Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 19: File systems

Prof. Ryan Huang

Considerations in storing files

Need to store metadata (besides data)

- Keep track of where file contents are on disk
- File size, owner/permissions, time of creation/last access

Access a file object through an initial handle

- We call this structure a file header (*e.g.*, inode in Unix)
- Use it to map file offset to disk block
- File header must be stored on disk, too!

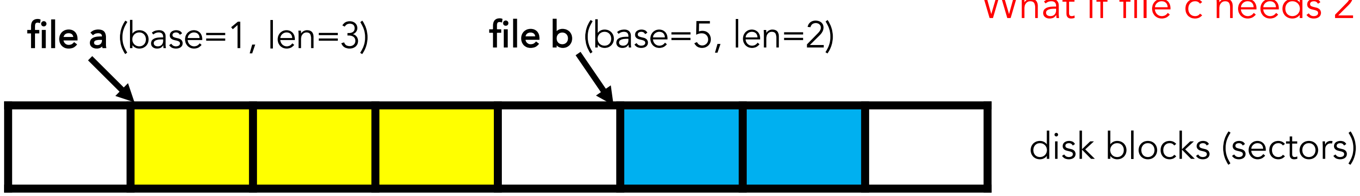
Things to keep in mind while designing file structure:

- Most files are small
- Much of the disk is allocated to large files
- Many of the I/O operations are made to large files
- Want good sequential and good random access

Approach 1: contiguous allocation

"Extent-based": allocate files like segmented memory
- When creating a file, user pre-specifies its length; all space allocated at once
- File header contents: location and size

What if file c needs 2 sectors?



Example: IBM OS/360

Pros?
- Fast sequential access: data sequential in file space is sequential in disk space
- Easy random access: easy to compute disk location

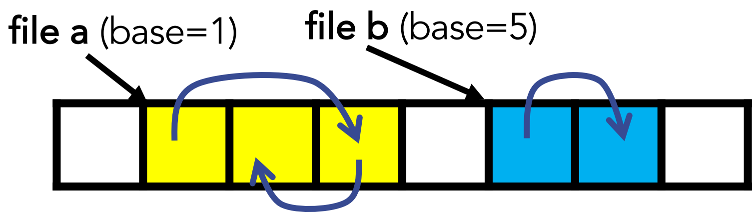
Cons? (Think of corresponding VM scheme)
- External fragmentation
- Difficult to dynamically grow files after creation

Approach 2: linked files

Basically a linked list on disk

- Keep a linked list of all free blocks
- File header contents: a pointer to file's first block
- In each block, keep a pointer to the next one

How do you find last block in a?



Examples (sort-of): Alto, TOPS-10, DOS FAT

Pros?

- Easy dynamic growth & sequential access
- No fragmentation

Cons?

- Linked list on disk is a bad idea because of access times
- Random access very slow (e.g., traverse whole file to find last block)

Example: DOS FS (simplified)

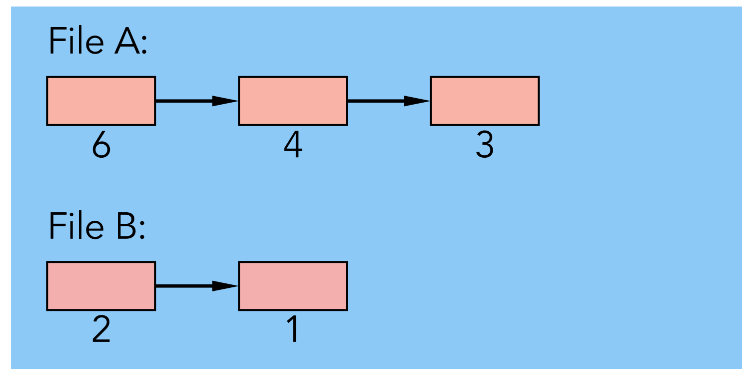
Linked files with a key optimization: *puts links in fixed-size "file allocation table" (FAT) rather than in each data block*

Directory

A: 6	
B: 2	

FAT

0	free
1	EOF
2	1
3	EOF
4	3
5	EOF
6	4
	...

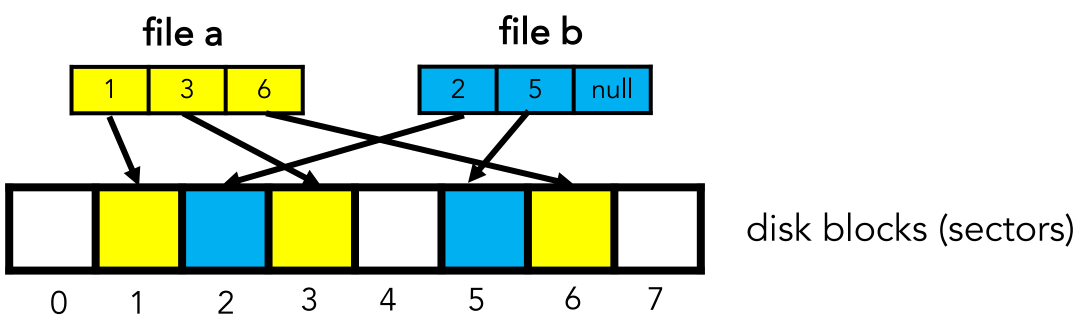


Still do pointer chasing, but can cache entire FAT in memory, so it can be cheap compared to disk access

Approach 3: indexed files

File header stores an array of its block pointers

- Like a page table
- Max file size fixed by array's size
- Allocate array to hold file's block pointers on file creation
- Allocate actual blocks on demand using free list



Pros?

- Easy random access
- Easy to grow file

Cons?

- Large files? → mapping table requires large chunk of contiguous space

... same problem we were trying to solve initially

Indexed files

Issues are the same as in page tables

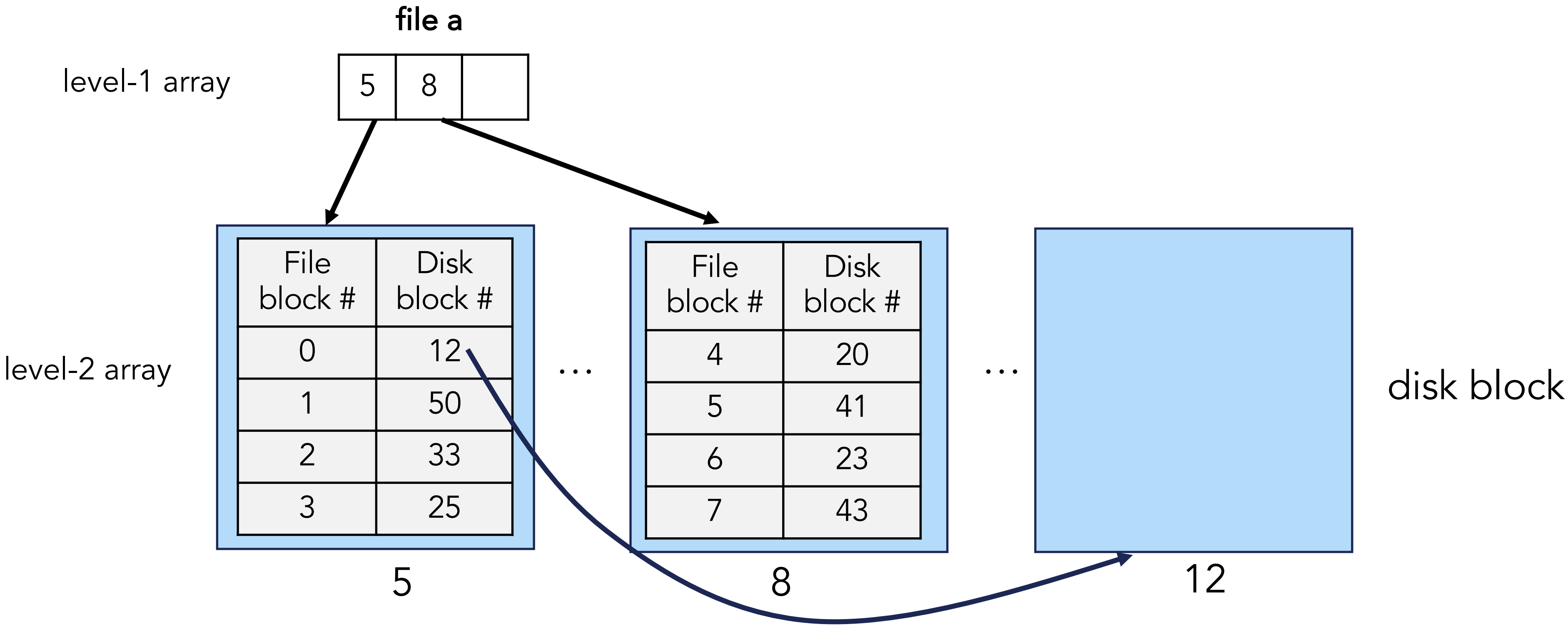
- Large possible file size = lots of unused entries
- Large actual size? table needs large contiguous disk chunk

Solve identically: small regions with index array, this array points to another array, …

- Allows large files, but small files don't waste header space

Multi-level indexed files

File header stores level-1 index array



Problems?

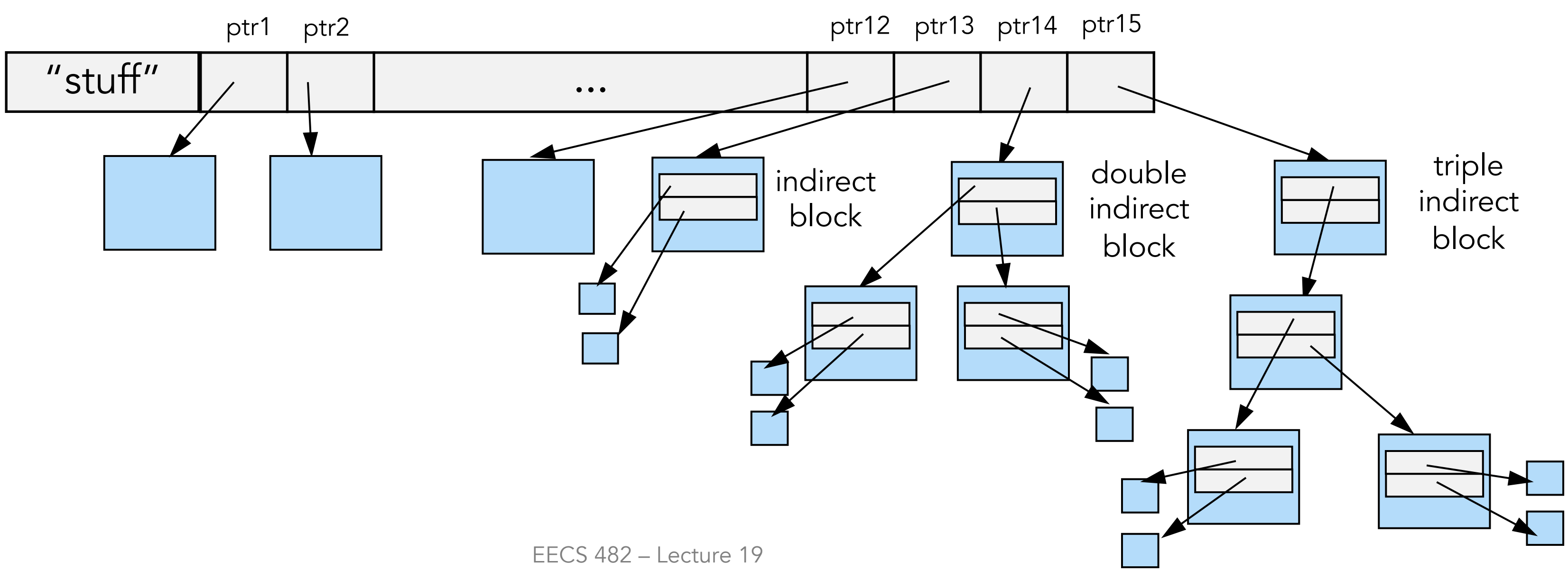
- Sequential access is slow, small files suffers poor performance

Multi-level indexed files: Unix inodes

Idea: use non-uniform depths

inode = 15 block pointers + “stuff”

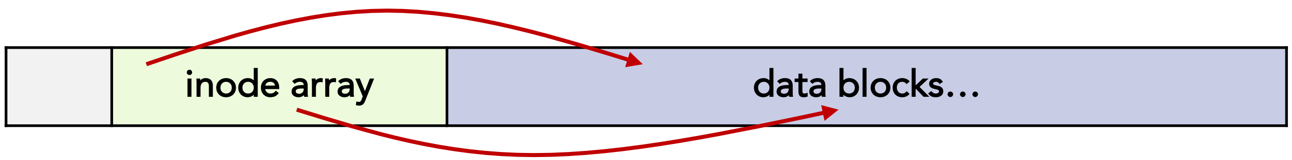
- first 12 are direct blocks (points to data blocks)
  - solve problem of first blocks access slow
- then single, double, and triple indirect block



More about Unix inodes

inodes are stored in a fixed-size array

- Size of array fixed when disk is initialized; can't be changed
- Lives in known location, originally at one side of disk:



- Index of an inode in the inode array called an i-number
    - Internally, the OS refers to files by *i-number*
    - `ls -i <file>` shows the i-number of a file
- When file is opened, inode brought in memory
- Written back when modified/file closed/time elapses

Directories

Problem: referencing files

How do users specify which file they want to access?
- Ask users to remember where on disk their files are (sector no.)?
  - E.g., like remembering your social security or bank account #

...People want human digestible names

Directories serve two purposes
- For users, they provide a structured way to organize files
- For FS, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk

Data structure to locate file headers for a set of files

Directories and files

What data is contained in a directory?

- Directory entries that ap name of a file → file header's disk block # for that file

lampson83.txt	20
home	5
…	

- Many data structures possible, e.g., list, hash table, B-tree

How to store directories?

- Often handle directories and files in the same way
  - Same storage structure (e.g., indexed files)
- But some differences
  - Users can read/write arbitrary data to a file
  - Why not allow users to read/write arbitrary data to a directory?

Directories in Unix

Unix inodes are *NOT* directories
- Inodes describe where on the disk the blocks for a file are placed

Directories are *files*, so a dir is stored as an inode

The inode of a directory describes where the data blocks for a directory are on disk
- The data blocks contain the directory entries
- Each directory entry is a mapping of <name, inode #>

Unix inodes do *NOT* store file names!

Hierarchical file path translation

How to look up a hierarchical file path?

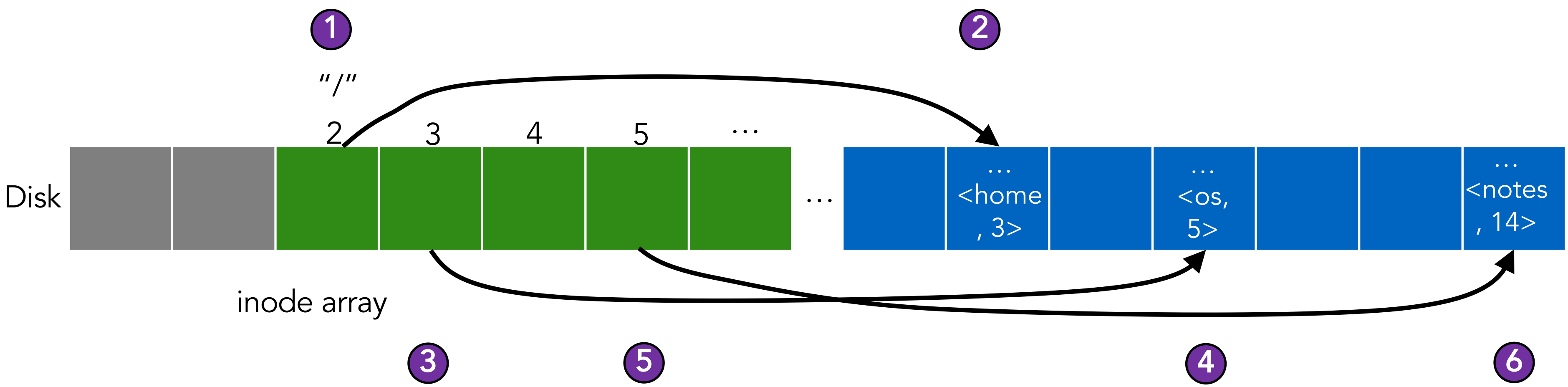
Starting point is root directory "/"

- Need a convention of where it resides
- On Unix, inode #2 is for "/"

Example: open "/note.txt"

1. Read inode #2 ("/") from disk into memory
2. Based on the block pointer of this inode, read the *content of "/"* from disk into memory, look for entry for "note.txt"
3. This entry gives the inode # for "note.txt"
4. Read the inode for "note.txt" into memory
5. This inode says where first data block is on disk
6. Read that block into memory to access the data in the file

Example: /home/os/notes



How many disk accesses to read the first byte in notes?

Unified view of multiple storage devices

Combine multiple storage devices into a file system
- Each device contains own file system (starting with its root)
- A directory entry can point to the root of a different device

Example: login.engin.umich.edu

/ (root)
  bin (same device as /)
  tmp (file system on separate storage device)
  afs (file system on network servers)

Directory entries
- File
- Directory
- Device

File buffer cache

Lots of on-disk data

- Inodes, file data, directories

Disk operations are slow…

Applications exhibit locality for file accesses

Idea: cache file blocks in memory to capture locality

- Called the file buffer cache
- Cache is system wide, used and shared by all processes
- Reading from the cache makes a disk perform like memory
- Even a small cache can be very effective

Caching writes

slow | fast
safe | dangerous

write-through ⟷ delayed write-back ⟷ write-back

Should cache be write-through or write-back?

- Write-through: poor performance
- Write-back: loses data on OS crash, power failure

Current file systems use delayed write-back

- Background daemon periodically flushes dirty pages (e.g., every 30 sec)
- If blocks changed many times in 30 secs, only need one I/O
- If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed

Unreliable, but practical

- On a crash, all writes within last 30 secs are lost
- Modern OSes do this by default; too slow otherwise
- System calls (Unix: fsync) enable apps to force data to disk

Virtual memory and file caching

Address spaces

Memory

Disk

Increase capacity via swap space

File systems

Memory

Increase speed via file cache

Disk

Both use physical memory as a cache for disk

- Virtual memory: Use disk for increased capacity
- File systems: Use memory for faster performance

Both compete for physical memory

- Local vs. global replacement

Why have two separate mechanisms for caching disk data in memory?

Memory-mapped files

Use the VM paging system to cache both virtual address space and file system data (`mmap()` in Unix)

- Map file into a virtual address space
- Point the backing store for that part of the address space at the file's data blocks

Advantages

- Uniform access for files and memory (just use pointers)
- Less copying

Drawbacks

- Process has less control over data movement
  - OS handles faults transparently
- Does not generalize to streamed I/O (pipes, sockets, etc.)

Read ahead

Many file systems implement "read ahead"

- FS predicts that the process will request next block
- FS goes ahead and requests it from the disk
- This can happen while the process is computing on previous block
  - Overlap I/O with execution
- When the process requests block, it will be in cache

For sequentially accessed files can be a big win

- Unless blocks for the file are scattered across the disk
- File systems try to prevent that, though (during allocation)