CSE 451: Operating Systems

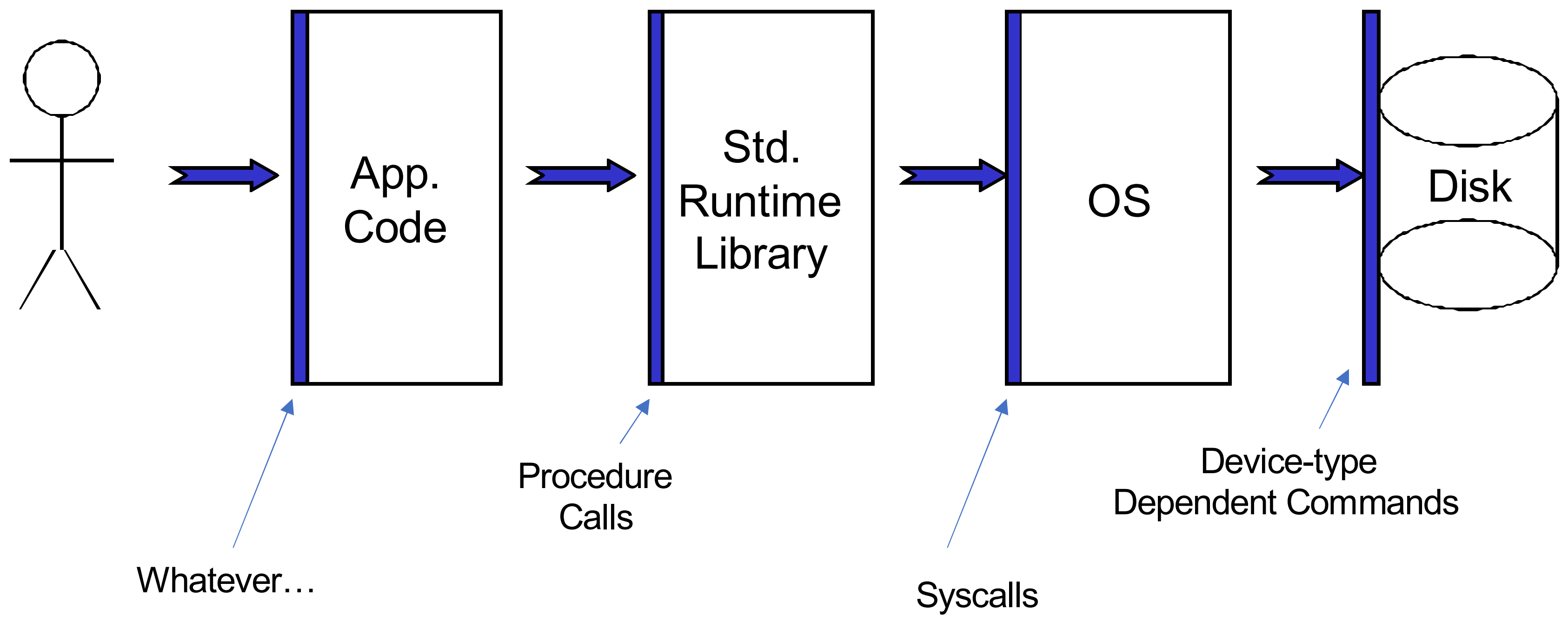
Autumn 2020

Module 10

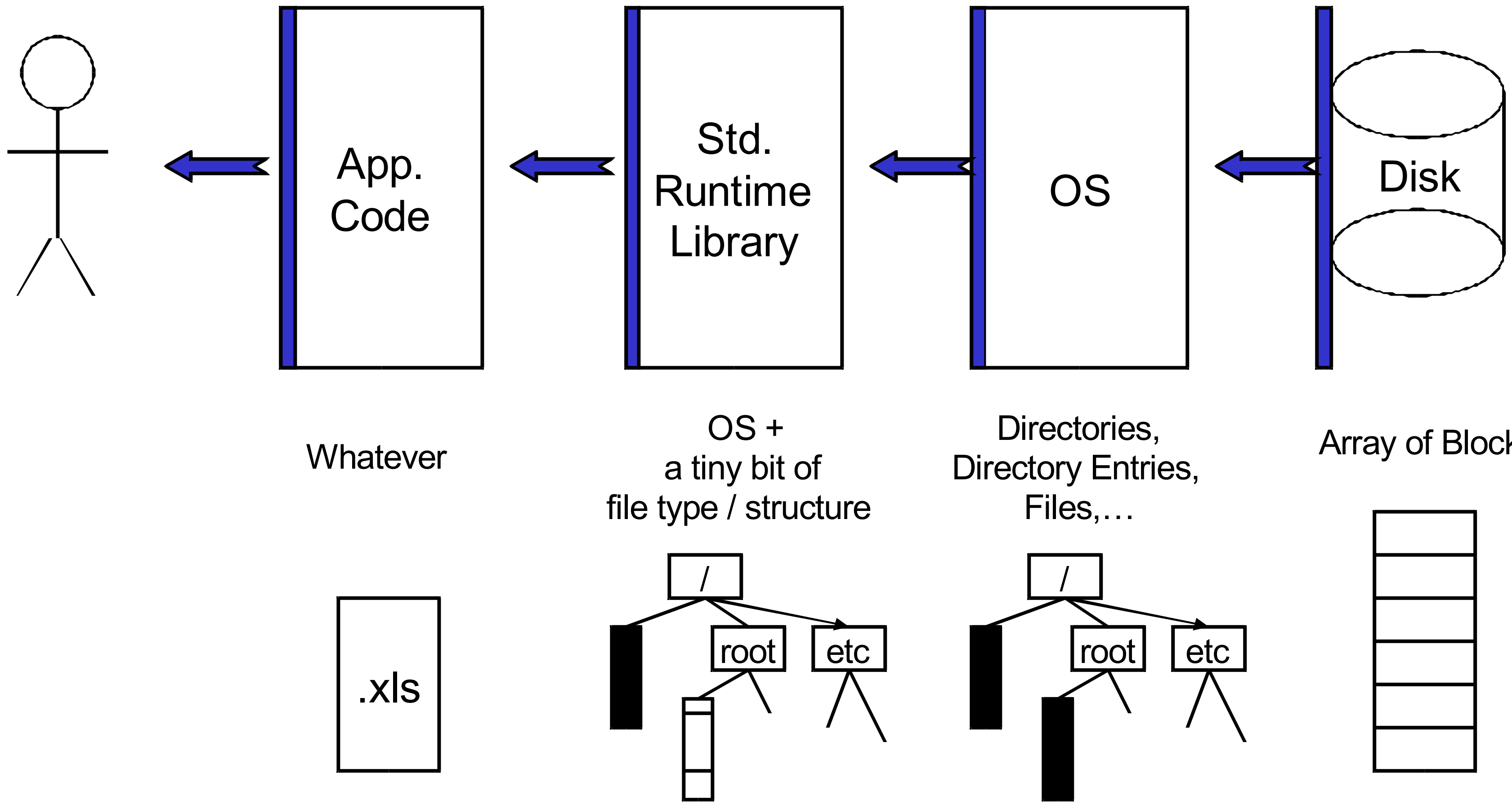
File Systems

John Zahorjan

Interface Layers



Exported Abstractions


App. Code
Std. Runtime Library
OS
Disk
Whatever
OS + a tiny bit of file type / structure
Directories, Directory Entries, Files,…
Array of Blocks
.xls
/
root
etc
/
root
etc

High Latency → Caching

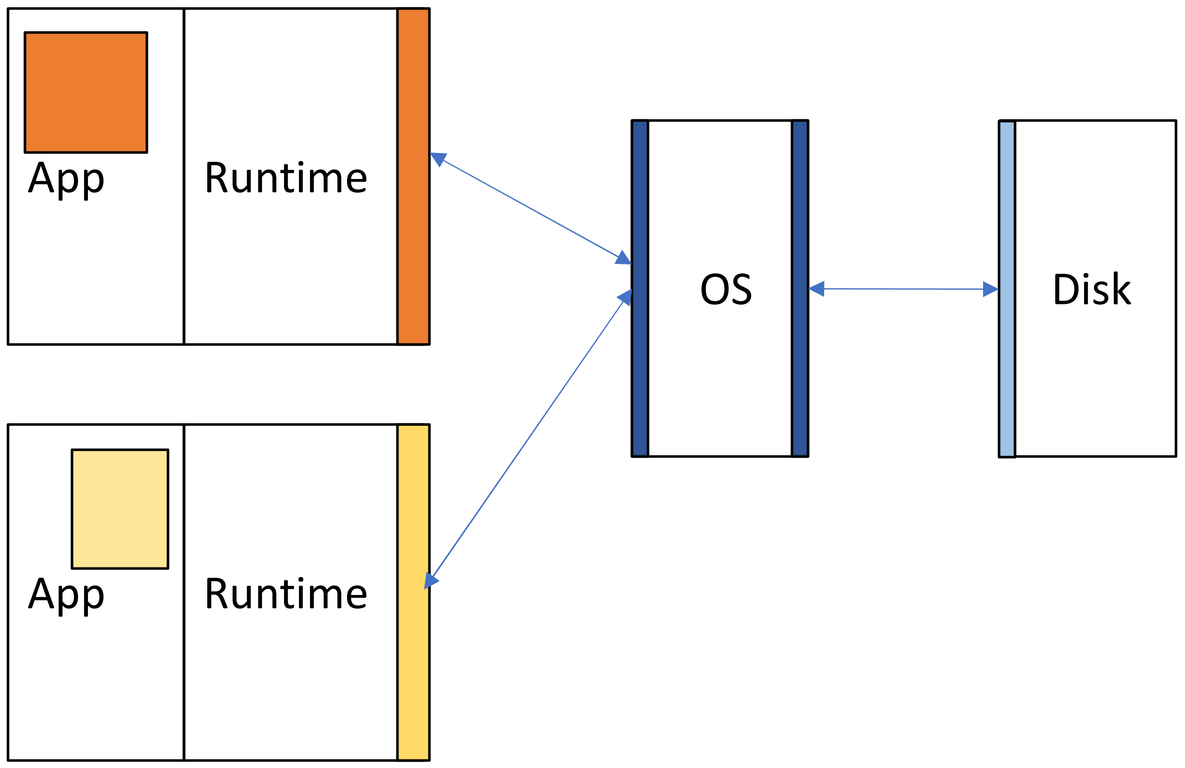
- General Rule: High latency → Caching

- (Other important approach: make big requests)
- (Other important approach: speculate / make request early)

- How do caches help with reads?
- How do caches help with writes?
- Are caches effective?
  - Temporal locality
  - Spatial locality
  - Other locality properties?

- Where would you put caches?

Caches



File systems

- A “file system” is an abstract data type
    - A set of byte arrays and a hierarchical naming scheme for identifying them
    - Operations on the bytes and on the names
- A “file system” is the representation of the ADT on secondary storage
    - ADT = files and file names
    - implementation = decisions about how to represent an instance on disk
    - file operations: create, read, write, delete (sort of)
    - name operations: directory create, read, write, delete
- A “file system” is an implementation of the above
    - E.g., NTFS, EXT4, ZFS
- Scope of file system names is typically larger than a single process
    - Promotes sharing of data among processes, people and machines
    - Requires access control, consistency, …

Files

- A file is a collection of data with some properties (*metadata*)
  - contents, size, owner, last read/write time, protection …

- Files may also have types
  - understood by file system
    - device, directory, symbolic link
  - understood by other parts of OS or by runtime libraries
    - executable, dll, source code, object code, text file, …

- Type may be encoded in the file's name or contents
  - windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, …
  - old Mac OS stored the name of the creating program along with the file
  - Unix has a smattering of both
    - in content via magic numbers or initial characters (e.g., #!)
    - see the file command

Reminder

- There is a difference between a file’s name and the file itself
  - Can a file have more than one name?

- Sometimes the namespace is borrowed for things that aren’t “files”
  - /proc/cpuinfo
  - named pipes (FIFOs)
  - network connections
  - etc.

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

File access methods

- Some file systems provide different access methods that specify ways the application will access data
  - sequential access
    - read bytes in order
  - direct access
    - random access given a block/byte #
  - record access
    - file is array of fixed- or variable-sized records
  - indexed access
    - FS contains an index to a particular field of each record in a file
    - apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
  - what might the FS do differently in these cases?

Directories

- Directories provide:
  - a way for users to organize their files
  - a convenient file name space for both users and FS's
- Most file systems support multi-level directories
  - naming hierarchies (/, /usr, /usr/local, /usr/local/bin, …)
- Most file systems support the notion of current directory
  - absolute names: fully-qualified starting from root of FS

    ```
    bash$ cd /usr/local
    ```
  - relative names: specified with respect to current directory

    ```
    bash$ cd /usr/local    (absolute)
    bash$ cd bin           (relative, equivalent to cd /usr/local/bin)
    ```

Directory internals

- A directory is typically just a file that happens to contain special metadata
  - directory = list of (name of file, file attributes)
  - attributes can include such things as:
    - size, protection, location on disk, creation time, access time, …
  - the directory list is usually unordered (effectively random)
    - when you type “ls”, the “ls” command sorts the results for you

Path name translation

- Let’s say you want to open “/one/two/three”

```
fd = open(“/one/two/three”, O_RDWR);
```

- What goes on inside the file system?
  - open directory “/” (well known, can always find)
  - search the directory for “one”, get location of “one”
  - open directory “one”, search for “two”, get location of “two”
  - open directory “two”, search for “three”, get loc. of “three”
  - open file “three”
  - (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
  - this is why open is separate from read/write (session state)
  - OS will cache prefix lookups to enhance performance
    - /a/b, /a/bb, /a/bbb all share the “/a” prefix

File protection

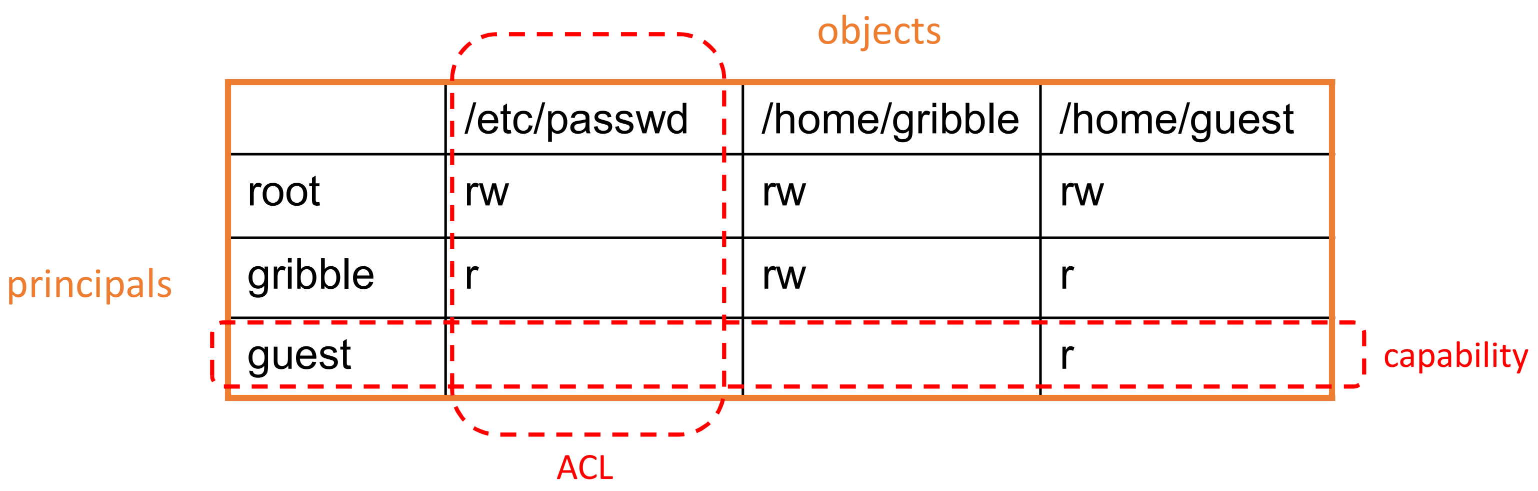
- FS must implement some kind of protection system
  - to control who can access a file (user)
  - to control how they can access it (e.g., read, write, or exec)

- More generally:
  - generalize files to objects (the “what”)
  - generalize users to principals (the “who”, user or program)
  - generalize read/write to actions (the “how”, or operations)

- A protection system dictates whether a given action performed by a given principal on a given object should be allowed
  - e.g., you can read or write your files, but others cannot
  - e.g., your can read `/etc/motd` but you cannot write to it

Model for representing protection

- Two different ways of thinking about it:
    - access control lists (ACLs)
        - for each object, keep list of principals and principals' allowed actions
    - capabilities
        - for each principal, keep list of objects and principal's allowed actions
- Both can be represented with the following matrix:


objects
principals
capability
ACL

	/etc/passwd	/home/gribble	/home/guest
root	rw	rw	rw
gribble	r	rw	r
guest			r

ACLs vs. Capabilities

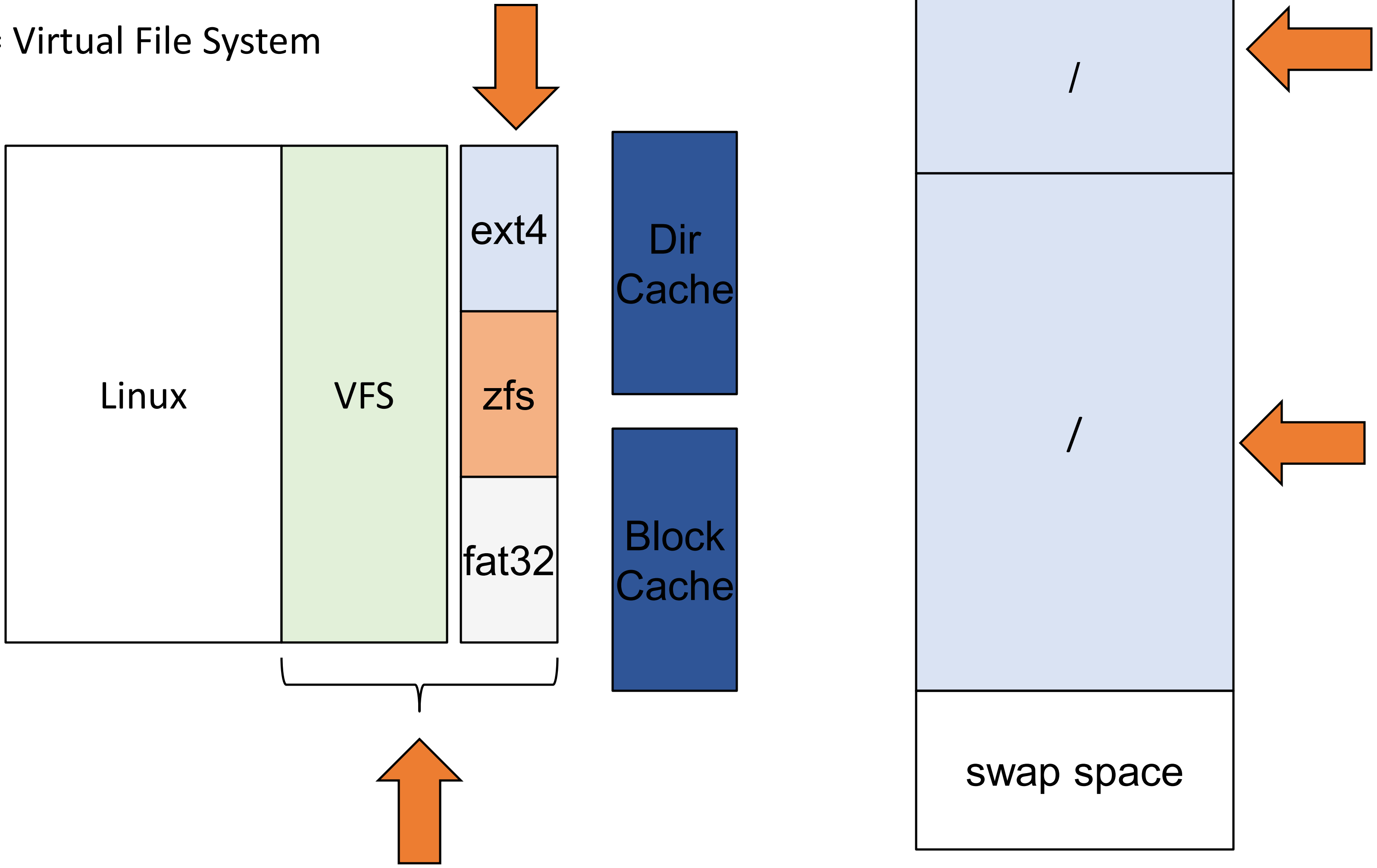
- Capabilities are easy to transfer
  - they are like keys: can hand them off
    - E.g., Google Drive sharing link
  - they make sharing easy

- ACLs are easier to manage
  - object-centric, easy to grant and revoke
    - to revoke capability, need to keep track of principals that have it
    - hard to do, given that principals can hand off capabilities

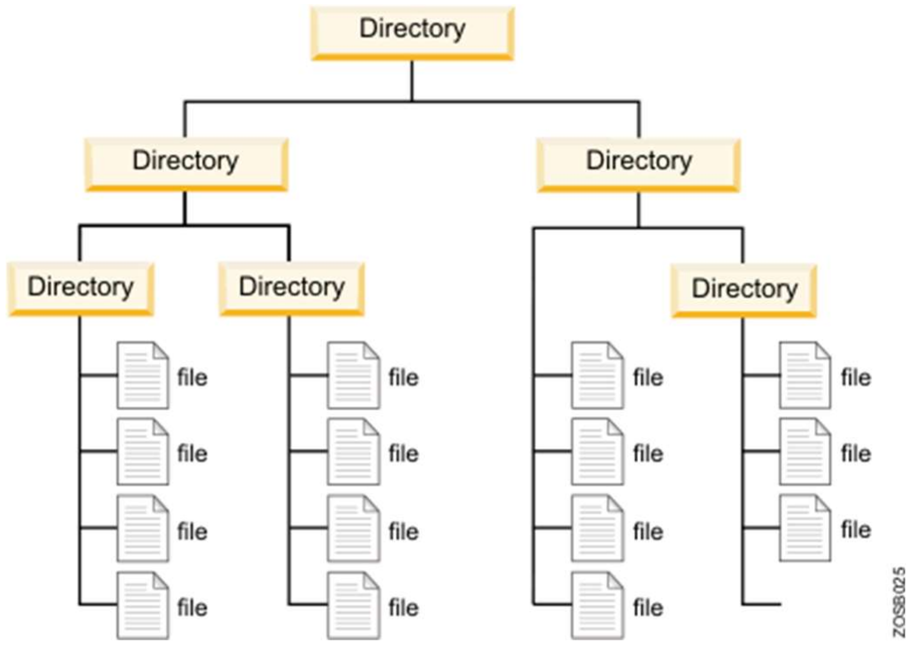
- ACLs grow large when object is heavily shared
  - can simplify by using “groups”
    - put users in groups, put groups in ACLs
    - you are probably in the “ugrad_cs” group on attu
  - additional benefit
    - change group membership, affects ALL objects that have this group in its ACL

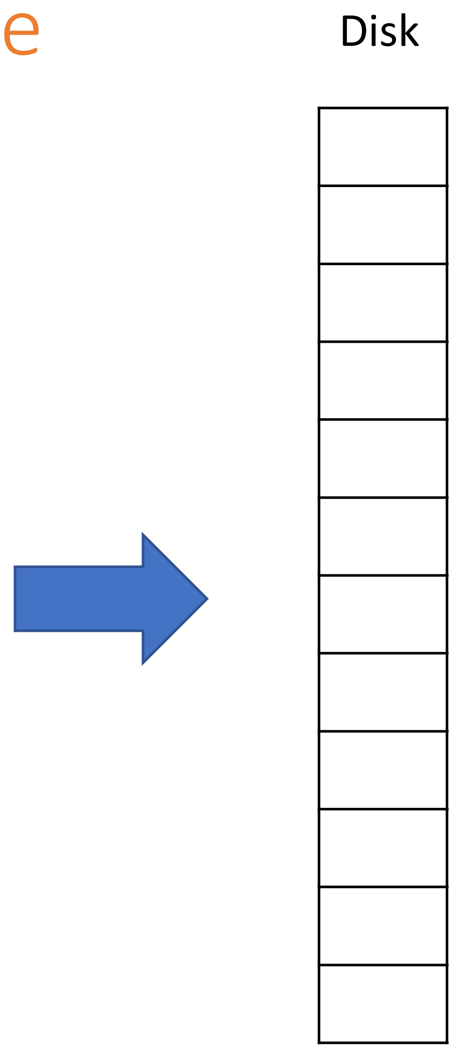
The Term "File System"

VFS = Virtual File System



File System Basic Layout Issue





https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconcepts_177.htm

File System Design Options

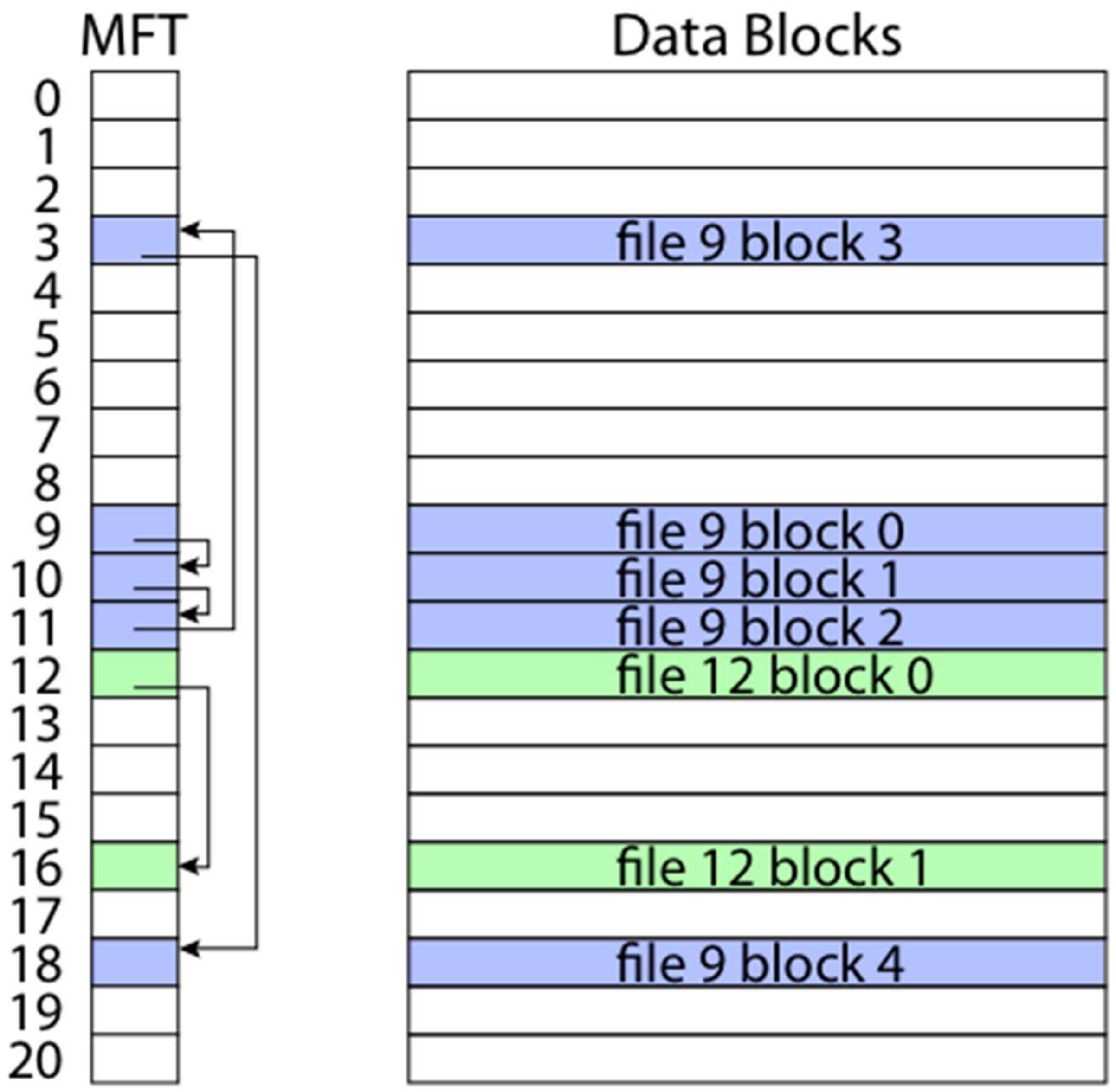
	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

1. Microsoft File Allocation Table (FAT)

- Linked list index structure
  - Simple, easy to implement
  - Still used (e.g., thumb drives)

- File allocation table:
  - Linear map of all blocks on disk
  - Each file is a linked list of blocks

FAT



MFT
Data Blocks
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
file 9 block 3
file 9 block 0
file 9 block 1
file 9 block 2
file 12 block 0
file 12 block 1
file 9 block 4

FAT

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file
- Cons:
  - Small file access is slow
  - Random access is very slow
  - Fragmentation
    - File blocks for a given file may be scattered across disk
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills
      - “Defrag”

2. The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- “UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system” – Multics
- Designed for a “workgroup” sharing a single system
- A wonderful study in engineering tradeoffs

- The legacy lives on...

(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
  - fixed-size blocks; head of freelist is in the superblock
- Swap area
  - holds processes that have been swapped out of memory

- Easy to find these structures…
  - Boot and superblock in fixed locations
  - The rest is located at locations that can be calculated from info in superblock

So …

- You can attach a disk to a system …
- (Boot it up …)
- Find, create, and modify files …
- Because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are

- By convention, the second i-node is the root directory of the *volume* (storage that has been formatted for/by the file system)

The tree (directory, hierarchical) file system

- A directory is represented as a flat file of fixed-size entries
  - Operations on directories are part of file system abstraction
  - ***Format** is file system implementation dependent*

- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my
144	a_directory

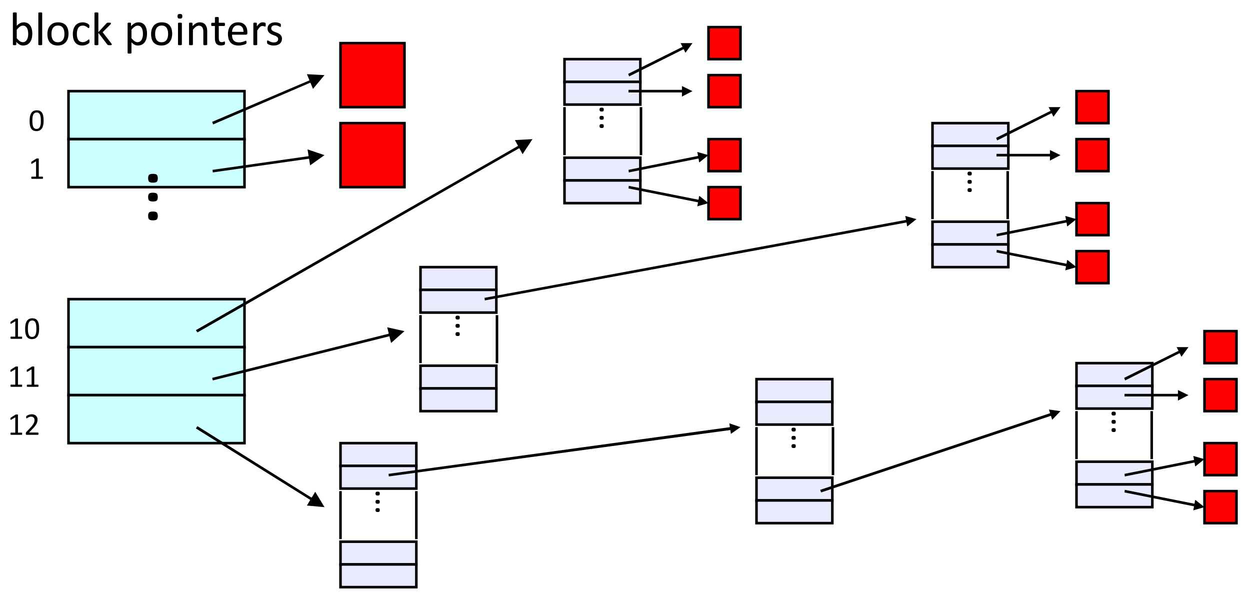
i-nodes (file metadata)

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a “special file” (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents (data) of file (in the file contents area)
  - more on this soon!
- Link count: number of directory entries referencing this i-node

Note: directories are part of the file system abstraction
inodes are part of the implementation

The “block list” portion of the i-node (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 13 block pointers
    - first 10 are “direct pointers” (pointers to 512B blocks of file data)
    - then, single, double, and triple indirect pointers



Why?

- Common case:
  - Most files are small.
  - Most accesses are sequential

- Index structure occupies 13 x 4B in the i-node
  - Most files are small $\rightarrow$ no additional index space required

- Supports sequential access patterns efficiently
  - i-node contains locations of first 10 blocks
  - each (final level) indirect block contains 128 consecutive block locations

- Supports random access patterns reasonably efficiently
  - Worst case access of any data byte requires five disk operations
    - Read inode
    - Read indirect block
    - Read indirect block
    - Read indirect block
    - Read data block

Details

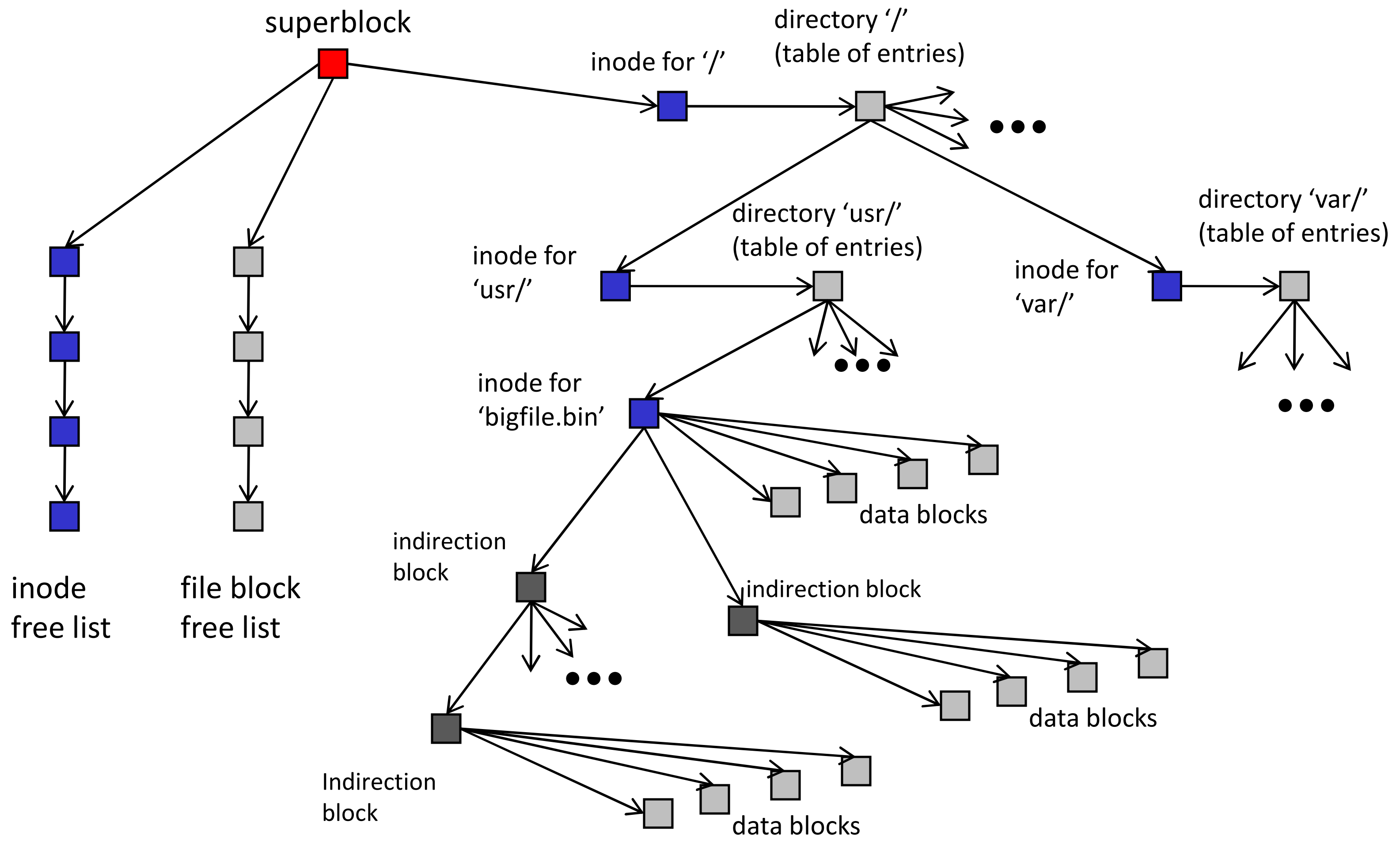
- Can get to 10 x 512B = a 5120B file directly
  - 10 direct pointers, blocks in the file contents area were 512B
  - *Most files are small*
- Can get to 128 x 512B = an additional 65KB with a single indirect reference
  - the 11th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to blocks holding file data
- Can get to 128 x 128 x 512B = an additional 8MB with a double indirect reference
  - the 12th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data
- Can get to 128 x 128 x 128 x 512B = an additional 1GB with a triple indirect reference
  - the 13th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data
- Maximum file size is 1GB + a smidge

Evolution

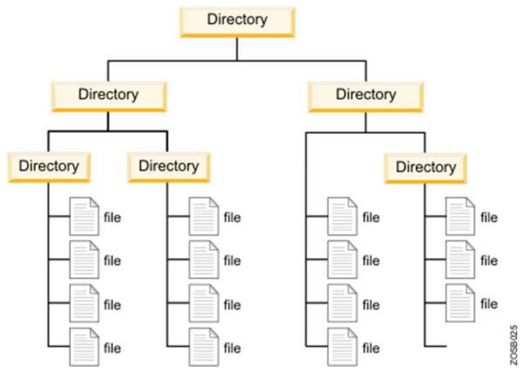
- A later version of Bell Labs Unix utilized 12 direct pointers rather than 10
  - Why?
- Berkeley Unix went to 1KB block sizes
  - Larger blocks better exploit spatial locality in access patterns
  - What's the effect on the maximum file size?
    - 1KB is enough for 256 4-byte block identifiers
    - 256x256x256x1K = 17 GB max file size
  - What is the maximum file system size?
  - What's the price?
- Subsequently went to 4KB blocks
  - 1Kx1Kx1Kx4K = 4TB

Putting it all together

- A file system is just a data structure

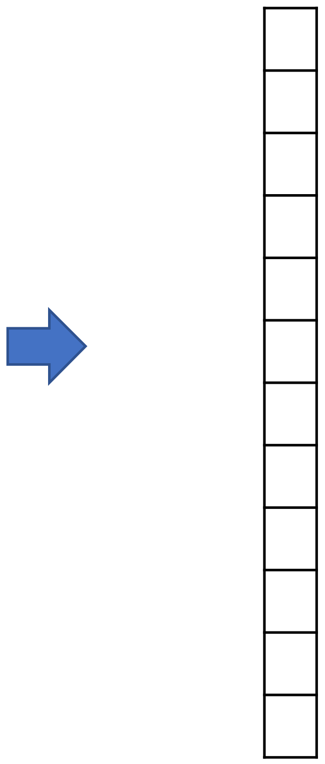


Block Allocation

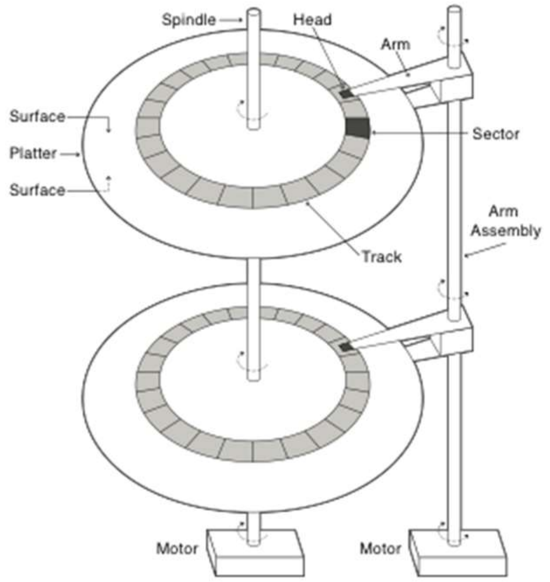


https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconcepts_177.htm

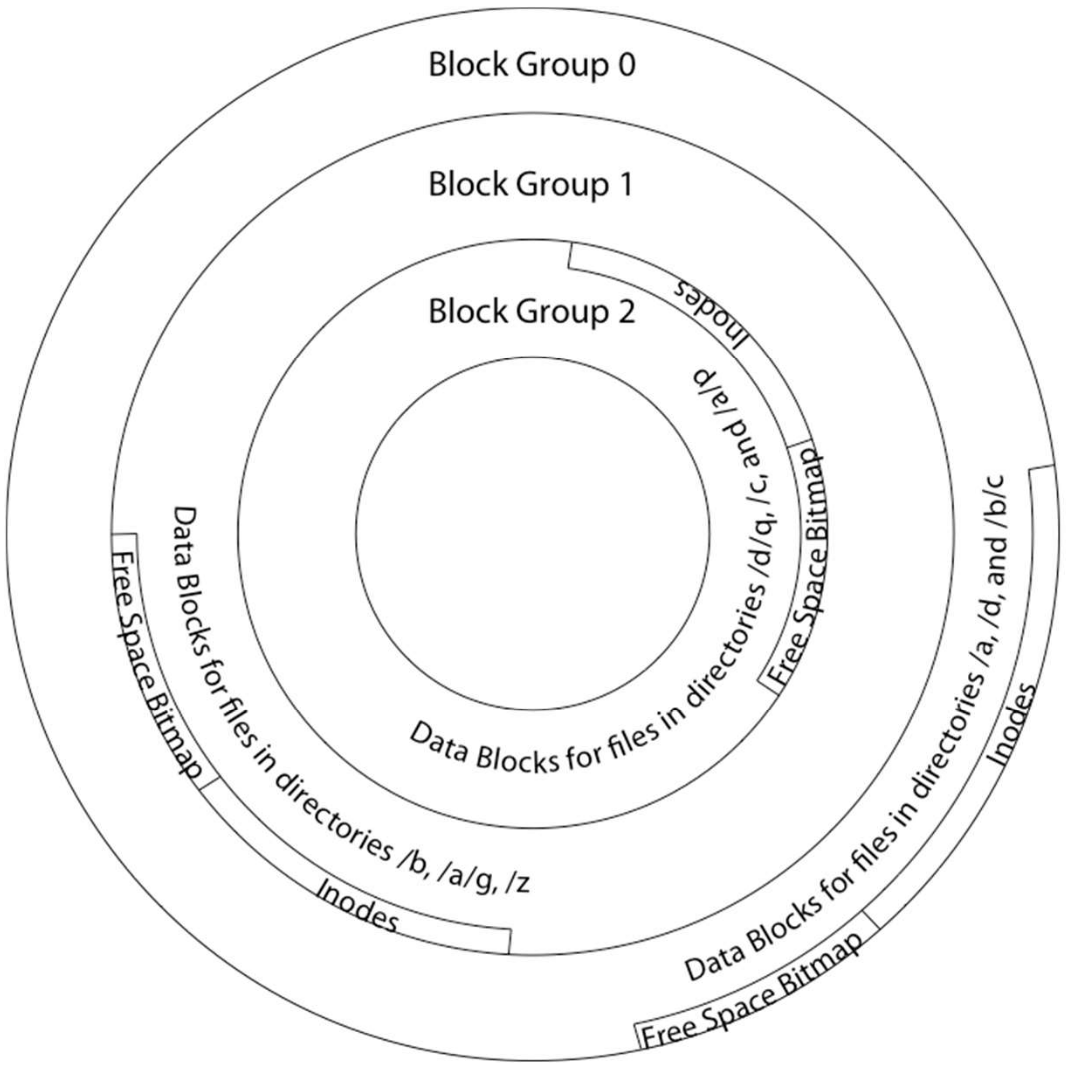
Logical



Physical



Block Allocation (FFS)



Block Group 0
Block Group 1
Block Group 2
Inodes
Data Blocks for files in directories /d/q, /c, and /a/p
Free Space Bitmap
Data Blocks for files in directories /b, /a/g, /z
Free Space Bitmap
Inodes
Data Blocks for files in directories /a, /d, and /b/c
Inodes
Free Space Bitmap

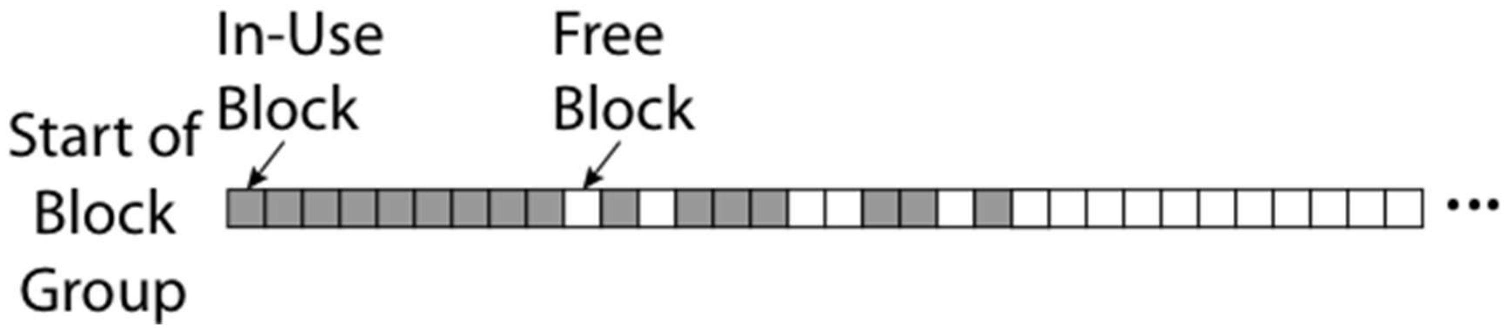
- Want to avoid seeks

- So want blocks of a file on same/nearby tracks
- (Want files in a directory near each other)

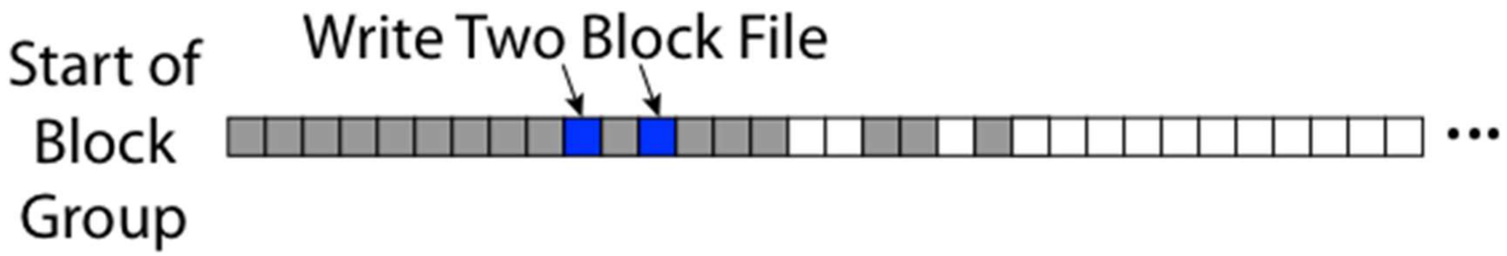
- Logically organize disk into block groups
- Superblock describes them

- Can still compute inode location given its number

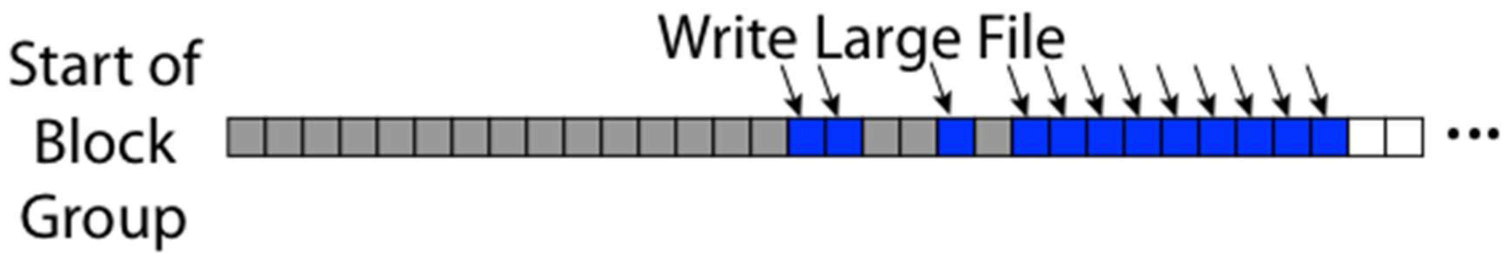
FFS First Fit Block Allocation



FFS First Fit Block Allocation



FFS First Fit Block Allocation



FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data

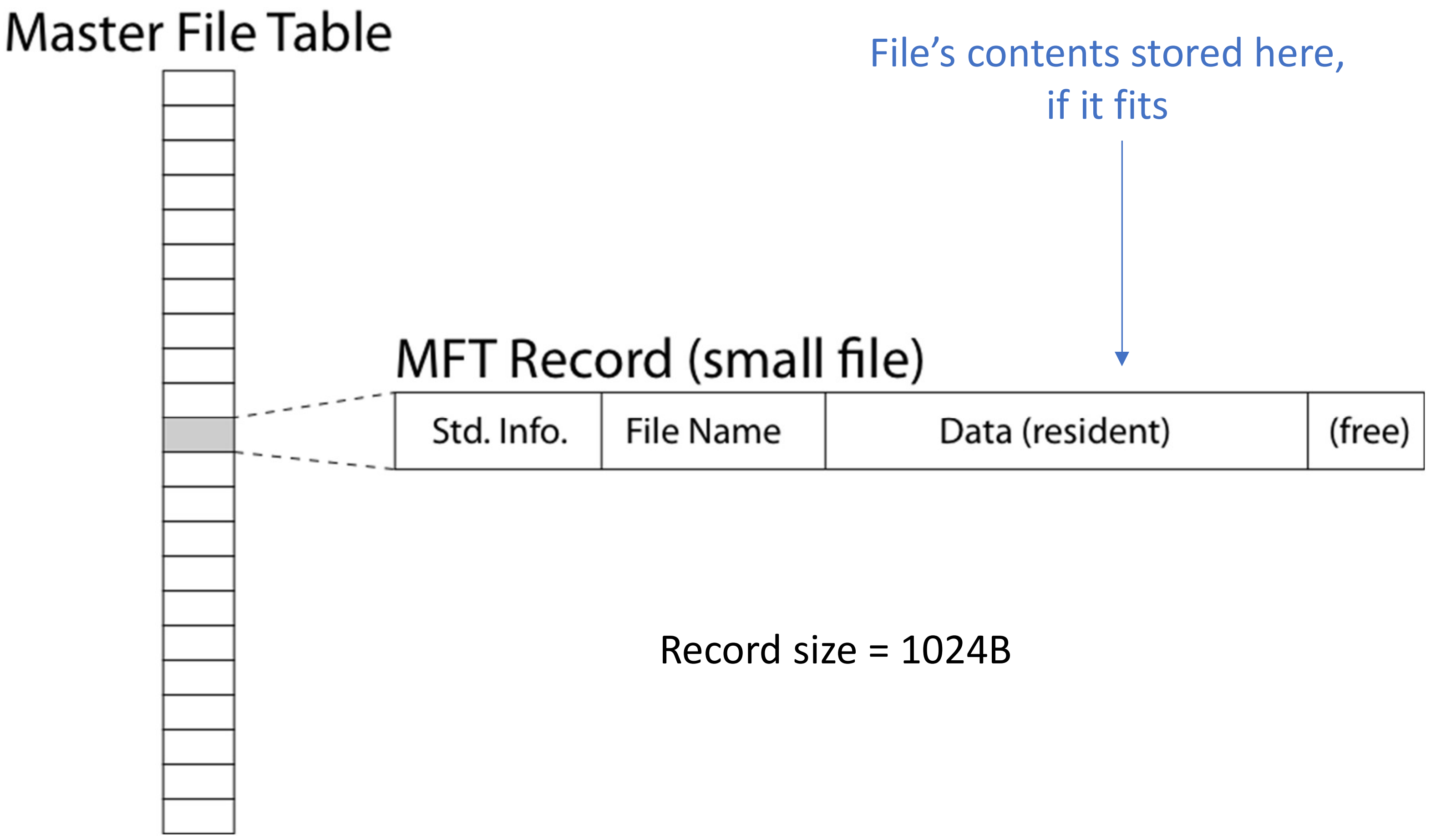
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on
    - Need to describe each block’s location, not [start, length]
  - Need to reserve 10-20% of free space to prevent fragmentation

  - *Later: terrible reliability characteristics when something unexpected happens*
    - *Can lose any amount of the file system on a crash*
    - *Trying to get it back is a slow and unreliable procedure*

NTFS

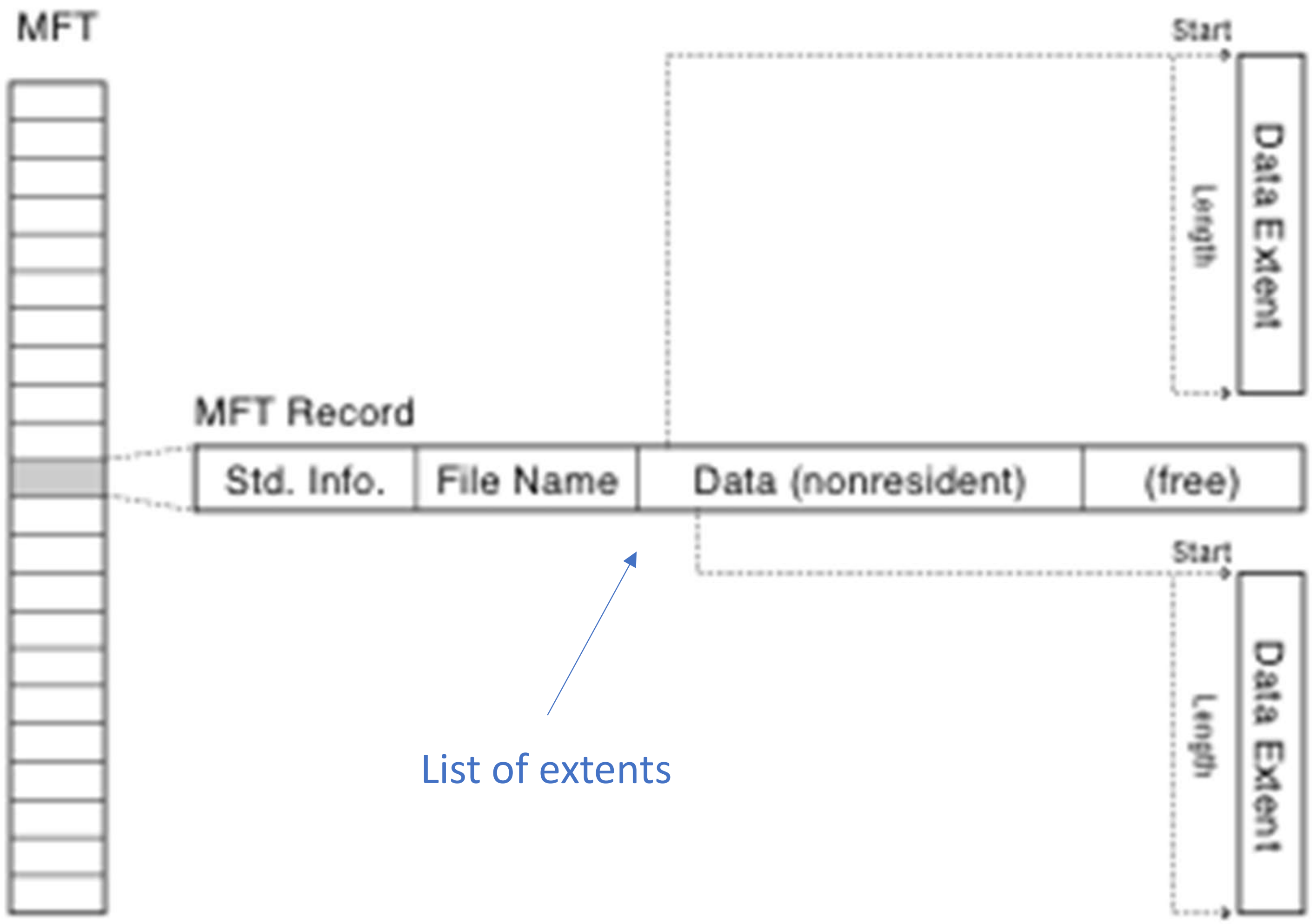
- Master File Table
  - Flexible 1KB storage for metadata and data
- Extents
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability
  - Next chapter / module

NTFS Small File



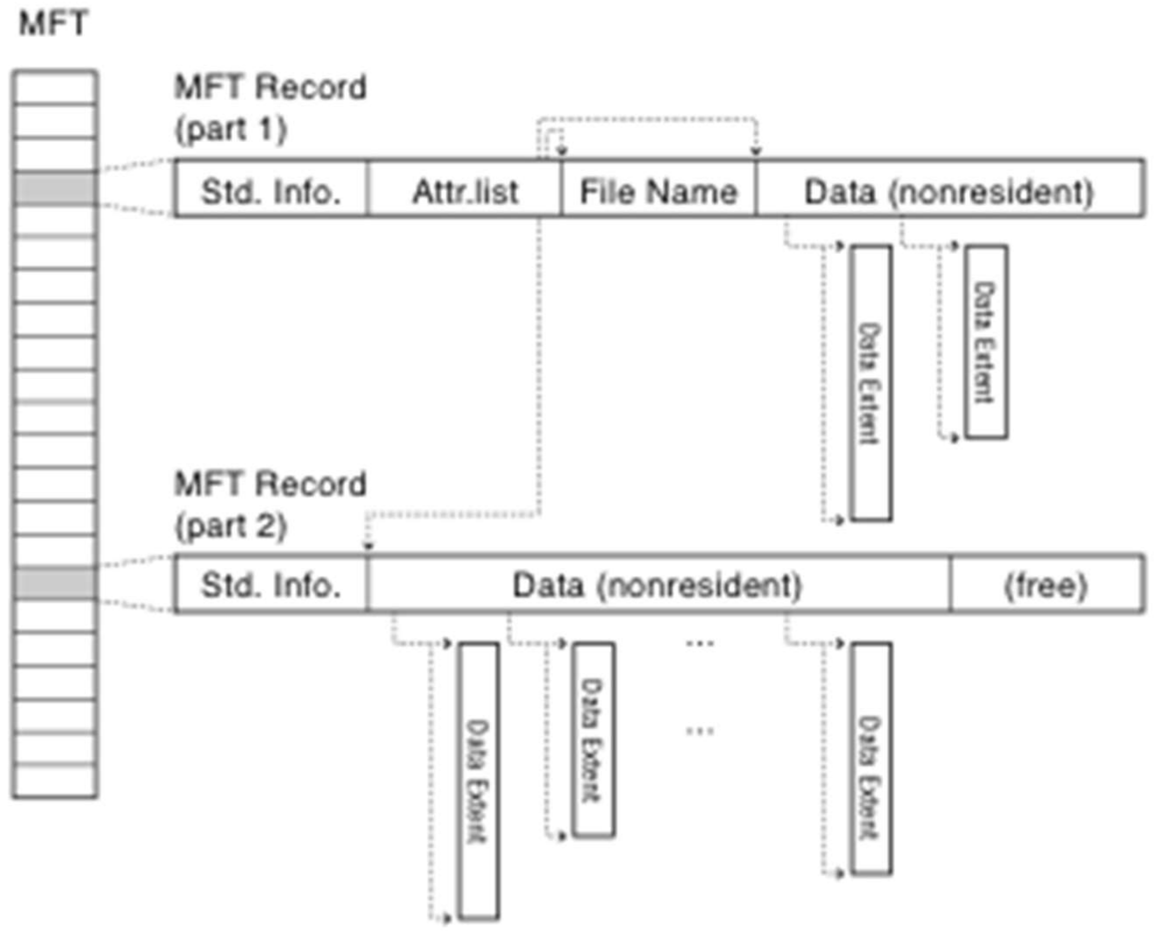
Record size = 1024B

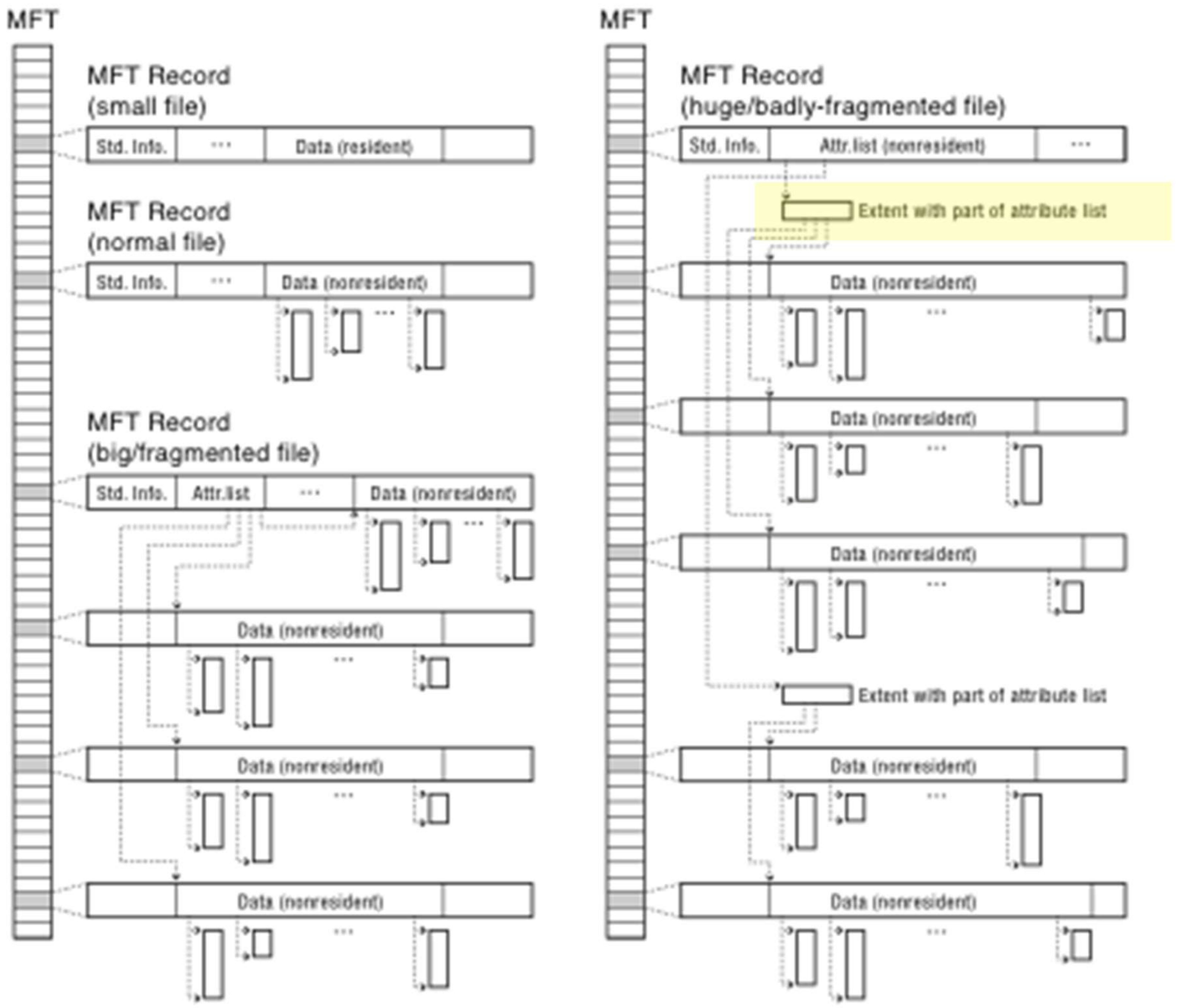
NTFS Medium-Sized File



List of extents

NTFS Indirect Block




MFT
MFT Record
(small file)
Std. Info.
Data (resident)
MFT Record
(normal file)
Std. Info.
Data (nonresident)
MFT Record
(big/fragmented file)
Std. Info.
Attr.list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Data (nonresident)
MFT
MFT Record
(huge/badly-fragmented file)
Std. Info.
Attr.list (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)

Summary

- File system decides how to make use of disk storage to represent the file system
  - directories
  - file data
  - file meta-data
- Layout has/had a big impact on performance on spinning disks
  - Try to achieve sequential read/write on physical device...
- Representation decisions limit maximum file size and maximum file system size
- So far we’ve considered only what we need to do when everything always works
  - It doesn’t...