File System Efficiency

Bringing It Together

- Virtual address spaces are mapped onto physical address spaces in physical memory
  - Pages are mapped to frames!
  - Lots of policies to consider: demand paging, page replacement, working set, …
- The heap is part of a process's virtual address space (and is thus paged!)
  - Can be explicitly or automatically managed
- I/O devices are important for system usability.
  - Each one consists of the device, the controller, a bus, and a port
  - One of the most important I/O devices is the disk, which provides stable storage.
- File systems make the disk usable
  - Goals include: efficiency, usability, reliability
  - File system metadata is in a superblock
  - A file system manages a disk partition

Goals in File System Design

- Efficiency (in space and time!)
  - Use disk space effectively
  - Optimize disk accesses
- Usability
  - Easy interface for application programmers
  - Easy interface for general computer users
- Reliability
  - Writes to stable storage can be interrupted at any time
  - Protecting user data from corruption is important

So…
We need to *efficiently* store the bits

File Storage: Considering Space and Time

File Layouts

Designing the File Layout

We need to:

- Store the bits
- Support sequential and random reads
- Maintain file location information
- Provide some reasonable amount of efficiency

Considering the typical file system profile:

- Most files are small
- Most disk space is consumed by large files
- I/O operations access both large and small files

File Data

File data: file contents

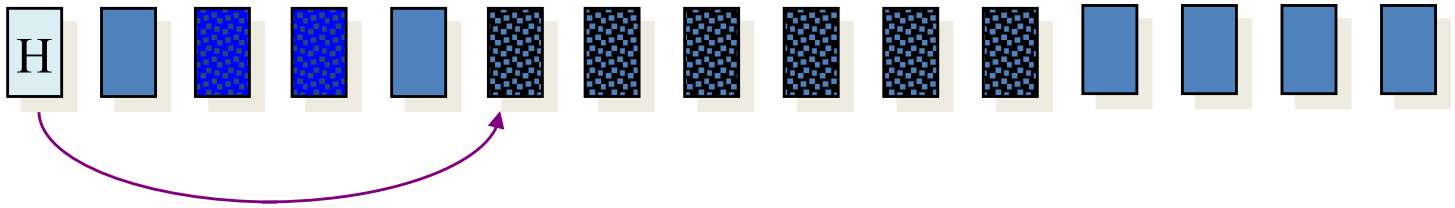
- The part that users actually care about
- Typically untyped bytes

Where is the file data?

How is the file data allocated on disk?

Contiguous Allocation

File system allocates a contiguous chunk of free blocks when it creates a file



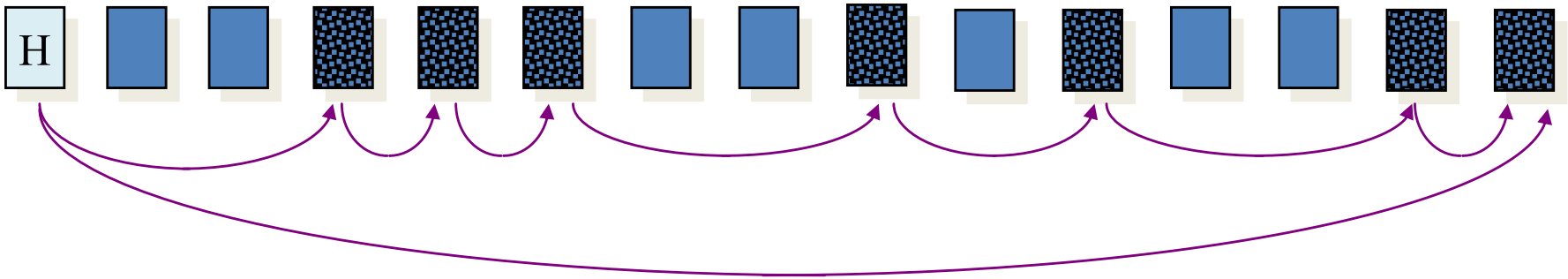
- Location information needed: start location and size
- Free space management: free list
- Allocation policies: *-fit!
- Evaluation:
  - Sequential access?
  - Random access?
  - File growth?
  - Fragmentation?

Contiguous Allocation: Text Description

- All file data stored contiguously on disk
- File header specifies starting block and length
- Placement/allocation policies
  - First-first, best-fit, worst-fit
- Best performance for the initial write of a file
  - Once space has been allocated
  - Later writes may cause the file to grow which would require it to be copied and moved

Linked Allocation

File stored as a linked list of blocks



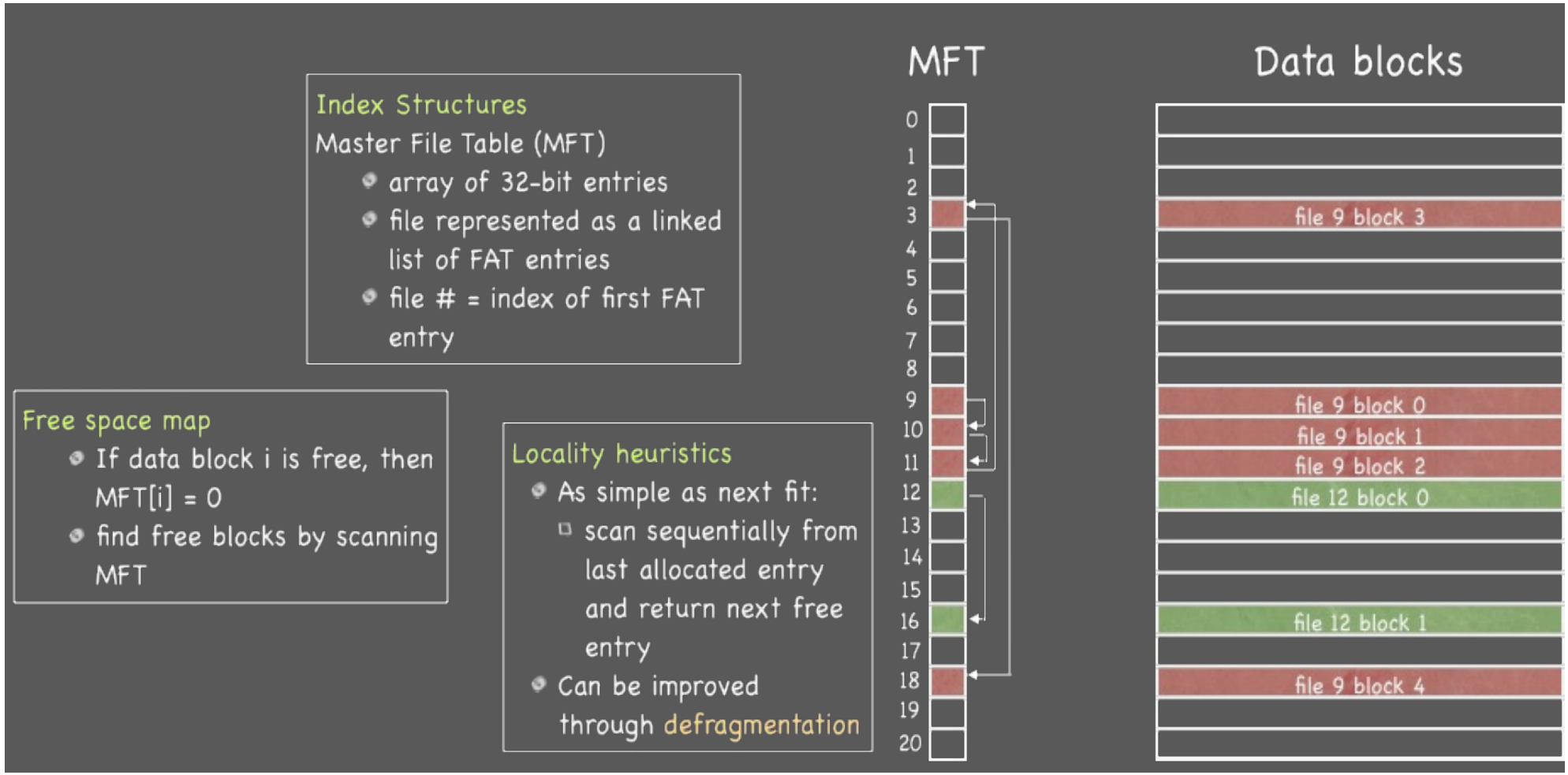
- Location information needed:
  - File header stores a pointer to the first block allocated
  - Each block, keeps a pointer to the next block
- Free space management: bitmap
- Allocation policies: on demand
- Evaluation:
  - Sequential access?
  - Random access?
  - File growth?
  - Fragmentation?

Is there a design tradeoff we could make?

Another Linked Design: FAT File System

File Allocation Table (FAT) stores linked list information

- Started with MS-DOS (Microsoft, late 70s)
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}-1 bytes
- Used in DVDs, flash drives, and Xbox consoles



Example: FAT File System Text Description

- File allocation table (FAT)
  - Started with MS-DOS (Microsoft, late 70s)
  - In FAT-32, supports 2 to the 28th blocks and files of 2 to the 32 - 1 bytes
- Parts:
  - Index structures - Master File Table (MFT)
    - Array of 32 bit entries
    - Each element in the array represents a data block in the system
    - File represented as an embedded linked list of the entries in the MFT
      - File number = index of first FAT entry, and also indicated first data block of file
      - That FAT entry will have number of next FAT entry/data block, which will have the next number, etc.
  - Free space map
    - If data block i is free, then MFT[i] = 0
    - Find free blocks by scanning over MTF
  - Locality heuristics
    - As simple as next fit
      - Scan sequentially from last allocated entry and return next free entry
    - Can be improved through defragmentation
      - Moving file data around so it is stored more contiguously on disk

FAT File System

Advantages

- Simple!

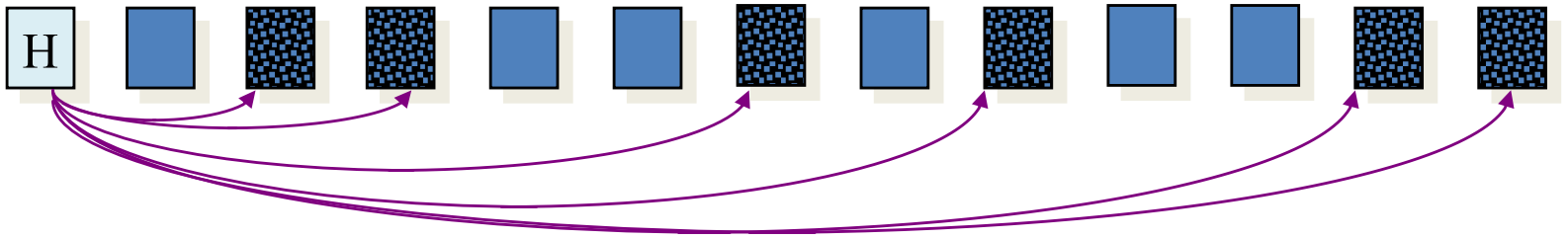
Disadvantages

- Poor random access
  - Requires sequential traversal
- Limited access control
  - No file owner or group ID metadata
  - Any user can read/write any file
- No support for hard links
  - Metadata stored in directory entry
- Volume and file size are limited
  - Example: FAT-32 entry is limited to 32 bits
    - If 4KB blocks, at most 2 TB volume
    - Individual files can be no larger than 4 GB
- No support for transactional updates (more later)

More File Layouts

Direct Allocation

File stored in individual blocks, location information in file header

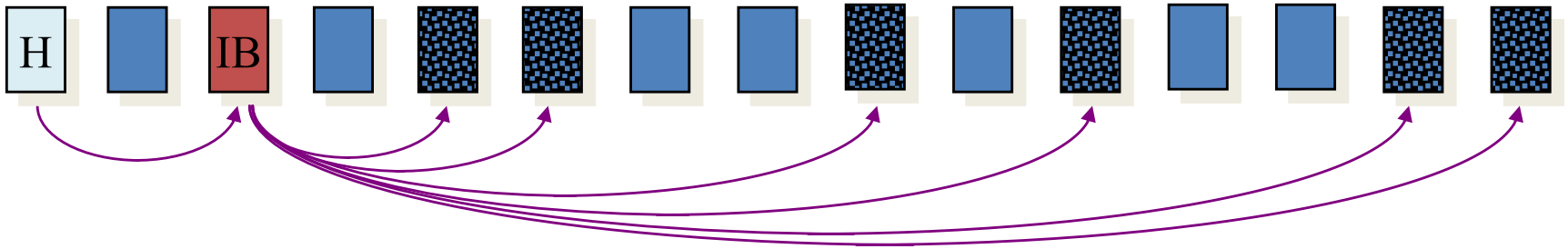


- Location information needed: file header points to each data block
- Free space management: bitmap
- Allocation policies: on demand
- Evaluation:
  - Sequential access?
  - Random access?
  - File growth?
  - Fragmentation?

What should we do about large files? What are our options?

Indexed Allocation

File stored in individual blocks, location information in index block

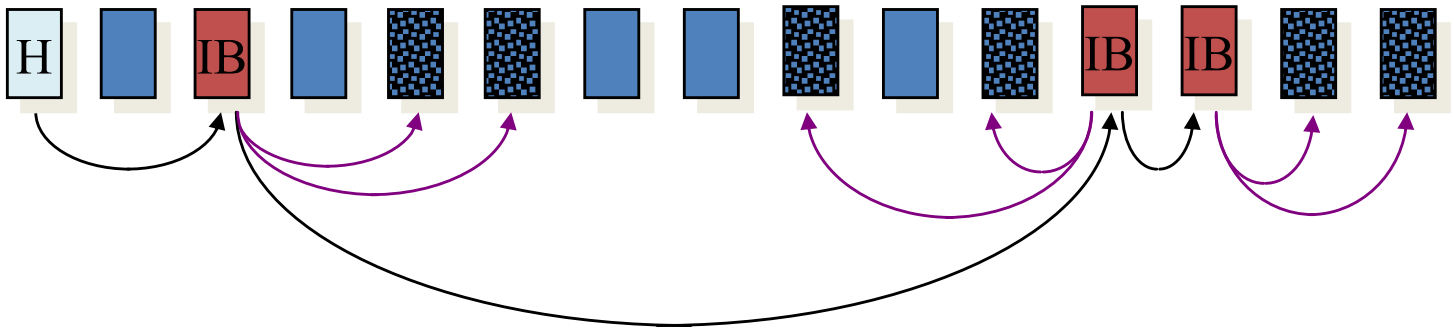


- Location information needed:
  - File header points to index block, a non-data block
  - Index block is an array of pointers to data blocks
- Free space management: bitmap
- Allocation policies: on demand
- Evaluation:
  - Sequential access?
  - Random access?
  - File growth?
  - Fragmentation?

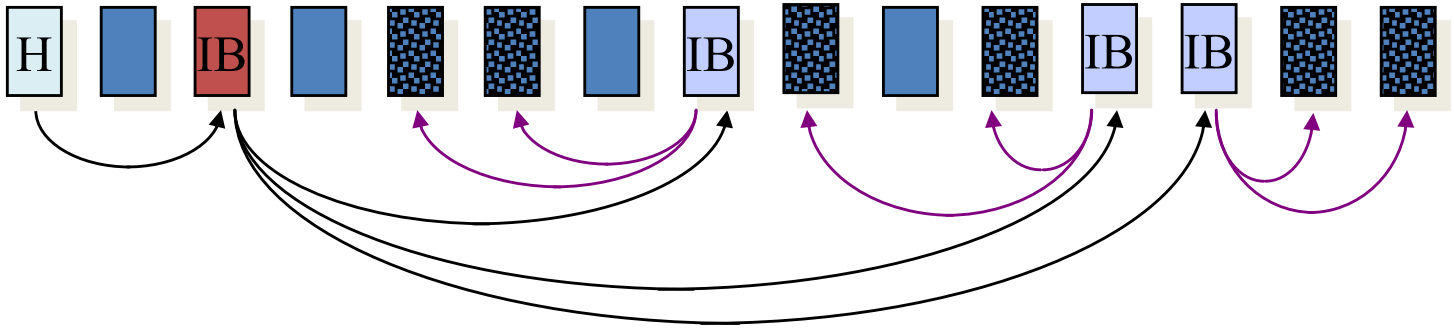
What should we do about large files? What are our options?

Indexed Allocation: Handling Large Files

- Linked index blocks (IB+IB+…)



- Multilevel index blocks (IB*IB*…)



Indexed Allocation: Handling Large Files Text Description

- Linked index blocks (IB + IB + …)
  - The file header points to an IB, and that IB points to data blocks and has a pointer to another IB and so on as the file grows
- Multi-level index blocks
  - Similar in structure to a multi-level page table
  - The file header points to an IB. This IB only contains pointers to other IBs, those other IBs are the ones that hold pointers to data blocks
  - Can grow in levels to support larger files

Question

Why would we want to add index blocks to direct allocation?

A. Allows greater file size
B. Faster to create files
C. Simpler to grow files
D. Simpler to prepend and append to files

Multilevel Indexed Files and FFS

Multilevel Indexed Files

File stored as a fixed, asymmetric tree, with data blocks as its leaves

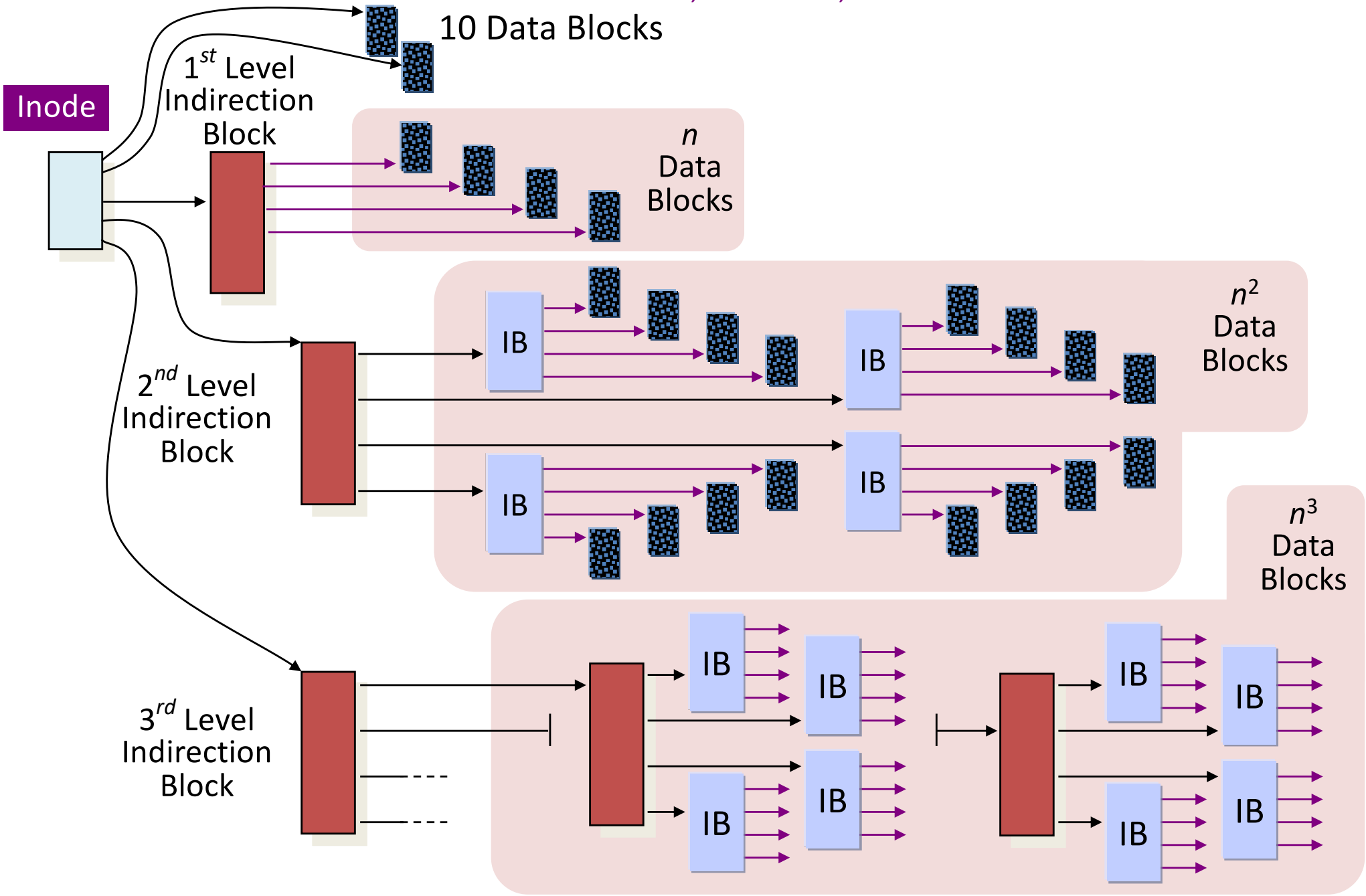
- Location information needed:
  - Direct pointers to data blocks
  - Indirect pointers to index blocks of varying levels of indirection
- Free space management: bitmap
- Allocation policies: on demand
- Evaluation:
  - Sequential access?
  - Random access?
  - File growth?
  - Fragmentation?

Example: Fast File System

- Implemented by UNIX in the 80s
- Each file header contains 13 block pointers.
  - First 10 pointers point to data blocks
  - 11^{th} pointer points to a block of 1024 pointers to 1024 more data blocks (One indirection).
  - 12^{th} pointer points to a block of pointers to indirect blocks (Two indirections).
  - 13^{th} pointer points to a block of pointers to blocks of pointers to indirect blocks (Three indirections).
- File headers are called *inodes.*

Fast File System File Allocation

Multilevel, indirection, index blocks



Indexed Allocation in UNIX: Text Description

- The inode holds 4 different kinds of pointers in addition to metadata
  - Direct pointers - point directly to data blocks from inode
    - usually the first 10 pointers in the inode
  - 1st level indirection block
    - inode points to 1st IB and it points to data blocks
    - would hold n data blocks
  - 2nd level indirection block
    - inode points to a block full of pointers to IBs
    - would hold n^2 data blocks
  - 3rd level indirection block
    - inode points to block full of pointers to 2^{nd} level indirection blocks, and each of those is full of pointers to IBs
    - would hold n^3 data blocks
- In total structure holds: $10 + n + n^2 + n^3$ blocks

Multilevel Indexed: Key Ideas

- Tree structure
  - efficient in finding blocks
- Efficient in sequential reads
  - once an indirect block is read, can read 100s of data blocks
- Fixed structure
  - simple to implement
- Asymmetric
  - efficiently supports files big and small
- Adaptable design to various contexts

Example: BigFS

- 4KB blocks, 8 byte pointers
- An inode stores
  - 12 direct pointers
  - 1 indirect pointer
  - 1 double indirect pointer
  - 1 triple indirect pointer
  - 1 quadruple indirect pointer

Total possible file size:

$12*4KB + 512*4KB + 512^2*4KB+512^3*4KB + 512^4*4KB = 256.5TB$

Question

If my file system only has lots of big video files, what block size do I want?

A. Large
B. Small

What else might be in the file header?

- File metadata:
  - Owner id
  - Size
  - Permissions
  - Last modified time
  - Location of all data blocks
- Each file has its own metadata and its own file header
- File headers are stored at a location known by the OS for easy access
  - Typically an array of fixed-size file headers

Okay, but…

So… a few questions:

1) How does the OS communicate with these devices, which need input and output data, but are much slower than the CPU? Does it do something to hide latency?
2) Does the I/O subsystem to anything to hide latency?
3) What about the file system? Does it anything to hide latency?

Mechanical Sympathy

Mechanical Sympathy

- Coined by Jackie Stewart (F1 driver)
  - To be a great race car driver you must have “mechanical sympathy” for HOW a car works to get the best out of it
- Co-opted for tech (Martin Thompson)
- System design reflects hardware
  - Very apparent in FSes
  - Frequent change in dominant medium





FFS demonstrates *mechanical sympathy*

Anatomy of a Magnetic Disk

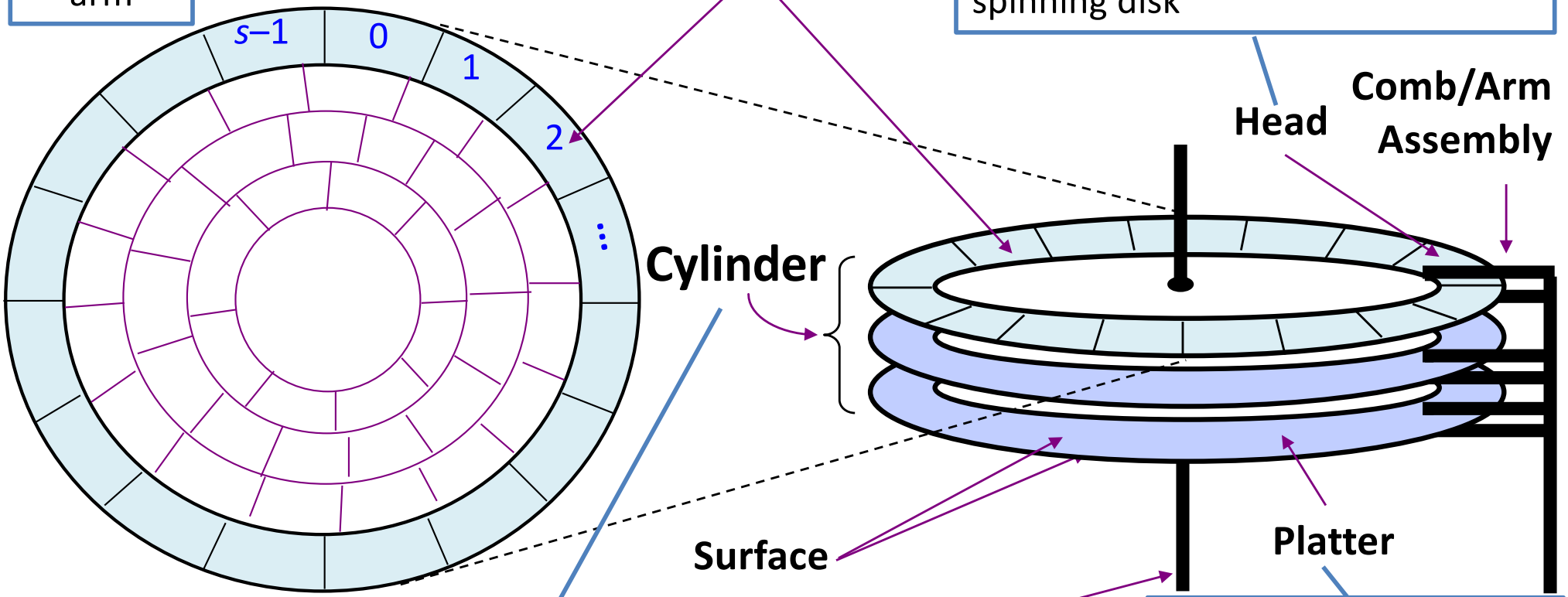
data on a track can be read without moving arm

Typically 512 bytes

Reads by sensing a magnetic field

Writes by creating one

Floats on air cushion created by spinning disk



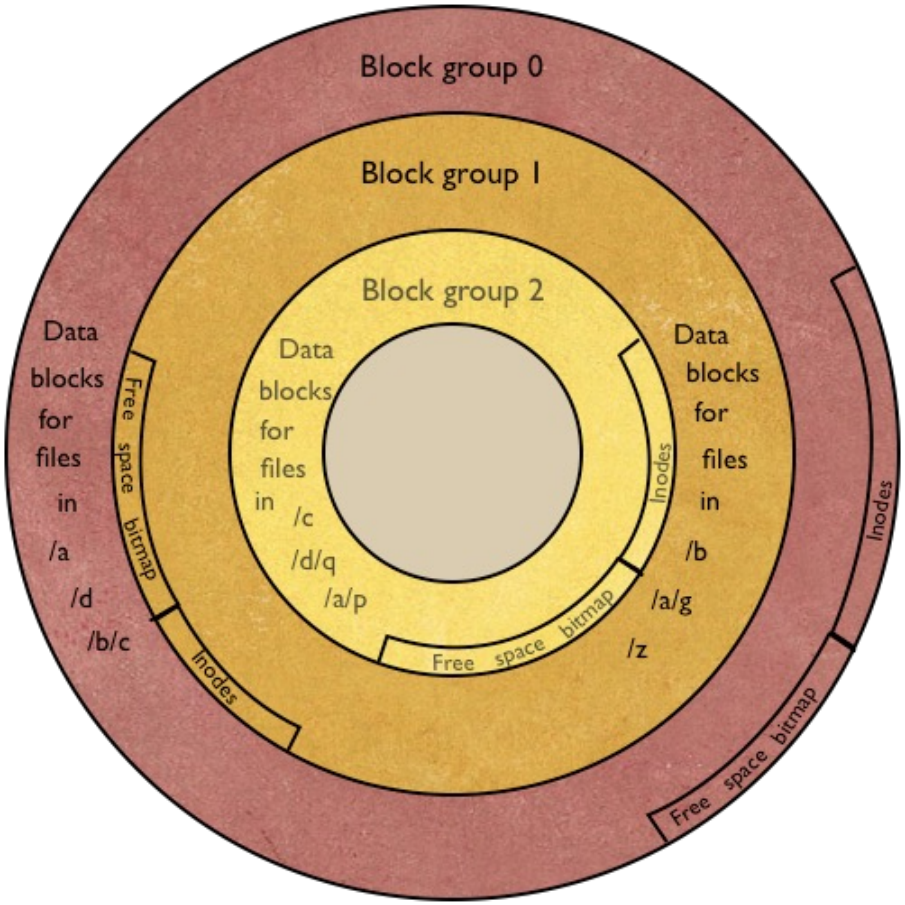
Set of tracks on different surfaces with same track index

4,200-15,000 RPM

Thin cylinder that holds magnetic material

Each platter has two surfaces

FFS Locality Heuristics: Block Groups



- Divide partition into block groups
  - Sets of nearby tracks
- Distribute metadata
  - Old design: free space bitmap and inode map in a single contiguous region
    - Lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups
- Place file in block group
  - When a new file is created, FFS looks for inodes in the same block as the file's directory
  - When a new directory is created, FFS places it in a different block from the parent's directory
- Place data blocks
  - First-free heuristics
  - Trade short-term for long-term locality

Example: 500 Random Reads

Size	
Platters/Heads	2/4
Capacity	320 GB
Performance	
Spindle Speed	7200 RPM
Average seek time	10.5/12.0 ms
Maximum seek time	19 ms
Track-to-track	1ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer Memory	16MB
Power	
Typical	16.35W
Idle	11.68W

- Workload
  - 500 read requests
  - Randomly chosen sector
  - Served in FIFO order
- How long to service them?
  - seek time: 10.5ms (avg)
  - rotation time
    - 7200 RPM = 120 RPS
    - Rotation time 8.3ms
    - On average, half of that: 4.15ms
  - transfer time
    - at least 54 MB/s
    - 512 bytes transferred in 9.25 microseconds
  - Total time:
    - 500 x (10.5 + 4.15 + .009) = 7.33 sec

Example: 500 Sequential Reads

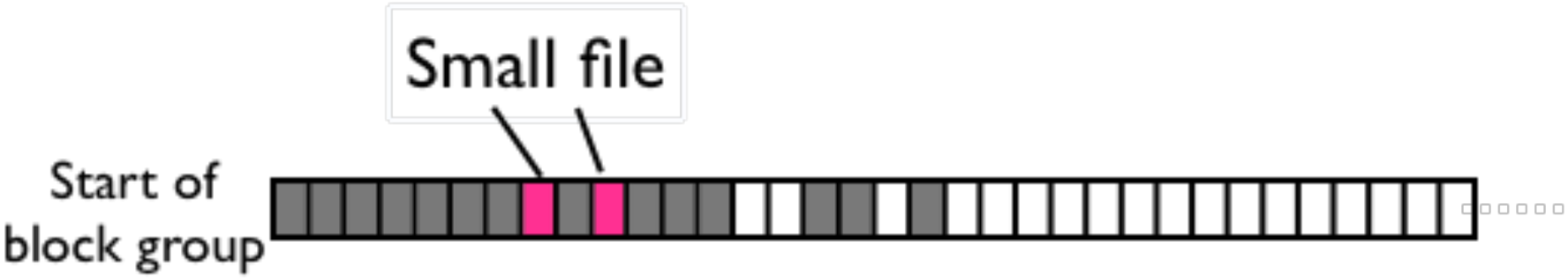
Size	
Platters/Heads	2/4
Capacity	320 GB
Performance	
Spindle Speed	7200 RPM
Average seek time	10.5/12.0 ms
Maximum seek time	19 ms
Track-to-track	1ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer Memory	16MB
Power	
Typical	16.35W
Idle	11.68W

- Workload
  - 500 read requests
  - Sequential sectors on the same track
  - Served in FIFO order
- How long to service them?
  - seek time: 10.5ms (avg)
  - rotation time
    - On average, half of that: 4.15ms
    - Same as before
  - transfer time
    - outer track: 500 x (.5/128000) = 2ms
    - inner track: 500 x (.5/54000) = 4.6ms
  - Total time:
    - outer track: 2 + 4.15 + 10.5 = 16.65ms
    - inner track: 4.6 + 4.15 + 10.5 = 19.25ms

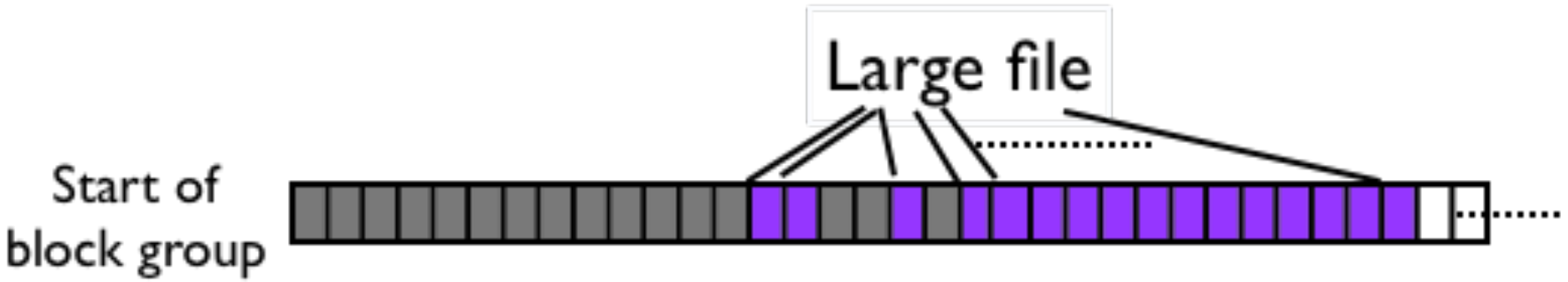
FFS Locality Heuristics: File Placement



FFS Locality Heuristics: File Placement



FFS Locality Heuristics: File Placement



FFS Locality Heuristics: Reserved Space

- When a disk is close to full, hard to optimize locality
  - file may end up scattered through disk
- FFS presents applications with a smaller disk
  - about 10% smaller
  - user write that encroaches on reserved space fails

File System Buffer Cache

- The file system also hides disk latency by reducing the accesses to the disk
- It caches current and recently used data and metadata structures
- It caches the root directory contents
- It buffers writes to disk
  - As in, it may not send them to disk immediately, but it will return success to the application anyway

Okay, so that’s Linux/Unix…

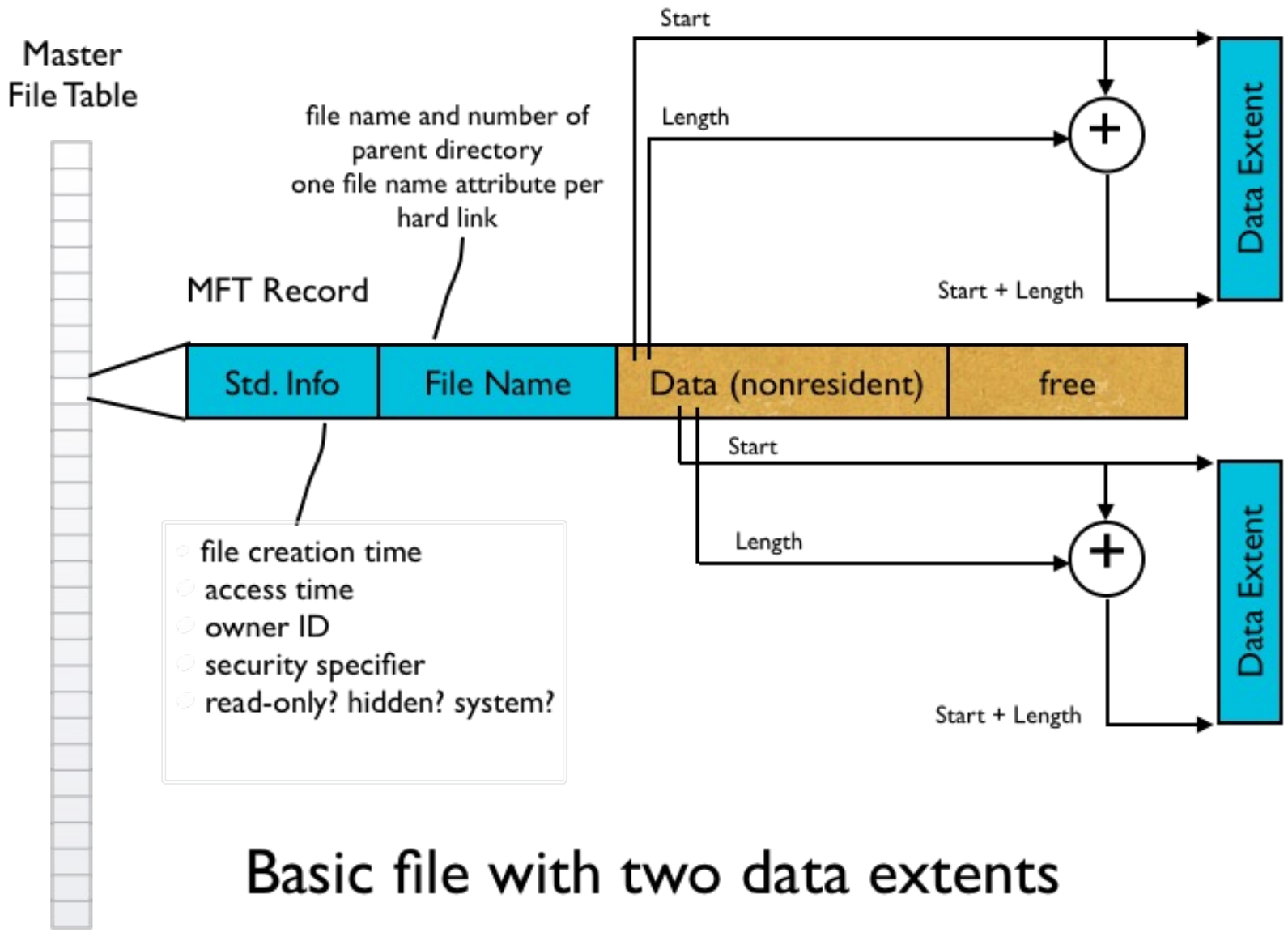
What about Windows?

NTFS

NTFS

- Default Windows file system
- Introduced by Microsoft 1993
- Uses *extents* with *flexible trees*
  - Extents: file data is stored in ranges of contiguous blocks
  - Flexible tree
    - Each file is represented by variable depth tree
    - Large file with few extents can be stored in a shallow tree
- A Master File Table (MFT) stores the *file records*
  - File records hold file metadata (similar to the inode)
  - Each record stores sequence of variable-sized attribute records

Example of NTFS

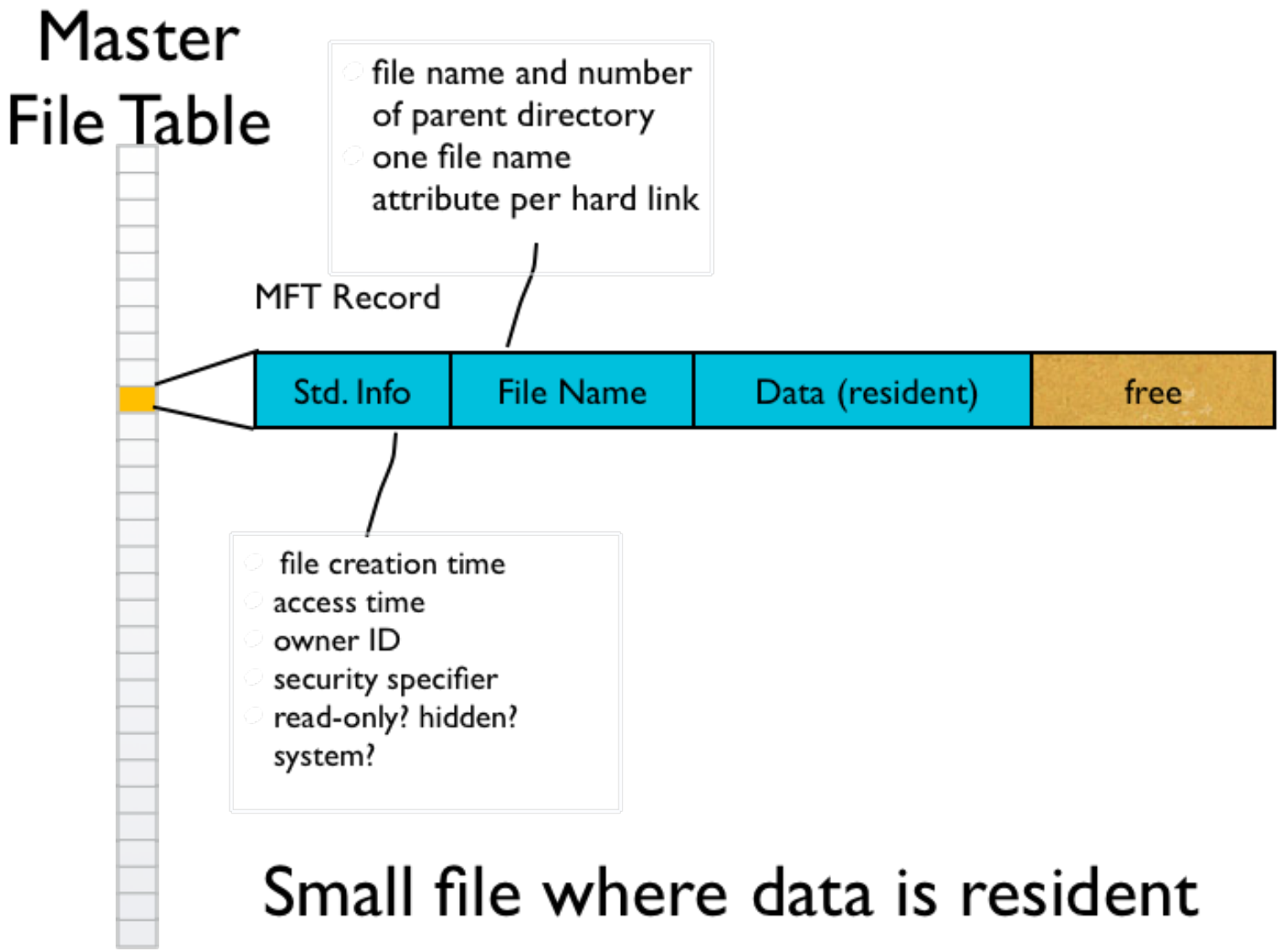


Basic file with two data extents

Example of NTFS: Text Description

- Basic file with 2 data extents
- Has a master file table (MFT) that holds all the file headers, which are called MFT records
- MFT record components
  - Standard info
    - File creation time
    - Access time
    - Owner ID
    - Security specifier
    - Read-only? Hidden? System?
    - File name and number of parent directory
  - File name
    - One file name attribute per hard link
  - Information about 2 extents
    - Start of extent and its length
    - Extents contain file data
  - May have leftover space

Example of NTFS

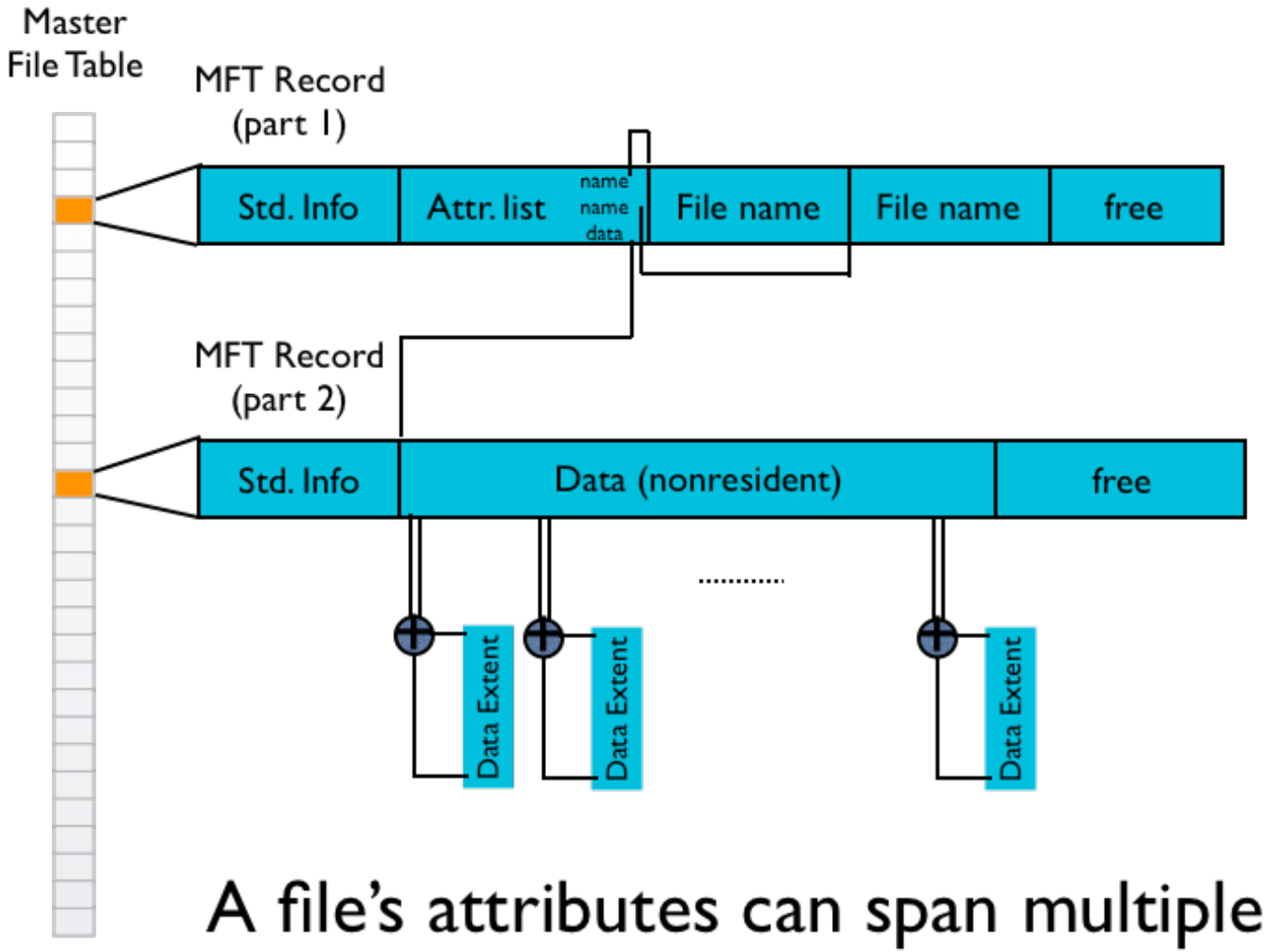


Small file where data is resident

Example of NTFS: Text Description

- Small file where data is resident
  - Resident means the data is stored directly in the file record
- MFT record still has the other information components
  - standard info, file name, data and free space

Example of NTFS

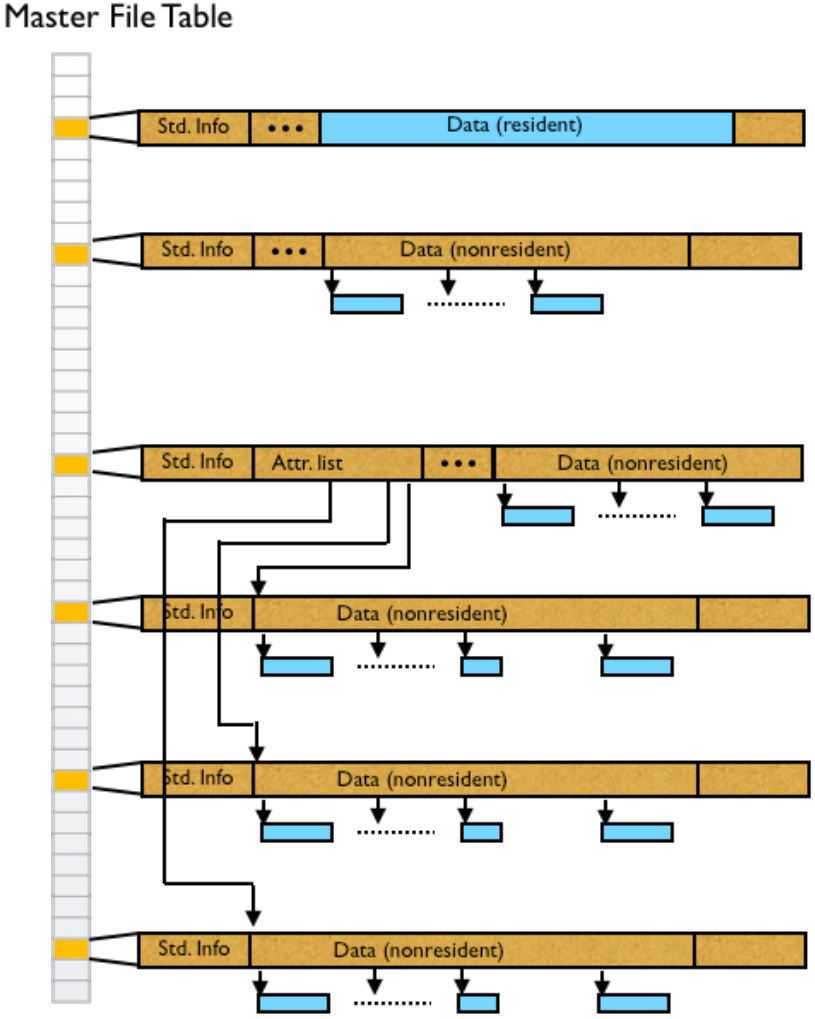


A file's attributes can span multiple records

Example of NTFS: Text Description

- A file's attributes can span multiple records, if necessary
  - For this example:
    - The first record holds an attribute list that points to the second record
    - Second record holds the pointers to extents

Small, normal, and big files



...and for really huge (or badly fragmented) files, even the attribute list can become nonresident

Small, Normal, and Big File NTFS Examples: Text Description

- Small file has data resident in MFT record
- Medium (or non-fragmented) file has a single MFT record with pointers to extents
- Large (or badly fragmented) files can span many MFT records
  - Each record has a pointers to extents until it is full
  - Also includes pointer to next record
- And for really huge (or badly fragmented) files, even the attribute list can become nonresident
  - File can span many many MFT records

NTFS: Metadata Files

- NTFS stores most metadata in ordinary files with well-known numbers
  - 5 (root directory); 6 (free space bitmap); 8 (list of bad blocks)
- $Secure (file no. 9)
  - Stores access control list for every file
  - Indexed by fixed-length key
  - Files store appropriate key in their MFT record
- $MFT (file no. 0)
  - Stores Master File Table
  - To read MFT, need to know first entry of MFT
    - a pointer to it stored in first sector of NTFS
  - MFT can start small and grow dynamically
  - To avoid fragmentation, NTFS reserves part of start of volume for MFT expansion

NTFS: Locality Heuristics

Best fit

- Finds smallest region large enough to fit file
- NTFS caches allocation status for a small area of disk
  - Writes that occur together in time get clustered together
- SetEndOfFile() lets users specify expected length of file at creation

NTFS: Summary

- Uses a lot of structures similar to what we've seen before
  - A single array for file headers
  - File records contain data block location information
- And a lot of cool stuff we haven't seen before!
  - Flexible use of space in file records
  - Flexible use of file records
  - Uses extents to more compactly store which data blocks belong to a file

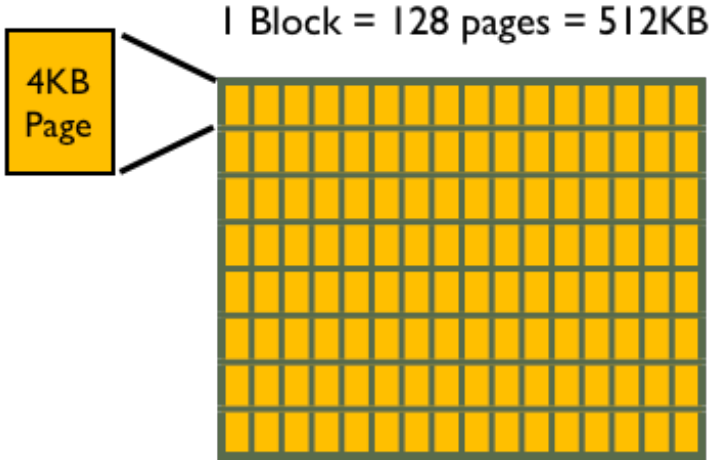
So we've seen mechanical sympathy in relation to spinning disks… what about SSDs?

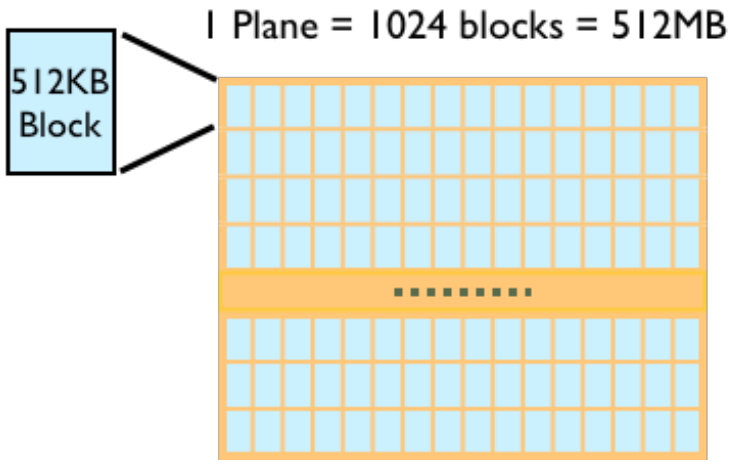
SSDs

Solid State Drives

- A form of stable storage
- No moving parts, so they:
  - Have better performance than spinning disks
  - Use less power
  - Are more resistant to physical damage
- Typically implemented using NAND
  - Cheap (largely because they are popular)
  - Some durability concerns

NAND Flash Units





- Pages (typically 4KB) make up blocks, which make up planes
  - Reads and writes *only* occur in page units
- Reads and write are fast
  - Read page (tens of µs)
  - Write page (tens of µs)
- But, pages can only be written if they are empty, and pages must be erased (emptied) a block at a time
  - Erasing a *block* takes several ms
  - Sets all bits in the block to 1
- Use remapping to reduce cost of writes
  - More in just a second
- Flash devices can have multiple independent data paths
  - Can issue multiple concurrent requests to maximize bandwidth

Remapping Flash Drives

- In remapping, the SSD controller writes pages (either new or updates to existing) to a block that already has a free space (if possible), rather than in-place updates

Consider a flash drive with the following specs:

4 KB page

3ms erase

512KB erasure block (128 pages)

50µs read page/write page

- How long to naively read/erase/and write a single page?

  $128 \times (50 \times 10^{-3} + 50 \times 10^{-3}) + 3 = 15.8\text{ms}$

- Suppose we use remapping, and we always have a free erasure block available. How long now?

  $3/128 + 50 \times 10^{-3} + 50 \times 10^{-3} = 123.4\mu\text{s}$

  We get to amortize the cost of the erase over all the writes in the block.

What does remapping imply for addresses on an SSD?

SSD Durability

- Flash storage stops reliably storing a bit
  - after many erasures (on the order of 10^3 to 10^6)
  - after a few years without power
  - after nearby cell is read many times (read disturb)
- To improve durability, the SSD controller uses many tricks:
  - Error correcting codes for data
  - Marks blocks as defective once the no longer erase correctly, pages as they no longer reliably store data
  - Wear-level the drive by spreading updates to hot logical pages to many physical pages
  - Reserving space: both pages and erasure blocks
    - for both wear leveling and managing bad pages and blocks

Example: Intel 710 series Solid State Drive

Consider 500 read requests to randomly chosen pages. How long will servicing them take?

- 500 x 26µs = 13ms
  - spinning disk: 7.3s

How do random and sequential read performance compare?

- effective random bw:
  - (500 x 4)KB / 13ms ≈ 154MB/s
- ratio: 154/270 = 57%

500 random writes

- 500s/2000 = 250ms

How do random and sequential writes compare?

- effective bw random
  - (500 x 4)KB / 250ms = 8MB/s
- ratio: 8/210 = 3.8%

Size	
Capacity	300 GB
Page Size	4KB
Performance	
Bandwidth (seq reads)	270 MB/s
Bandwidth (seq writes)	210 MB/s
Read/Write Latency	75µs
Random reads/sec	38,500 (one every 26 µs)
Random writes/sec	2,000 (2,400 with 20% space reserve)
Interface	SATA 3GB/s
Buffer Memory	16MB
Endurance	
Endurance	1.1 PB (1.5 PB with 20% space reserve)
Power	
Active	3.7W
Idle	0.7W

SSDs and Writes

- SSDs are so fast any scheduling algorithm slows down the writes
- So SSD writes occur in no order
  - *Not* FIFO order, since FIFO means the first write issued is the first one completed. That's too rigid for this use case.

Mechanical Sympathy and SSDs

- Today's file systems have adapted to SSDs
- Most use extents
  - How does this help on an SSD?

Copy-on-Write File Systems

- Data and metadata not updated in place, but written to new location
  - Transforms random writes to sequential writes
- Several motivations
  - Small writes are expensive
  - Caches filter reads
  - Widespread adoption of flash storage
    - Wear leveling, which spreads writes across all cells, important to maximize flash life
    - COW techniques used to virtualize block addresses and redirect writes to cleared erasure blocks
  - Large capacities enable versioning