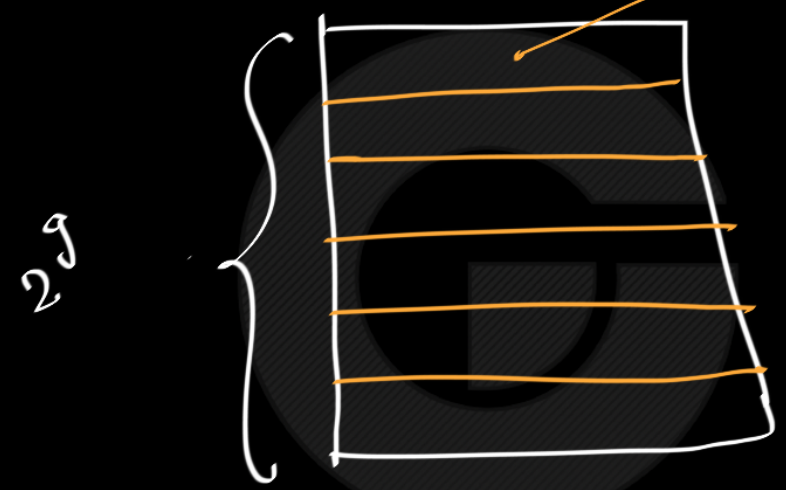
   block size = 2^{11} B

   2^{32} blocks disk

   (b) What is the maximum file size?



https://inst.eecs.berkeley.edu/~cs162/fa19/static/sections/section10-sol.pdf



block addrss = 32 bits

what are no. of bits we require to address one block?

= 32 bits = 4 B

each block can hold = $\frac{2^{11}}{2^2} = 2^9$ entries

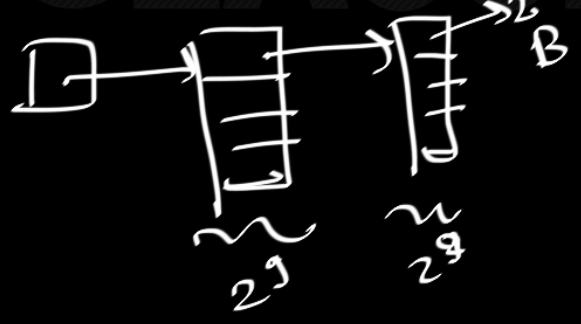
4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

(a) How large of a disk can this file system support?

block size = 2^{11} B

= 2^9 entries

(b) What is the maximum file size?



Singly indirect

$2^9 \times 2^{11}$

$$\left[12 \times \underline{\underline{2^{11}}} \quad + \quad 1 \times 2^9 \times 2^{11} \quad + \quad 1 \times 2^9 \times 2^9 \times 2^{11} \quad + \right.$$

$$\left. 1 * \; 2^9 \times 2^9 \times 2^9 \times 2^{11} \right]$$

4. Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

2^{32} Blocks

(a) How large of a disk can this file system support?

2^{32} blocks x 2^{11} bytes/block $= 2^{43} = 8$ Terabytes.

(b) What is the maximum file size?

There are 512 pointers per block (i.e. 512 4-byte pointers in 2048 byte block), so:
blockSize $\times$ (numDirect + numIndirect + numDoublyIndirect + numTriplyIndirect)

$$\begin{aligned} 2048 \times (12 + 512 + 512^2 + 512^3) &= 2^{11} \times (2^2 \times 3 + 2^9 + 2^{9\times 2} + 2^{9\times 3}) \\ &= 2^{13} \times 3 + 2^{20} + 2^{29} + 2^{38} \\ &= 24K + 513M + 256G \end{aligned}$$

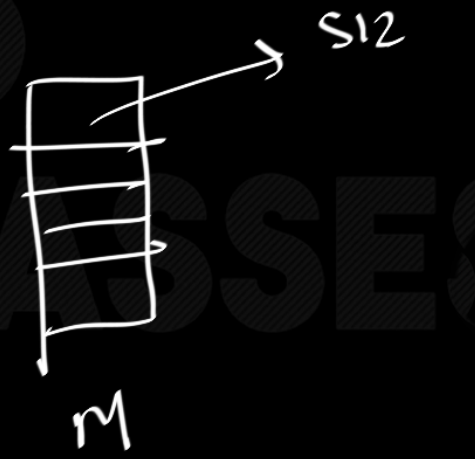
11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

$512 \times N + M \times 512$

No. of entries $= M$



https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

Ans: (N+M)*512 bytes

13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

$1TB = 2^{40} B$

2^{41} Bytes $\leftarrow$ 2TB

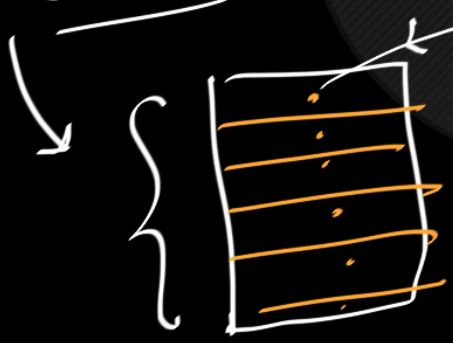
https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

$1\,TB = 2^{40}\,B$

2^{41} Bytes $\leftarrow$ 2TB

64 Bytes



$$\frac{2^{41}}{2^{9}} = 2^{32}$$

no. of blocks

https://www.cse.iitb.ac.in/~mythili/os/ps/file/ps-file.pdf

GO CLASSES

Block size = 2^9 B

entry size = 32 bits = 2^2 bytes

No. of entries = 2^7

each double can point to this much data

No. of pointers $2^7 \times 2^7 \times 2^9$ ←

in 64 Bytes, we can store # of pointers = $\frac{64\,B}{4\,B} = 16$

$16 \times 2^7 \times 2^7 \times 2^9$ ← Answer

$= 2^7$ B

13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

16 ⟶ 1 double + 1 single + 14 direct

Ans: Number of data blocks = $2^{41}/2^{9} = 2^{32}$, so 32 bits or 4 bytes are required to store the number of a data block.

Number of data block pointers in the inode = 64/4 = 16, of which 14 are direct blocks. The single indirect block stores pointers to 512/4 = 128 data blocks. The double indirect block points to 128 single indirect blocks, which in turn point to 128 data blocks each.

3. (28 points total) File Systems.

a. (8 points) Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

i) (4 points) How large of a disk can this file system support?

H.W.

ii) (4 points) What is the maximum file size?

http://www.cs.cornell.edu/courses/cs4410/2009sp/oldexams/midterm2-solutions.pdf

3. (28 points total) File Systems.

a. (8 points) Consider a file system with 2048 byte blocks and 32-bit disk and file block pointers. Each file has 12 direct pointers, a singly-indirect pointer, a doubly-indirect pointer, and a triply-indirect pointer.

i) (4 points) How large of a disk can this file system support?

2^{32} blocks x 2^{11} bytes/block = 2^{43} = 8 Terabytes.

-3 missing block or disk size in calculation

ii) (4 points) What is the maximum file size?

There are 512 pointers per block (i.e. 512 4-byte pointers in 2048 byte block), so:
blockSize x (numDirect + numIndirect + numDoubly-indirecty+numTriply indirect)
2048 x (12 + 512 + 512^2 + 512^3) = 2^{11} x (2^2 x 3+ 2^9 + 2^{9x2} + 2^{9x3})
= 2^{13} x 3+ 2^{20} + 2^{29} + 2^{38}
= 24K + 513M + 256 G

Question

b. (15 points) Large files. We have seen different filesystems that support fairly large files. Now let's see just how large a file various types of filesystems can support. Assume, for all of the questions in this part, that filesystem blocks are 4 KBytes. *Show your solutions in unsimplified form for partial credit.*

BS = 4KB

i) (3 points) Consider a really simple filesystem, **directfs**, where each inode only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). What is the maximum file size for **directfs**?

ii) (3 points) Consider a filesystem, called **extentfs**, with a construct called an extent. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly one extent. What is the maximum file size for **extentfs**?

iii) (3 points) Consider a filesystem that uses direct pointers, but also adds indirect pointers and double-indirect pointers. We call this filesystem, **indirectfs**. Specifically, an inode within **indirectfs** has 1 direct pointer, 1 indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for **indirectfs**?

Question

b. (15 points) Large files. We have seen different filesystems that support fairly large files. Now let's see just how large a file various types of filesystems can support. Assume, for all of the questions in this part, that filesystem blocks are 4 KBytes. *Show your solutions in unsimplified form for partial credit.*

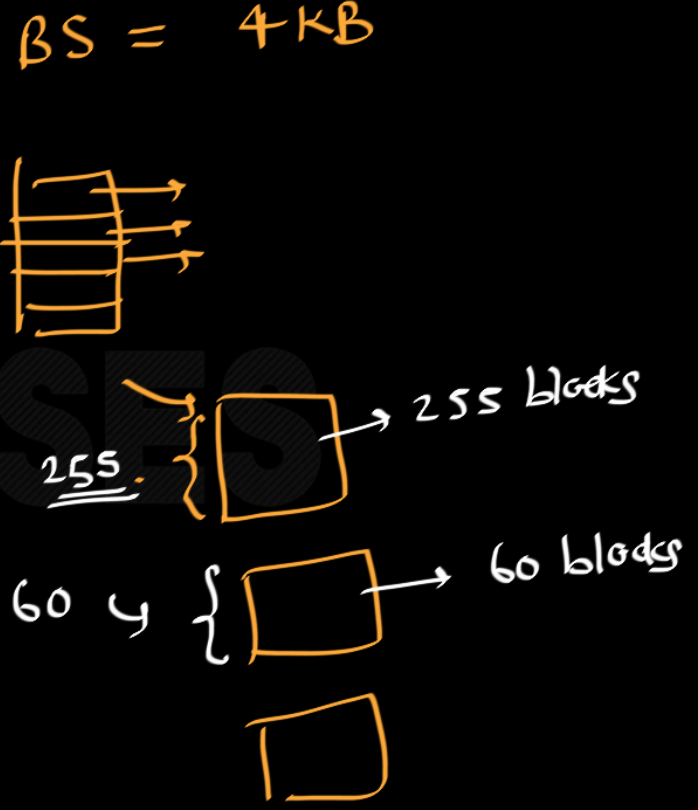
i) (3 points) Consider a really simple filesystem, **directfs**, where each inode only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). What is the maximum file size for **directfs**?

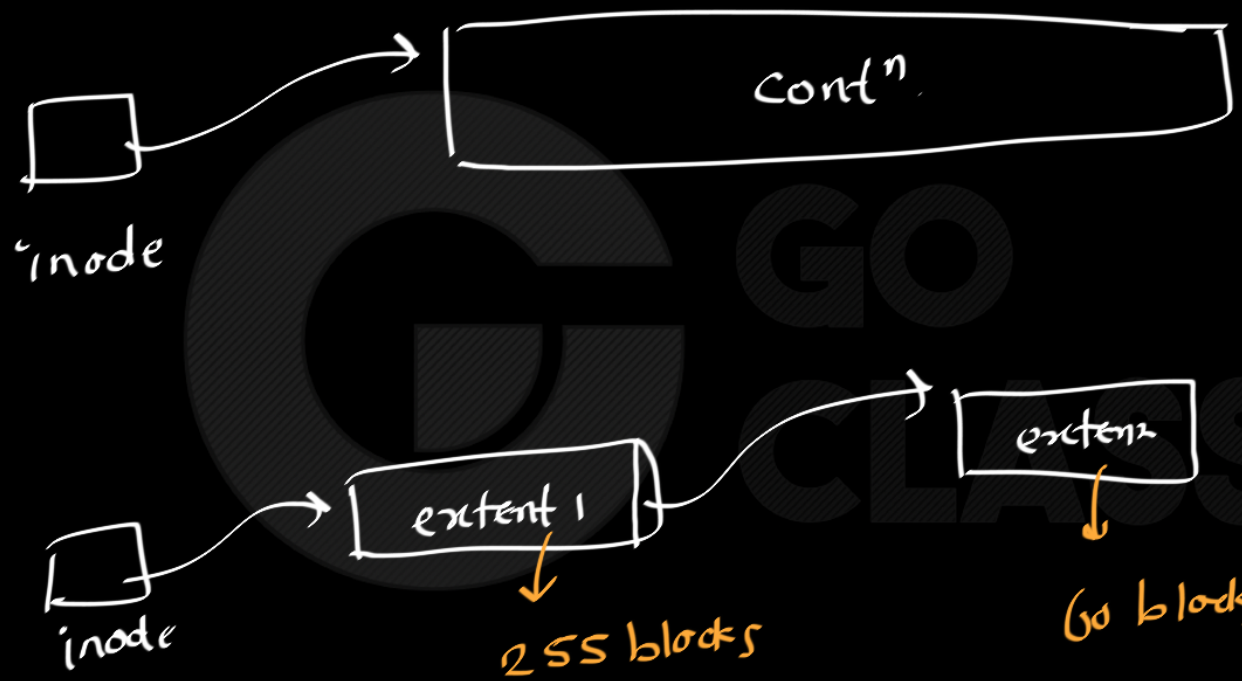
4 X10 = 40 kB

ii) (3 points) Consider a filesystem, called **extentfs**, with a construct called an extent. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly one extent. What is the maximum file size for **extentfs**?

255 X4kB

iii) (3 points) Consider a filesystem that uses direct pointers, but also adds indirect pointers and double-indirect pointers. We call this filesystem, **indirectfs**. Specifically, an inode within **indirectfs** has 1 direct pointer, 1 indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for **indirectfs**?



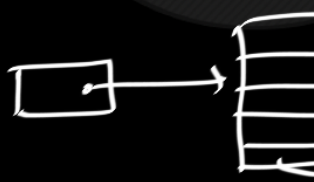


max size of any extent = 255 blocks = 255 x 4KB

iii) (3 points) Consider a filesystem that uses direct pointers, but also adds indirect pointers and double-indirect pointers. We call this filesystem, **indirectfs**. Specifically, an inode within **indirectfs** has 1 direct pointer, 1 indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for **indirectfs**?

1 block + 2^{10} blocks + $2^{10} \times 2^{10}$ blocks

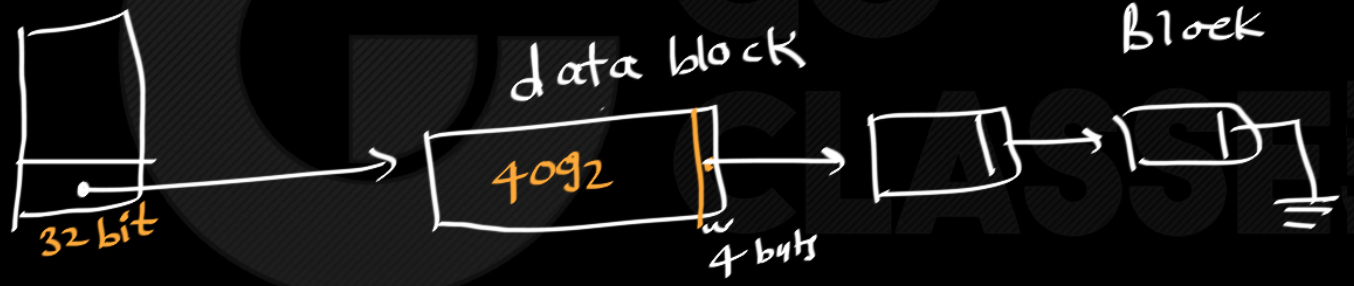
$= \left(1 + 2^{10} + 2^{10} \times 2^{10}\right)$ blocks



$= 2^{10}$ entries

$= \left(1 + 2^{10} + 2^{20}\right) \times 4 \times 2^{10}$ B

iv) (3 points) Consider a compact file system, called **compactfs**, tries to save as much space as possible within the inode. Thus, to point to files, it stores only a single 32-bit pointer to the first block of the file. However, blocks within **compactfs** store 4,092 bytes of user data and a 32-bit next field (much like a linked list), and thus can point to a subsequent block (or to NULL, indicating there is no more data). First, draw a picture of an inode and a file that is 10 KBytes in size.



No. of data blocks = 2^{32}

Block size = 2^{12} (= 4096)

4092 + 4

v) (3 points) What is the maximum file size for **compactfs** (assuming no other restrictions on file sizes)?

Disk size = 2^{44} B

No. of blocks = 2^{32}

GO CLASSES

No. of blocks = 2^{32}

No. of pointers = 2^{32}

No. of bytes wasted = $2^{32} \times 4$ bytes

= 2^{34} B → wasted in pointers

max. file size = $2^{44} - 2^{34}$

Solution

b. (15 points) Large files. We have seen different filesystems that support fairly large files. Now let's see just how large a file various types of filesystems can support. Assume, for all of the questions in this part, that filesystem blocks are 4 KBytes. *Show your solutions in unsimplified form for partial credit.*

i) (3 points) Consider a really simple filesystem, **directfs**, where each inode only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). What is the maximum file size for **directfs**?

*The maximum **directfs** file size is 10 x 4 KByte = 40 KByte*

ii) (3 points) Consider a filesystem, called **extentfs**, with a construct called an extent. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly one extent. What is the maximum file size for **extentfs**?

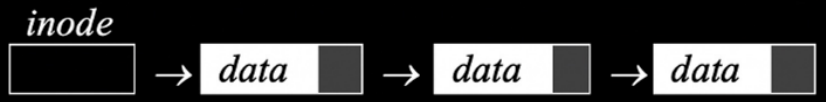
*The maximum **extentfs** file size is $(2^8 - 1)$ x 4 KByte = 255 x 4 KByte = 1 MByte*

https://cs162.org/static/exams/fa16-mt3-solutions.pdf

iii) (3 points) Consider a filesystem that uses direct pointers, but also adds indirect pointers and double-indirect pointers. We call this filesystem, **indirectfs**. Specifically, an inode within **indirectfs** has 1 direct pointer, 1 indirect pointer, and 1 doubly-indirect pointer field. Pointers, as before, are 4 bytes (32 bits) in size. What is the maximum file size for **indirectfs**?

*The maximum **indirectfs** file size is (1 + 1024 + (1024 x 1024)) x 4 KByte = 4 GB + 4100 KByte*

iv) (3 points) Consider a compact file system, called **compactfs**, tries to save as much space as possible within the inode. Thus, to point to files, it stores only a single 32-bit pointer to the first block of the file. However, blocks within **compactfs** store 4,092 bytes of user data and a 32-bit next field (much like a linked list), and thus can point to a subsequent block (or to NULL, indicating there is no more data). First, draw a picture of an inode and a file that is 10 KBytes in size.



v) (3 points) What is the maximum file size for **compactfs** (assuming no other restrictions on file sizes)?

*The maximum **compactfs** file size is* 2^{32} *x 4 KByte –* 2^{32} *x 4 byte = 16 TByte*

$2^{44} - 2^{34}$

Problem 3d[4pts]: Consider a file system with 4096 byte blocks and 32-bit disk and file block pointers. Each file has 13 direct pointers, 4 singly-indirect pointers, a doubly-indirect pointer, and a triply-indirect pointer. In the following be explicit about your work:

a) What is the maximum disk size that can be supported? Explain.

b) What is the maximum file size? Explain.

c) Give some reasonable assumptions and compute the number of **inodes** that can fit into a disk block.

Problem 3d[4pts]: Consider a file system with 4096 byte blocks and 32-bit disk and file block pointers. Each file has 13 direct pointers, 4 singly-indirect pointers, a doubly-indirect pointer, and a triply-indirect pointer. In the following be explicit about your work:

a) What is the maximum disk size that can be supported? Explain.

$2^{32} \times 2^{12}$

BS = 4096 B

No. of blocks = 2^{32}

b) What is the maximum file size? Explain.

$13 + 4 \times 2^{10} + 1 \times 2^{10} \times 2^{10} + 2^{10} \times 2^{10} \times 2^{10}$

No. of entries in one block = $\frac{4096\ B}{32\ bits} = 2^{10}$

$= 2^7$

$= 128\ B$

c) Give some reasonable assumptions and compute the number of **inodes** that can fit into a disk block.

No. of inodes

inode has 19 pointers

19 × 4 B = 76 B

pointers

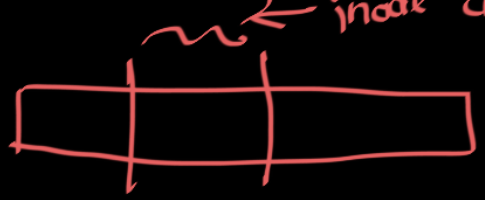
GO CLASSES

Disk block size = 2^{12} B

inode size = 76 B in pointers + something extra

= 128 B (assume)

No. of inodes in disk block = $\frac{2^{12}}{2^7} = 2^5 = 32$ inodes



Problem 3d[4pts]: Consider a file system with 4096 byte blocks and 32-bit disk and file block pointers. Each file has 13 direct pointers, 4 singly-indirect pointers, a doubly-indirect pointer, and a triply-indirect pointer. In the following be explicit about your work:

a) What is the maximum disk size that can be supported? Explain.

Since block pointers are 32 bits, total size = $2^{32} \times 4096 = 2^{44}$ bytes maximum.

b) What is the maximum file size? Explain.

Since pointers are 32 bits (4 bytes), one indirect block can address 1024 blocks. Thus, maximum file= $4096 \times (13 + 4 \times 1024 + 1024^2 + 1024^3) = 4402358308864$ bytes

c) Give some reasonable assumptions and compute the number of **inodes** that can fit into a disk block.

An inode has 13 + 4 + 1 + 1=19 pointers = 76bytes. Assuming various other file information such as ACLs, etc, we round up to 128bytes and can fit 32 inodes/block

https://people.eecs.berkeley.edu/~kubitron/courses/cs162-F10/exams/fa06mt2-solutions.pdf

In lecture three file descriptor structures were discussed: (a) Indexed files. (b) Linked files. (c) Contiguous (extent-based) allocation.

Each of the structures has its advantages and disadvantages depending on the goals for the file system and the expected file access pattern. For each of the following situations, rank the three structures in order of preference. Be sure to include the justification for your rankings.

(a) You have a file system where the most important criteria is the performance of sequential access to very large files.

(b) You have a file system where the most important criteria is the performance of random access to very large files.

https://people.eecs.berkeley.edu/~kubitron/courses/cs162-F15/fa15/static/sections/section10.pdf

(a) You have a file system where the most important criteria is the performance of sequential access to very large files.

1. c (extent-based)
2. b (linked)
3. a (indexed)

It is easy to see that (c) is the best structure for sequential access to very large files, since in (c) files are contiguously allocated and the next block to read is physically the next on the disk. No seek time to find the next block, and each block will be read sequentially as the disk head moves.

Both (b) and (a) require some look up operation in order to know where the next block is. However, (b) may be slightly more expensive, since for "very large files", multiple disk accesses are required to read the indirect blocks.

(b) You have a file system where the most important criteria is the performance of random access to very large files.

1. c (extent-based)
2. a (indexed)
3. b (linked)

(c) is still the best structure here: just need to use an offset.

(a) will probably need to look at some levels of indirect blocks in order to find the right block to access (we are dealing with very large files).

(b) is absolutely the worst structure. In fact, in order to find a random block, we will need to traverse a linked list of blocks, which will take a time linear in the offset size.

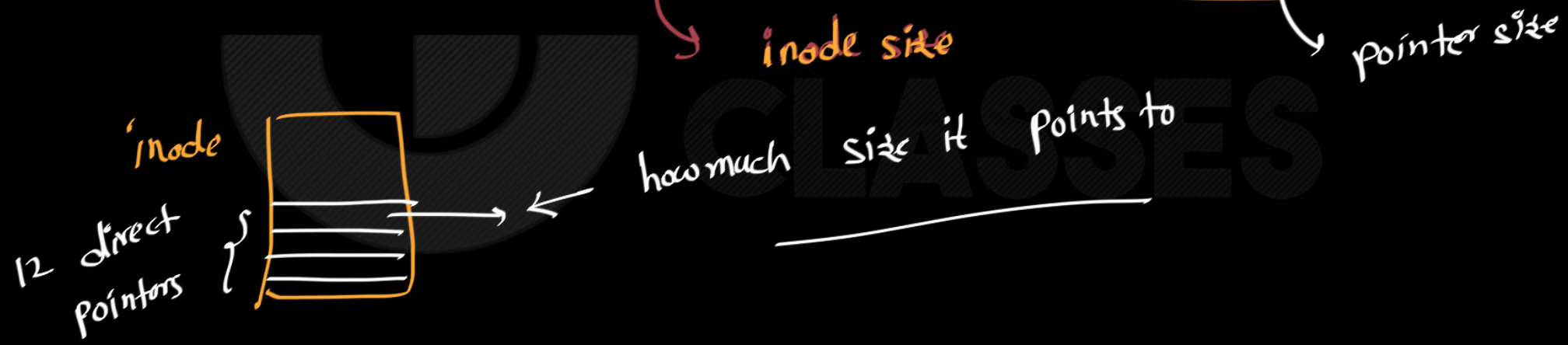
https://people.eecs.berkeley.edu/~kubitron/courses/cs162-F15/fa15/static/sections/section10soln.pdf

GATE CSE 2002 | Question: 2.22

30

In the index allocation scheme of blocks to a file, the maximum possible size of the file depends on

A. the size of the blocks, and the size of the address of the blocks.
B. the number of blocks used for the index, and the size of the blocks.
C. the size of the blocks, the number of blocks used for the index, and the size of the address of the blocks.
D. None of the above



https://gateoverflow.in/852/gate-cse-2002-question-2-22

GO CLASSES

Disk Size = 2^{12} Blocks $\times$ 2^{20} B

→ Size of inode

→ [Pointer size (no. of bits used to point to one block)

↳ 12 bits

how many Pointers you can hold $\times$ BS

↓

direct

↳ 2^{20} B

final answer

file = inode + Data

Data size = Disk size − Inode size

52

Best answer

In Index allocation size of maximum file can be derived like following:

No of addressable blocks using one Index block (A) = Size of block / Size of block address } ← How many pointers can inode can hold

No of block addresses available for addressing one file (B) =

No of Maximum blocks we can use for the Index * No of addressable blocks using one Index block (A)

Size of File = B * Size of Block

So, it is clear that:

Answer is **(C)**.

A & B are incomplete.

GATE CSE 2008 | Question: 20

30

The data blocks of a very large file in the Unix file system are allocated using

A. continuous allocation
B. linked allocation
C. indexed allocation
D. an extension of indexed allocation

multi level indexing

https://gateoverflow.in/418/gate-cse-2008-question-20

GATE CSE 2008 | Question: 20

30

The data blocks of a very large file in the Unix file system are allocated using

A. continuous allocation
B. linked allocation
C. indexed allocation
D. an extension of indexed allocation

multi level indexing

https://gateoverflow.in/418/gate-cse-2008-question-20

GATE IT 2005 | Question: 63

asked in **Operating System** Nov 4, 2014 • edited Jun 23, 2018 by **kenzou**

4,630 views

28

In a computer system, four files of size 11050 bytes, 4990 bytes, 5170 bytes and 12640 bytes need to be stored. For storing these files on disk, we can use either 100 byte disk blocks or 200 byte disk blocks (but can't mix block sizes). For each block used to store a file, 4 bytes of bookkeeping information also needs to be stored on the disk. Thus, the total space used to store a file is the sum of the space taken to store the file and the space taken to store the book keeping information for the blocks allocated for storing the file. A disk block can store either bookkeeping information for a file or data from a file, but not both.

What is the total space required for storing the files using 100 byte disk blocks and 200 byte disk blocks respectively?

A. 35400 and 35800 bytes
B. 35800 and 35400 bytes
C. 35600 and 35400 bytes
D. 35400 and 35600 bytes

116 blocks (Bs = 100)

11050
4990
5170
12640

Bs = 100 B or 200 B

H.W.

https://gateoverflow.in/3824/gate-it-2005-question-63

GO CLASSES

file size = 11050 Bytes

Block size = 100 bytes

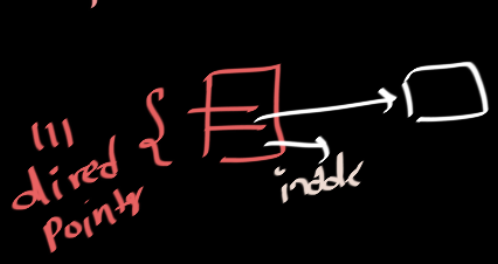
No. of blocks = $\frac{11050}{100}$ = 111 blocks

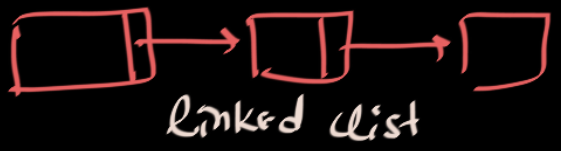
total space = 111 block + 111 × 4 Bytes

= 111 blocks + 5 blocks

= 116 blocks

for each block we need 4 bytes extra.





GO CLASSES

11050 → 116 blocks

4990 → 52 blocks

5170 → 52 + 3 = 55 blocks

12640 → 127 + 6 = 133 blocks

(ii) 4990 → 50 blocks for data

50 × 4 bytes for book keeping = 200 B = 2 blocks

52

$\frac{127 \times 4}{100}$ =

53

Best answer

for 100 bytes block:

$11050 = 111$ blocks requiring $111 \times 4 = 444$ bytes of bookkeeping info which requires another 5 disk blocks. So, totally $111 + 5 = 116$ disk blocks. Similarly,

$4990 = 50 + (50 \times 4)/100 = 52$
$5170 = 52 + (52 \times 4)/100 = 55$
$12640 = 127 + (127 \times 4/100) = 133$

$356 \times 100 = 35600$ bytes

For 200 bytes block:

$56 + (56 \times 4/200) = 58$
$25 + (25 \times 4/200) = 26$
$26 + (26 \times 4/200) = 27$
$64 + (64 \times 4/200) = 66$

$177 \times 200 = 35400$

So, (**C**) option.

GATE CSE 2017 Set 2 | Question: 08

37

In a file allocation system, which of the following allocation scheme(s) can be used if no external fragmentation is allowed?

1. Contiguous
2. Linked
3. Indexed

A. 1 and 3 only
B. 2 only
C. 3 only
D. 2 and 3 only

https://gateoverflow.in/118437/gate-cse-2017-set-2-question-08

Both Linked and Indexed allocation free from external fragmentation

D is answer

41

Refer: Galvin

Reference: https://webservices.ignou.ac.in/virtualcampus/adit/course/cst101/block4/unit4/cst101-bl4-u4-06.htm

Best answer

answered Feb 14, 2017 • edited Jun 22, 2021 by **Lakshman Patel RJIT**

comment Follow share this

Aboveallplayer

GATE CSE 2019 | Question: 42

13

The index node (inode) of a Unix -like file system has 12 direct, one single-indirect and one double-indirect pointers. The disk block size is 4 kB, and the disk block address is 32-bits long. The maximum possible file size is (rounded off to 1 decimal place) ____ GB

gatecse-2019 numerical-answers operating-system file-system

H.W.

https://gateoverflow.in/302806/gate-cse-2019-question-42

23

Best answer

Given 12 direct, 1 single indirect, 1 double indirect pointers

Size of Disk block $= 4kB$

Disk Block Address $= 32 \text{ bit } = 4B$

Number of addresses= Size of disk block/address size $= \frac{4kB}{4B} = 2^{10}$

Maximum possible file size$= 12 * 4kB \ + \ 2^{10} * 4kB \ + \ 2^{10} * 2^{10} * 4kB$

$= 4.00395 \ GB \simeq 4 \ GB$

Hence $4GB$ is the correct answer

GATE CSE 2021 Set 1 | Question: 15

MSQ

6

Consider a linear list based directory implementation in a file system. Each directory is a list of nodes, where each node contains the file name along with the file metadata, such as the list of pointers to the data blocks. Consider a given directory foo.

Which of the following operations will necessarily require a full scan of foo for successful completion?

A. Creation of a new file in foo
B. Deletion of an existing file from foo
C. Renaming of an existing file in foo
D. Opening of an existing file in foo

https://gateoverflow.in/357437/gate-cse-2021-set-1-question-15

Useful answer

Correct Options: A, C

15

Best answer

> *(**Note:** In the question it's given "which of the following options require a full scan of foo for successful completion" . Meaning the best algorithm scans the list entirely for each type of input to verify the correctness of the procedure and ,can't partially scan and complete for any particular instance...)*

Each File in Directory is uniquely referenced by its **name**. So **different files** must have **different names!**

So,

A. **Creation of a New File:** For creating new file, we've to check whether the new name is same as the existing files. Hence, the linked list must be scanned in its entirety.

B. **Deletion of an Existing File:** Deletion of a file doesn't give rise to name conflicts, hence if the node representing the files is found earlier, it can be deleted without a through scan.

C. **Renaming a File:** Can give rise to name conflicts, same reason can be given as option A.

D. **Opening of existing file**: same reason as option B.

GATE IT 2004 | Question: 67

30

In a particular Unix OS, each data block is of size 1024 bytes, each node has 10 direct data block addresses and three additional addresses: one for single indirect block, one for double indirect block and one for triple indirect block. Also, each block can contain addresses for 128 blocks. Which one of the following is approximately the maximum size of a file in the file system?

A. 512 MB
B. 2 GB
C. 8 GB
D. 16 GB

https://gateoverflow.in/3710/gate-it-2004-question-67

38

Answer: (**B**)

Maximum file size $= 10 \times 1024$ Bytes $+1 \times 128 \times 1024$ Bytes $+1 \times 128 \times 128 \times 1024$ Bytes $+1 \times 128 \times 128 \times 128 \times 1024$ Bytes $=$ approx $2\ GB$.

Best answer

answered Apr 6, 2015 • edited Jun 26, 2018 by **kenzou**

comment Follow share this

Rajarshi Sarkar

GATE CSE 2022 | Question: 53

asked in **Operating System** Feb 15 • edited Feb 19 by **Anjana5051**

2,002 views

9

Consider two files systems A and B, that use contiguous allocation and linked allocation, respectively. A file of size 100 blocks is already stored in A and also in B. Now, consider inserting a new block in the middle of the file (between 50^{th} and 51^{st} block), whose data is already available in the memory. Assume that there are enough free blocks at the end of the file and that the file control blocks are already in memory. Let the number of disk accesses required to insert a block in the middle of the file in A and B are n_A and n_B, respectively, then the value of $n_A + n_B$ is_____.

gatecse-2022 numerical-answers operating-system file-system

https://gateoverflow.in/371883/gate-cse-2022-question-53?show=371883#q371883

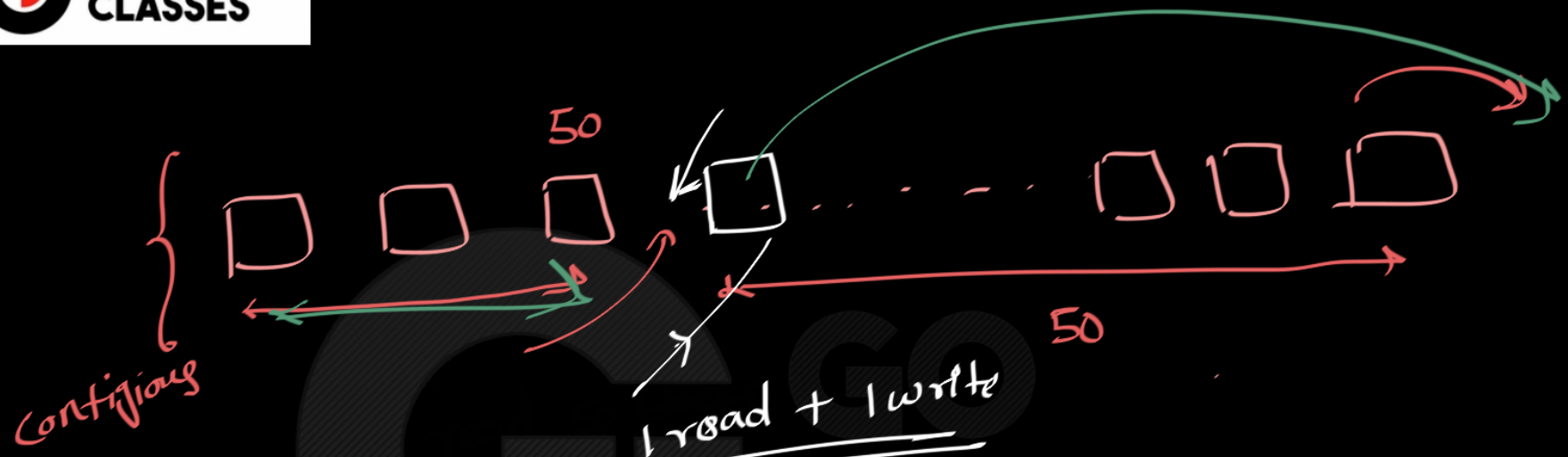
GATE CSE 2022 | Question: 53

asked in **Operating System** Feb 15 • edited Feb 19 by **Anjana5051**

2,002 views

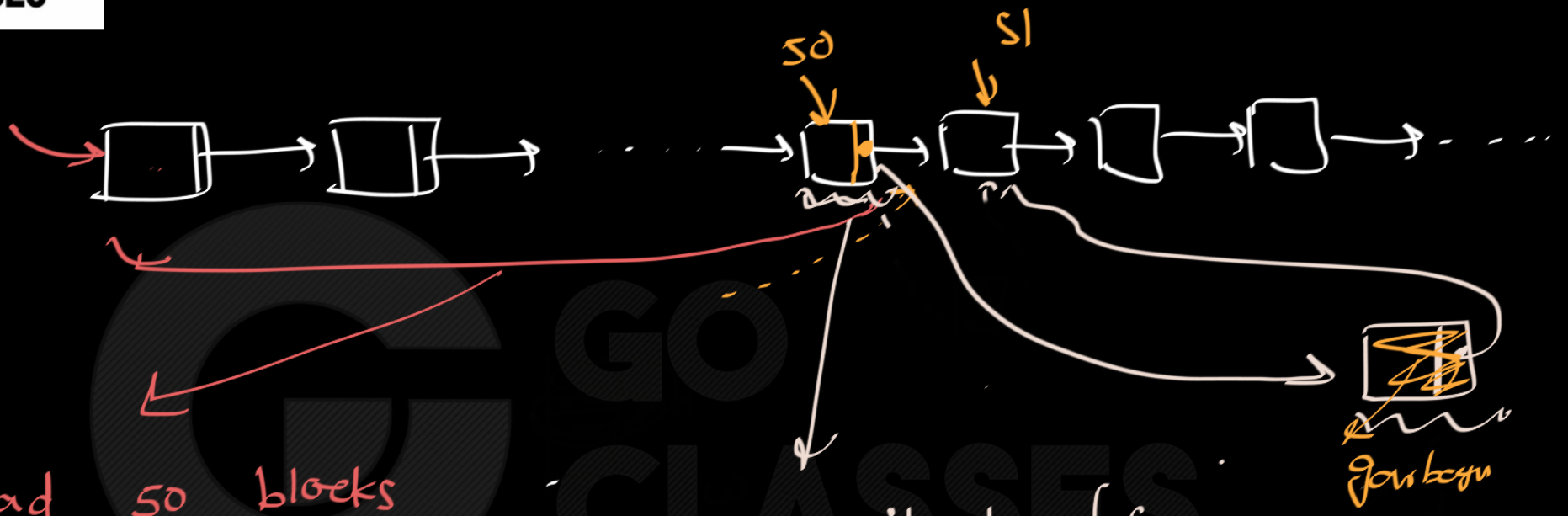
9

Consider two files systems A and B, that use contiguous allocation and linked allocation, respectively. A file of size 100 blocks is already stored in A and also in B. Now, consider inserting a new block in the middle of the file (between 50^{th} and 51^{st} block), whose data is already available in the memory. Assume that there are enough free blocks at the end of the file and that the file control blocks are already in memory. Let the number of disk accesses required to insert a block in the middle of the file in A and B are n_A and n_B, respectively, then the value of $n_A + n_B$ is_____.

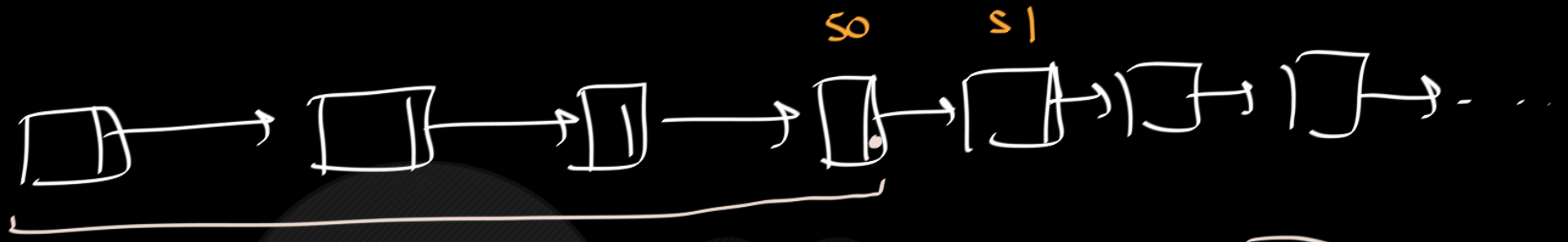


IO operation ← each read is one IO
each write is also another one IO

2 operations each for later 50 blocks
= 100 operation + 1 write
= 101



50 + 2 = 52 operations



[50 blocks read
1 write to 50th block
1 write to new block]

= 52 I O operations

11

Answer: 153

Explanation:

-No Free blocks- **1, 2, 3,..,49, 50,** New Block **, 51, 52, . . .,99, 100, ...** Free Blocks ...

Contiguous Allocation:

In case of **Contiguous** allocation we can directly go to the 50^{th} element. After this, we have to insert a block here, and since the allocation is **Contiguous**, therefore you need to shift all the remaining 50 blocks to the right. [As enough free blocks are available to the right and no free blocks are available in the beginning, so we can only shift the blocks to he right only].

So,

$50 \text{ Read Operations } + 50 \text{ Write Operations} + \mathbf{1}\ [1 \text{ operation to write a newly inserted block}] = 101$ Op

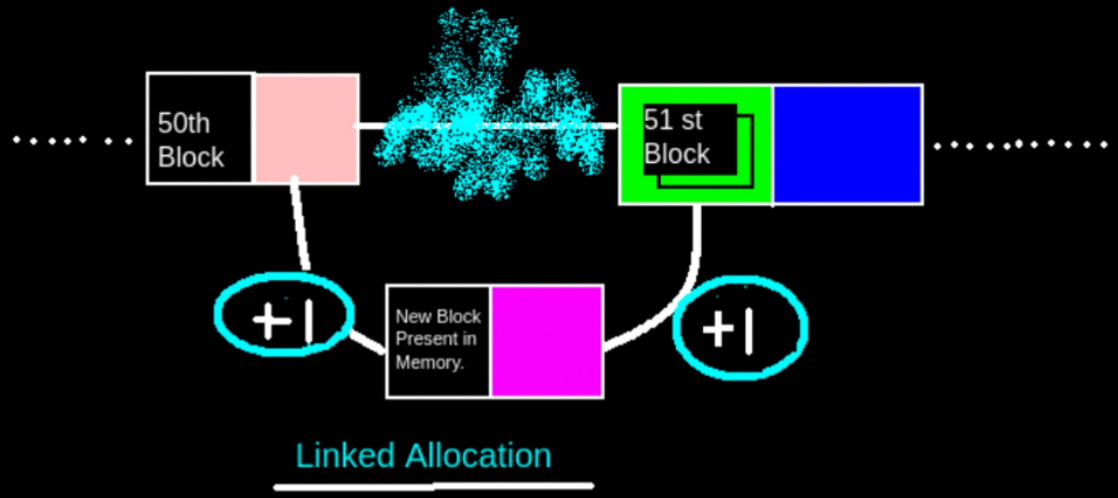
.

Also, we know from the Contiguous Memory allocation concept that overwriting an element simply means deleting it. Therefore, we don't have to worry about deleting an element specifically. We can just overwrite them, thus saving the cost of operations.

Linked Allocation:

In **Linked Allocation** , 50 operations to read first 50 elements , 2 operations are needed to delete the next pointer of the 50^{th} element, connect that link to the block which is to be inserted, and then connect the next pointer of that block to the 51^{st} element. This takes **2 operations.**

So, Total **52** operations are needed in this case.



Linked Allocation

Note:

- The statement "The file Control Blocks are already there in memory " means the information regarding the file structure is already present in the memory. (Eg: index block in the case of indexed allocation is already present in memory.)
- I/O Operations or Operations or Read & Write Operations or Disk Accesses, all of them simply represent the same thing.

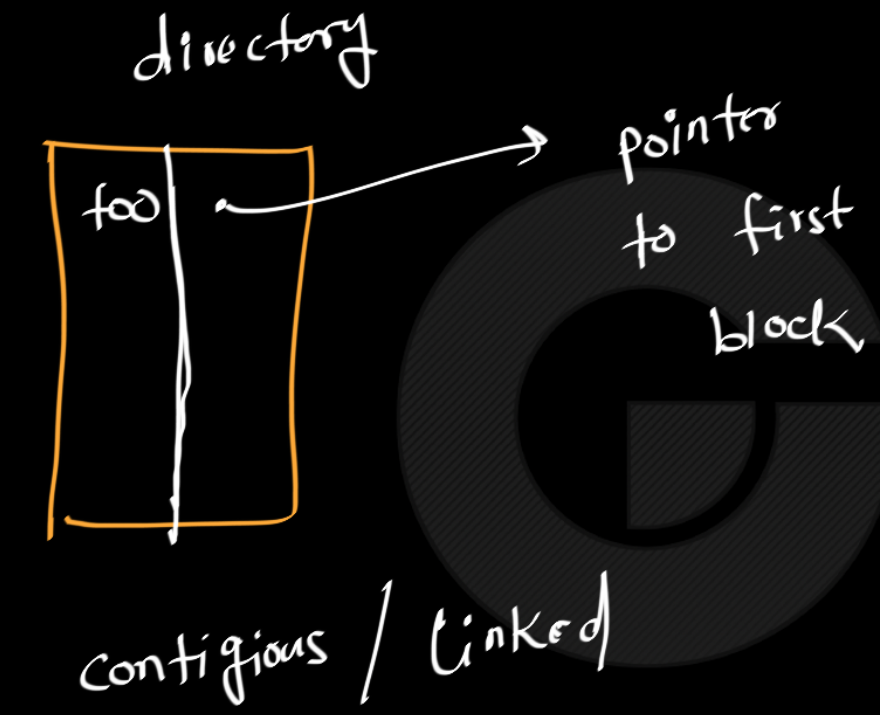
c. (9 Points) Suppose a file system can have three disk allocation strategies, contiguous, linked, and indexed. We have just read the information for a file from its parent directory. For contiguous and linked allocation, this gives the address of the first block, and for indexed allocation this gives the address of the index block. Now we want to read the 10^{th} data block into the memory. How many disk blocks (R) do we have to read for each of the allocation strategies? *For partial credit, explicitly list which block(s) you have to read.*

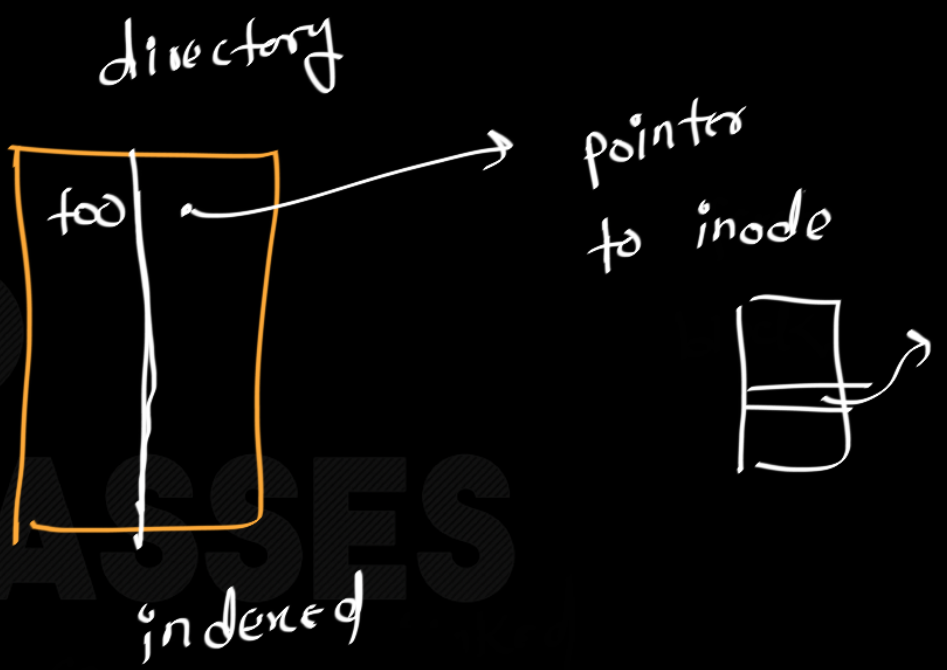
Contiguous allocation:

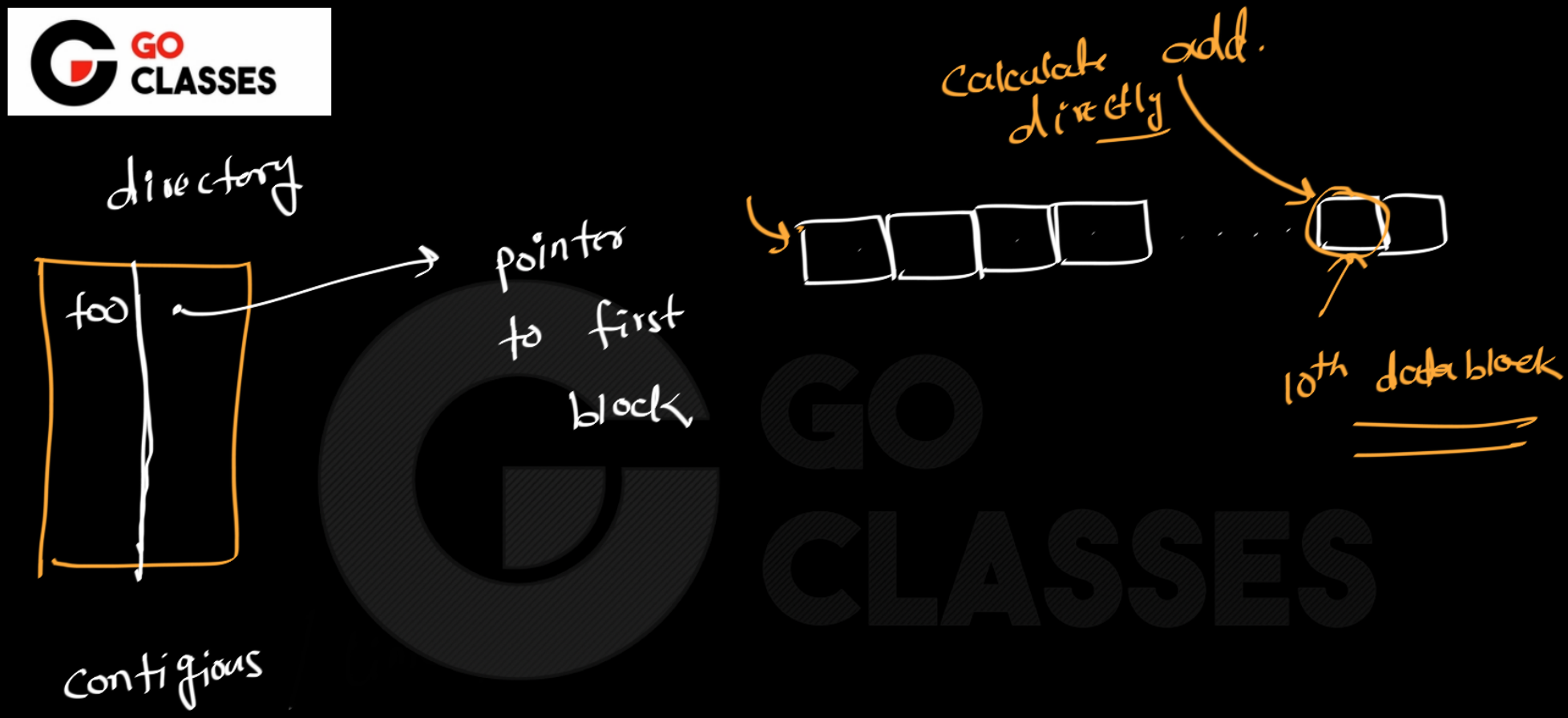
Linked allocation:

Indexed allocation:

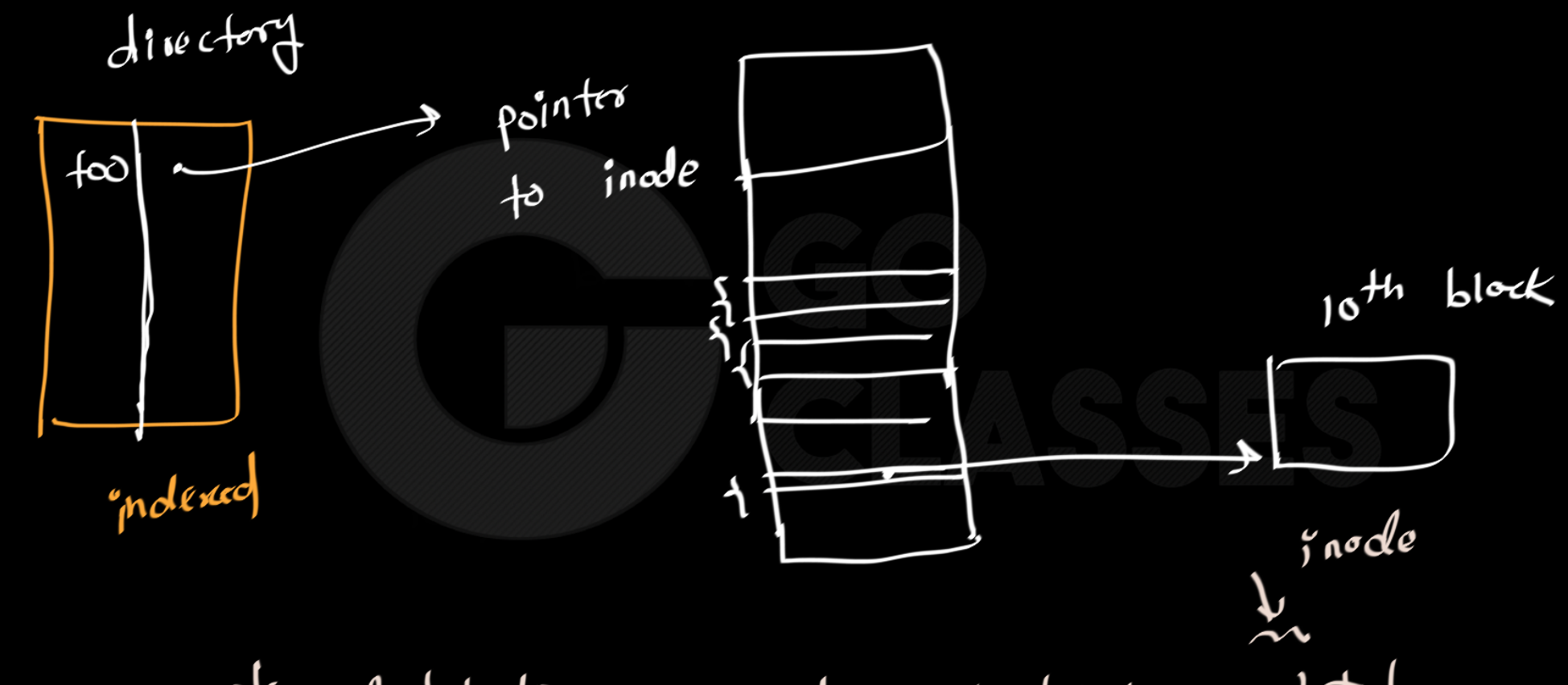
https://hkn.eecs.berkeley.edu/examfiles/cs162_sp03_mt2_sol.pdf


directory
foo
pointer to first block
contigious / linked


directory
foo
pointer to inode
indexed


directory
foo
pointer to first block
contigious
Calculate add. directly
10th data block


directory
foo
pointer to first block
Linked
10 reads



No. of blocks to read 10th block = 1 + 1

c. (9 Points) Suppose a file system can have three disk allocation strategies, contiguous, linked, and indexed. We have just read the information for a file from its parent directory. For contiguous and linked allocation, this gives the address of the first block, and for indexed allocation this gives the address of the index block. Now we want to read the 10^{th} data block into the memory. How many disk blocks (R) do we have to read for each of the allocation strategies? *For partial credit, explicitly list which block(s) you have to read.*

Contiguous allocation:

$R =$

1 block

We took off one point if you included reading a file header.
For major errors, we deducted two points.

Linked allocation:

$R =$

10 blocks

We took off one point if you included reading a file header.

Indexed allocation:

$R =$

2 blocks

We took off one point if you did not read the index block. We took off two points if you assumed the index block was already in memory.

7. **[15 points]** Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grows at the end. Also assume that the block information to be added is stored in memory.

   (a) The block is added at the beginning.

   (b) The block is added in the middle.

   (c) The block is added at the end.

   (d) The block is removed from the beginning.

   (e) The block is removed from the middle.

   (f) The block is removed from the end.

H.W.

Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**

add middle

	Contiguous	*Linked*	*Indexed*
(a)			
(b)			
(c)			
(d)			
(e)			
(f)			

Answer: The correct answers are shown in the table below:

http://www.csl.mtu.edu/cs4411.ck/www/EXAMS/FINAL/final-2019-Spring-sol.pdf

(a) The block is added at the beginning.

i. **Contiguous**: All 100 blocks have to be shifted one block. Because each shift requires one read and one write, $200 = 100 \times 2$ disk I/O operations are needed. Then, the new block is written to the first location, which requires one write. As a result, the number of disk I/O operations is $201 = 2 \times 100 + 1$.

ii. **Linked**: The address to the first block is found in the directory. Thus, this address is add to the address field of the new block, which is written to an available location. Hence, one 1 disk operation is needed.

iii. **Indexed**: The pointers in the index block are shifted to make the first position available so that the new block could be recorded there. Therefore, only 1 disk I/O is needed to write the new block to disk.

(b) The block is added in the middle. This is very similar to adding at the beginning.

i. **Contiguous**: The middle block can be computed from the directory because blocks are allocated in a contiguous way. Then, the 50 blocks in the second half are moved one block, which requires 50 reads and 50 writes. Finally, the new block is written to the original 50th position, and the number of disk I/O operations is $101 = 2 \times 50 + 1$.

ii. **Linked**: We need 50 reads to retrieve the address of the 51th block. The address of the 51th block is added to the address of the new block, which is written to disk with 1 disk I/O operation. The address of the original 50th block has to be updated with the address of the new block. Therefore, the number of disk I/O operations is $52 = (50 + 1) + 1$.

iii. **Indexed**: This is the same as the previous case, and only 1 disk I/O is needed to write the new block to disk.

(c) The block is added at the end.

i. **Contiguous**: One disk I/O operation is needed to write the new block to the end of the original contiguous allocation provided that there is a free block available. Otherwise, compaction is needed.

ii. **Linked**: First we need to retrieve the last block from directory (1 read). Its address field is updated with the address of the new block, and written back to disk (1 write). One more write is needed to write the new block to disk. Therefore, the number of disk I/O operation is $3 = 1 + 1 + 1$.

iii. **Indexed**: This is the same as the previous case, and only 1 disk I/O is needed to write the new block to disk.

(d) The block is removed from the beginning.

i. **Contiguous**: The remaining 99 blocks are moved forward and each move requires 1 read and 1 write, The total number of disk I/O operations is $198 = 2 \times 99$.

ii. **Linked**: Read the first block to find the second one. The second block's address is used to update the directory of this file. Hence, only 1 disk I/O operation is needed.

iii. **Indexed**: Shifting the addresses stored in the index block one position upward does not need any disk I/O operation.

(e) The block is removed from the middle.

i. **Contiguous**: The remaining 49 blocks are moved forward and each move requires 1 read and 1 write, The total number of disk I/O operations is $98 = 2 \times 49$.

ii. **Linked**: This is the same as inserting in the middle. The number of disk I/O operations is 52.

iii. **Indexed**: Shifting the addresses stored in the index one position upward does not need any disk I/O operation.

(f) The block is removed from the end.

i. **Contiguous**: Just modify the directory to reflect the number of blocks is one less than the original. No disk I/O operation is needed.

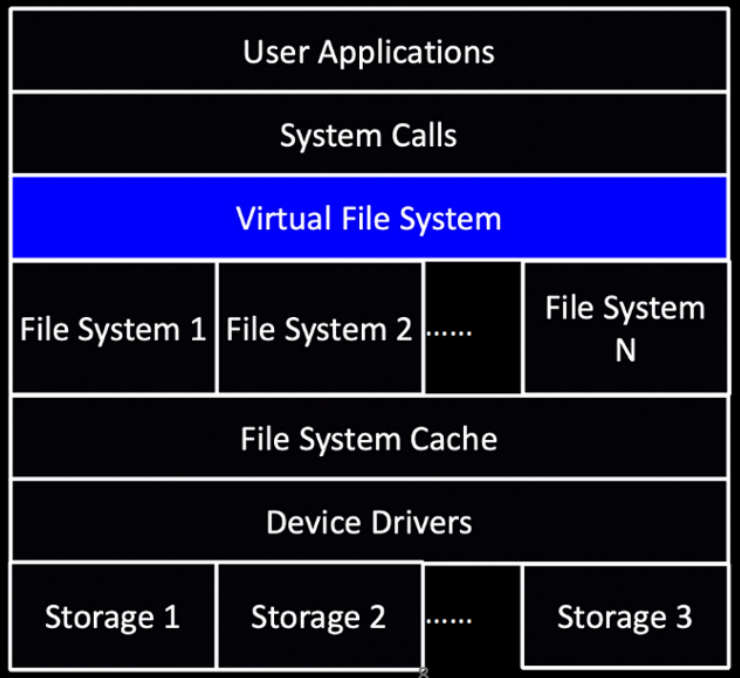
ii. **Linked**: We need to read the first 99 blocks, change the address field of the 99th to `NULL`, and write it back. The number of disk I/O operations is $100 = 99 + 1$.

iii. **Indexed**: Shifting the addresses stored in the index one position upward does not need any disk I/O operation.

(Not for GATE)

Virtual File System (VFS)

- VFS provides
  1. A common system call interface to user applications to access different file systems implemented in the OS.
  2. A common interface to file systems to "plug into" the operating system and provide services to user applications.



[One OS can support multiple FS]

Free Space

- Find the block to use when one is needed
  - Find space quickly
  - Keep storage reasonable
- Options
  - Bit vector
  - Linked List
  - Grouping
  - Counting

using

indexed

contiguous

Linked list

[there are free blocks here and there]

- → i should get there free blocks quickly.
- → i should not waste too much of space to store info. of free blocks

- Bit vector (*n* blocks)

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

1 0 0 1 0 0 0 0 1 0

- Bit vector downside
  - Space
- Example:

  block size = 2^{12} bytes ← 4 KB

  disk size = 2^{30} bytes (1 gigabyte)

  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

no. of blocks

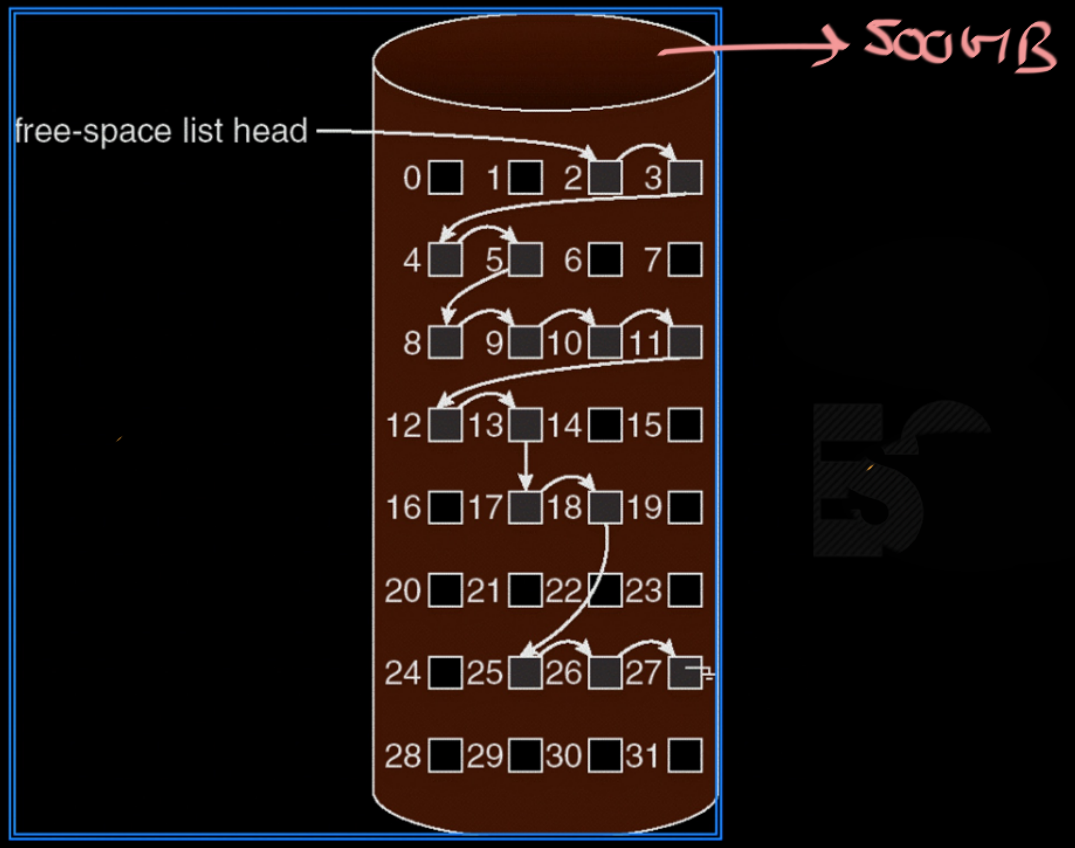
32 KB ⇒ 8 blocks for bit map

Free-Space Linked List

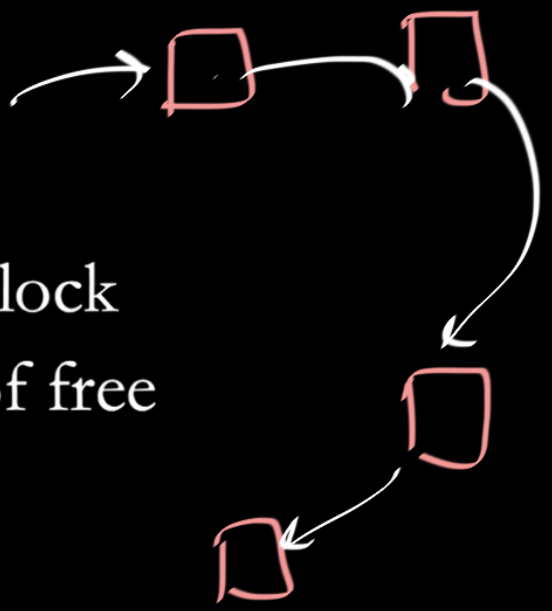
Free list

→ counting

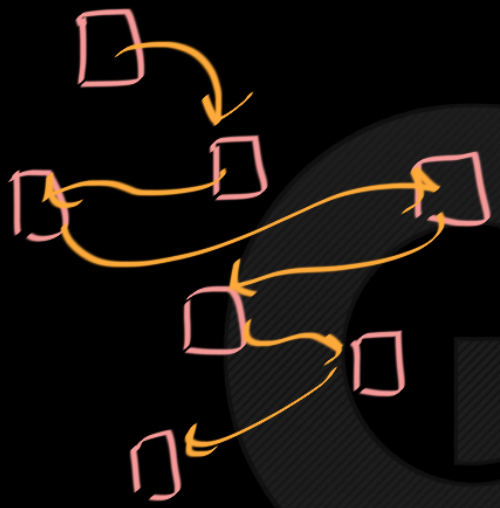
→ grouping

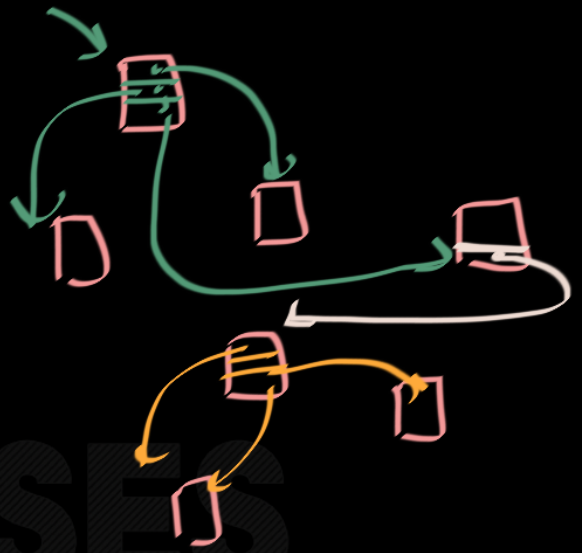


Free-Space Linked Optimizations

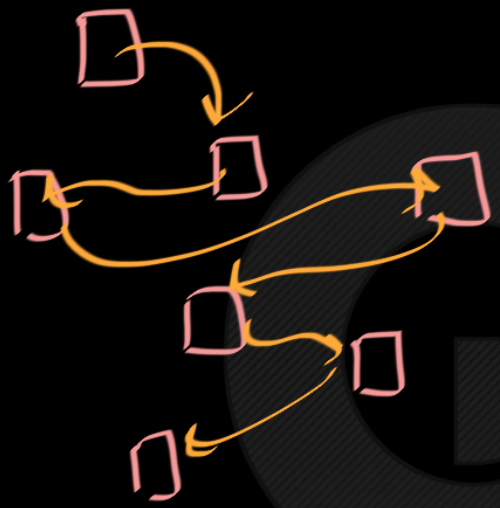


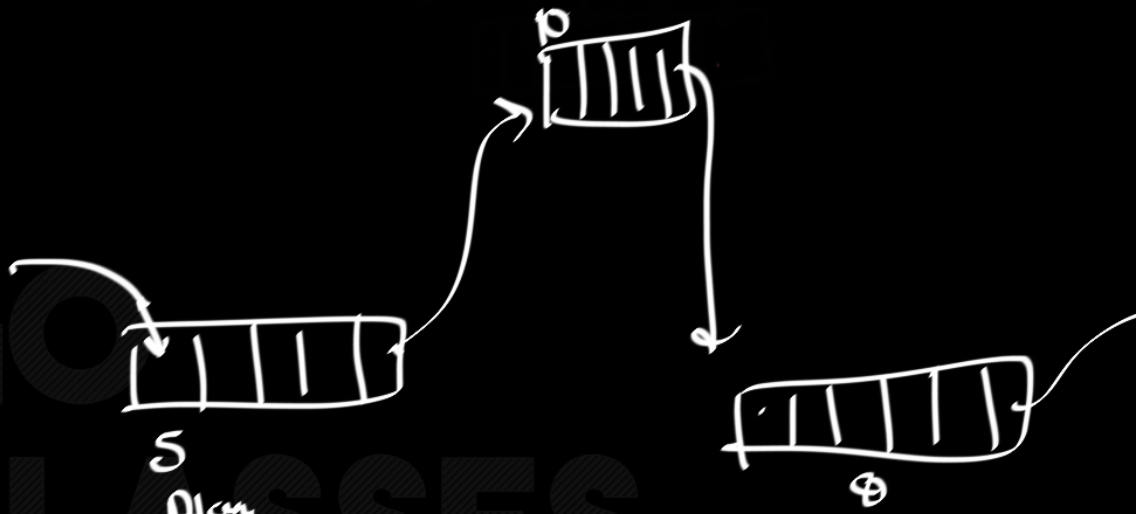
- Grouping
  - Store n free blocks in first free block
  - Last entry points to next block of free blocks
- Counting
  - Specify start block and number of contiguous free blocks





free list





free list

counting

There are numerous file systems in use; for example, FAT32 and NTFS are Windows file systems, and HFS is used on Macs. Linux uses ext2, ext3 and FAT32. Unix systems use UFS, ext2, ext3 and ZFS.

As such, the file system is highly operating-system-dependent. In most cases you don't generally have a "choice" between different file system types. However, in some operating systems you do, and there can be a performance impact from the choice. Windows NT and 2000 typically give you your choice of file system; Windows 2000 supports FAT16, FAT32 and NTFS.

File System Formulas (Types)

Below some types of file system:

Hierarchical File System (HFS)

File Allocation Table (FAT)

New Technology File System (NTFS)

Hierarchical File System (HFS)

HFS is a file system type developed by Apple Inc. for use on computers running Mac OS. Two main variants of HFS exist: Mac OS Standard ("HFS Standard" or "HFS") and Mac OS extended ("HFS extended" or "HFS+"). If you are running Mac OS X, your bootable drive is almost certainly using HFS+, not standard HFS. HFS+ allows for larger files with longer file names to be stored on the disk.

File Allocation Table (FAT)

FAT is a brief for File Allocation Table, which dates back to the beginnings of DOS programming. The File Allocation Table (FAT) file system was the primary file system in Microsoft's older operating systems, it is a file system that was created by Microsoft in 1977. FAT was the primary file system used in all of Microsoft's consumer operating systems from MS-DOS through Windows ME.

Over the years, the file system has been expanded from FAT12 to FAT16 and FAT32. Various features have been added to the file system including subdirectories, extended attributes, and long filenames. Below is more information on the versions of the FAT file system:

New Technology File System (NTFS)

NTFS is a file system type that is commonly used for Microsoft Windows. It is the standard file system for Windows NT, Windows 2000, Windows XP, Windows Vista and Windows 7. It provides numerous improvements over the FAT file system, including better security and better disk utilization. **NTFS** is a proprietary file system developed by Microsoft Corporation for its Windows line of operating systems, beginning with Windows NT 3.1 and Windows 2000, including Windows XP, Windows Server 2003, and all their successors to date.

NTFS offers several features that are not available with FAT 32: file compression, encryption, permissions, and mirroring drives. NTFS supersedes the FAT file system as the preferred file system for Microsoft's Windows operating systems. NTFS has several technical improvements over FAT and HPFS (High Performance File System), such as the use of advanced data structures to improve performance, reliability, and disk space utilization, plus additional extensions, such as security access control lists (ACL).