Disks and File System Fundamentals: Use, API, and Implementation

CS439: Principles of Computer Systems

March 30, 2015

Last Time

- I/O Interfaces
  - Polling
  - Interrupts
  - DMA
- Disks
  - How they work
  - Scheduling Algorithms
  - Optimizations

Today's Agenda

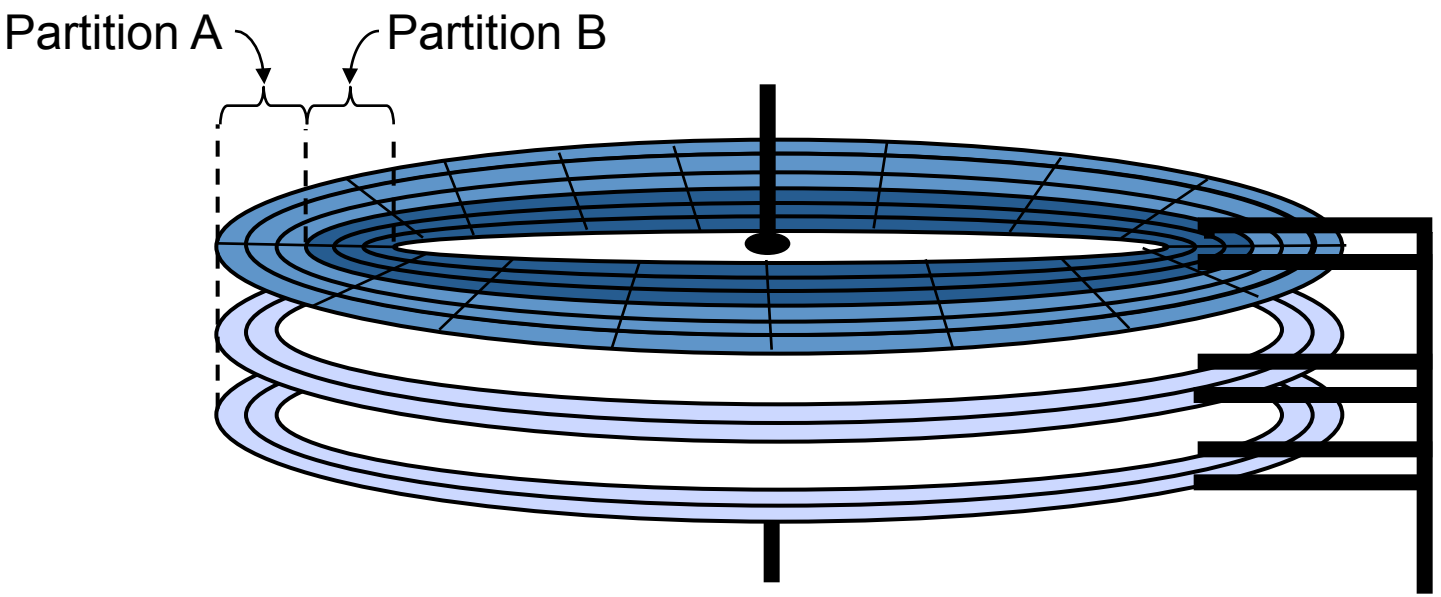
- Solid State Drives
- File System Use
  - system calls (ish) to create, remove, open, read, write, and close files
- File System Implementation
  - What is a file?
    - How are files represented?
    - How are files organized?

Disks

Other Ways to Improve Performance: Partitioning

Disks are typically partitioned to minimize the largest possible seek time

- A partition is a collection of cylinders
- Each partition is a logically separate disk



Other Ways to Improve Performance: Interleaving

- *Problem:* Contiguous allocation of files on disk blocks only makes sense if the OS can react to one disk response and issue the next command before the disk spins past the next block
- *Idea*: Interleaving. Don't allocate blocks that are physically contiguous, but those that are temporally contiguous relative to the speed with which a second disk request can be received and the rotational speed of the disk

Other Ways to Improve Performance: Buffering

- Read blocks from the disk ahead of the user's request and place in buffer on the disk controller
  - Read those that are spinning by the head anyway
- Reduces the number of seeks
- Avoids "shoulda coulda"
  - Exploits locality

Reducing Overhead Summary

To get the quickest disk response time, we must minimize seek time and rotational latency:

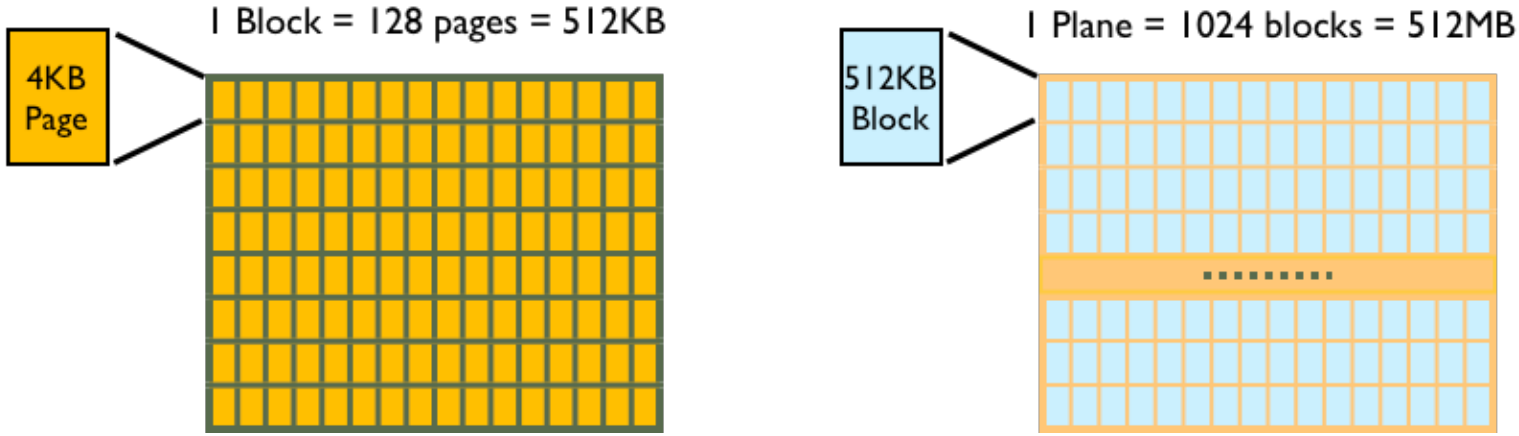
- Make disks smaller
- Spin disks faster
- Schedule disk operations to minimize head movement
- Lay out data on disk so that related data are on nearby tracks
- Place commonly used files where? on disk
- We should also pick our block size carefully:
  - If block size is too small, we will have low transfer rate because we need to perform more seeks for same amount of data
  - If block size is too large, we will have lots of internal fragmentation

Another Way to Improve Performance: Flash Storage

No moving parts

- Better random access performance
- Less power
- More resistant to physical damage

NAND Flash Units



- Operations
  - Erase block
    - Before a page can be written, needs to be set to logical “1”
    - Erases must occur in block units
    - Operation takes several ms
  - Write page
    - tens of µs
  - Read page
    - tens of µs
  - Reads and writes *only* occur in page units
- Flash devices can have multiple independent data paths
  - OS can issue multiple concurrent requests to maximize bandwidth

NAND Flash Units: Text Description

- 1 block is made of 128 pages is 512 KB
  - Each page is 4 KB
  - Note that block and page in this context have different meanings than a page in the virtual memory context or a block in the spinning disk context
- 1 plane is 1024 blocks, so it holds 512 MB
- Operations
  - Erase block
    - Before it can be written, needs to be set to logical “1”
    - Erases must occur in block units
    - Operation takes several ms
  - Write page
    - takes tens of microseconds
  - Read page
    - takes tens of microseconds as well
  - Reads and writes *only* occur in page units
- Flash devices can have multiple independent data paths
  - OS can issue multiple concurrent requests to maximize bandwidth

Example: Remapping Flash Drives

- Flash drive 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?
  - $3/128 + 50 \times 10^{-3} = 73.4\mu\text{s}$

Flash Durability

- Flash memory 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
  - error correcting codes
    - extra bytes in every page
  - management of defective pages/erasure blocks
    - firmware marks them as bad
    - wear leveling
      - spreads updates to hot logical pages to many physical pages
    - spares (pages and erasure blocks)
      - for both wear leveling and managing bad pages and blocks

series
Solid State Drive

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

- 500 x 26µs = 13ms
  - spinning disk: 7.8s

How do random and sequential read performance compare?

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

500 random writes

- 500s/2000 = 250ms

How do random and sequential write 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

Example: Intel 710 Series Solid State Drive
Plain Text

- Drive Metrics
  - Size metrics
    - 300 GB capacity
    - Page size is 4 KB
  - Performance Metrics
    - Bandwidth for sequential reads is 720 MB/s
    - Bandwidth for sequential writes is 210 MB/s
    - Read/Write latency is 75 microseconds
    - Random reads/sec is 38,500, so one every 26 microseconds
    - Random writes/sec is 2,000, can be 2,400 with 20% space reserve
    - Interface is SATA 3 GB/s
    - Buffer memory is 16 MB in size
  - Endurance Metrics
    - Endurance is 1.1 PB (1.5 PB with 20% space reserve)
  - Power Metrics
    - Active is 3.7 W
    - Idle is 0.7 W
- Consider 500 read requests to randomly chosen pages. How long will servicing them take?
  - 500 * 26 microseconds = 13 ms
    - spinning disk would be 7.8 s
- How do random and sequential read performances compare?
  - Effective bw random
    - (500 * 4)KB / 13 ms is about 154 MB/s
  - Ratio: 154/ 270 = 57%
- 500 random writes
  - 500s/ 2000 = 250 ms
- How do random and sequential write compare?
  - Effective bw random
    - (500 * 4)KB / 250 ms = 8 MB/s
  - Ratio: 8/ 210 = 3.8 %

Spinning Disk vs. Flash

Metric	Spinning Disk	Flash
Capacity/Cost	Excellent	Good
SequentialBW/ Cost	Good	Good
Random I/O per sec/Cost	Poor	Good
Power Consumption	Fair	Good
Physical Size	Good	Excellent

Spinning Disk vs. Flash: Plain Text

- Spinning Disk
  - Capacity/Cost ratio is excellent
  - Sequential BW/Cost ratio is good
  - Random I/O per sec/Cost ratio is poor
  - Power consumption is fair
  - Physical size is good
- Flash
  - Capacity/Cost ratio is good
  - Sequential BW/Cost ratio is good
  - Random I/O per sec/Cost ratio is good
  - Power consumption is good
  - Physical size is excellent

Disk Summary

- Disks are slow devices relative to CPUs
- Spinning disks:
  - made up of sectors, tracks, cylinders, platters
  - Read/Write overhead = seek time+rotational delay +transfer time
    - Seek time is the largest cost
    - disk head scheduling algorithms to minimize it
  - We can also improve disk performance using partitions, interleaving, buffering
- Solid State drives:
  - Much faster
  - Use less power
  - Have durability issues

File System Use and API

The File System Abstraction

- Presents applications with **persistent**, **named** data
- A **file** is a named collection of related information recorded on secondary storage. Has two parts:
  - Data: what a user or application puts in the file
    - array of untyped bytes (In the end, it's all just bits!)
  - Metadata: file attributes added and managed by the OS
    - Name, type, location, size, creator, creation time, security info, modification time
  - File operations include create, open, read, write, seek, delete ….
- A directory provides names for files
  - list of human readable names
  - a mapping from each name to a specific underlying file or directory (hard link). [A soft link is a mapping from a file name to another file name]

User Interface: Common Operations

- Create and delete files
  - create(), link(), unlink()
- Open and close files
  - open(), close()
- File access
  - read(), write(), seek(), fsync()

create()

Creates a new file with some metadata and a name for the file in a directory

On create(), the OS:

- Allocates disk space (checks disk quotas, permissions, etc.)
- Creates metadata for the file including name, location on disk, and all file attributes
- Adds an entry to the directory that contains the file

link()

Creates a hard link---a new name for some underlying file

On link(), the OS:

- Adds entry to the directory with new name
- Increments counter in file header that tracks the number of directory entries pointing to that file
- Will point to same underlying file

unlink()

Removes a name for a file from its directory. If last link, file and its resources are deleted.

On delete(), the OS:

- Finds directory containing the file
- Clears headers
- Frees the disk blocks used by the file and its headers
- Removes the entry from the directory

open()

Creates per file data structure referred to by file descriptor. Returns file descriptor to the caller.

On open(), the OS:

- Checks if the file is already open by another process. If not:
  - Finds the file
  - Copies the file descriptor into the system-wide open file table
- Checks the protection of the file against the requested mode. If not okay, aborts the open()
- Increments the open count
- Creates an entry in the process's file table pointing to the entry in the system-wide file table.
- Initializes the current file pointer to the start of the file
- Returns the index into the process's file table.
  - Index is used for read, write, seek, and close operations

close()

Closes the file.

On close(), the OS:

- Removes the entry for the file in the process's file table
- Decrements the open count in the system-wide file table
- If the open count == 0, removes the entry in the system-wide file table

read()

Read(fileId, from , size, bufAddress) [Random Access]

- OS reads <size> bytes from the file position <from> into <bufAddress>.
- So:

```
for(i=from; i < from+size; i++)
   bufAddress[i-from] = file[i]
```

Read(fileId, size, bufAddress) [Sequential Access]

- OS reads <size> bytes from current file position, <fp>, into <bufAddress> and increments file position by size
- So:

```
for(i=0; i < size; i++)
   bufAddress[i] = file[fp+i]
fp += size
```

Other Commands

- write() is similar to read(), but copies the buffer to the file
- seek() updates the file pointer
- fsync() does not return until all data is written to persistent storage

File System Design and Implementation

File System Functionality

- File system translates **from** file name and offset **to** data block
- Manage disk layout
  - Pick the blocks that constitute a file
  - Balance locality with expandability
  - Manage free space
- Provide the file naming organization, such as hierarchical name space

Basic Data Structures

- Disk
  - An array of blocks, where a block is a fixed size data array
- File
  - Sequence of blocks
- Directory
  - Creates the namespace of files

File System Concepts

- Metadata
  - The *superblock* has important file system metadata
    - file system type, number of blocks in the file system, …
  - The *file header* describes where the file is on disk and the attributes of the file
- Data
  - The contents that users actually care about
  - Files
    - Contain data and have metadata like creation time, length, etc.
  - Directories
    - Map file names to file headers

File System Implementation

- File header: owner id, size last modified time, and location of all data blocks
  - OS should be able to find metadata block number N without a disk access (e.g., by using math or cached data structure)
- Data blocks
  - Directory data blocks (human readable names)
  - File data blocks (data)
- Other pieces to the puzzle: superblocks, group descriptors, and other metadata…

File Layout

How do we find and organize files on the disk?

We need to convert:

fileID 0, Block 0 to Disk block 19

fileID 0, Block 1 to Disk block 4,528

Things to consider in our design:

1. We need to support sequential and random access.
2. We need to lay out the files on the physical disk.
3. We need to maintain file location information (in what data structure?).

Typical File System Profile

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

How do these properties affect our design?

Typical File System Profile

- Most files are small
  - Need strong support for small files
  - Block size can't be too large (why?)
- Most disk space is consumed by large files
  - Must allow large files
  - Large file access should be reasonably efficient
- I/O operations target both small and large files
  - The per-file cost must be low, but large files must also have good performance

Block vs. Sector

- The OS may choose to use a larger block size than the sector size of the physical disk. (Why?)
  - Each block consists of consecutive sectors (Why?)
  - A larger block size increases the transfer efficiency (Why?)
  - It may be convenient if the block size matches (a multiple of) the machine's page size (Why?)
- Most systems allow transferring of many sectors between interrupts

iClicker Question

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

A. Large

B. Small

How do we find and organize files on the disk?

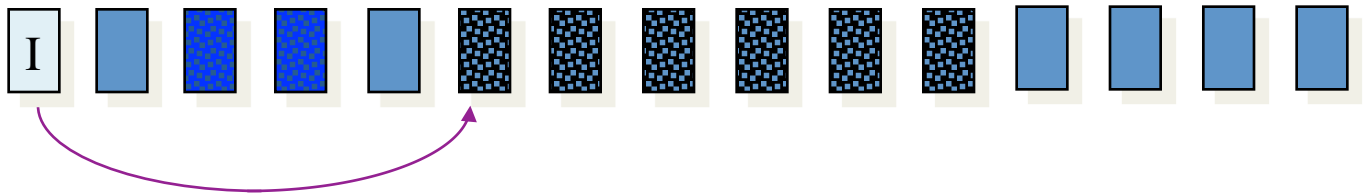
Approaches:

1. Contiguous allocation
2. Linked files
3. Indexed files
4. Multilevel indexed files

Contiguous Allocation

- OS maintains an ordered list of free disk blocks
- OS allocates a contiguous chunk of free blocks when it creates a file
- The location information in the file header need only contain the start location and the size
- Advantages: simple, what about access time?
- Disadvantages: changing file sizes? fragmentation?

Contiguous Allocation



- File header specifies starting block and length
- Placement/Allocation policies
  - First-fit, best-fit, worst-fit

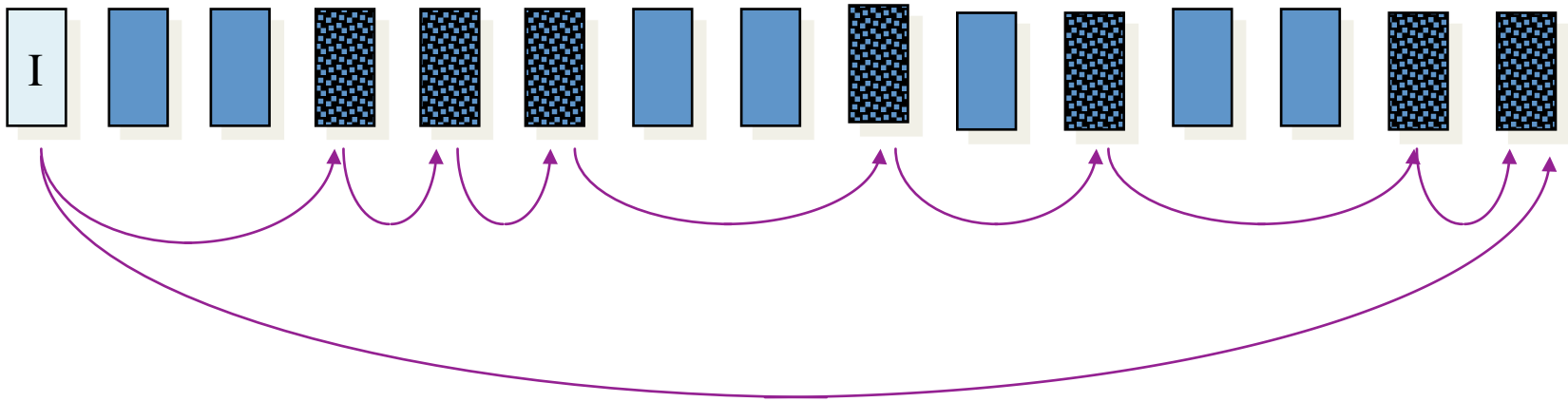
Contiguous Allocation: Text Description

- 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
- In the file header, keep a pointer to the first and last sector/block allocated to that file
- In each sector, keep a pointer to the next sector
- Advantages: fragmentation? file size changes? efficiently supports which type of access?
- Disadvantages: Does not support which type of access? Why? Access time? Reliability?

Linked Allocation



- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks
- Two implementations
  - linked list of disk blocks (seen above)
  - linked list in a table (FAT)

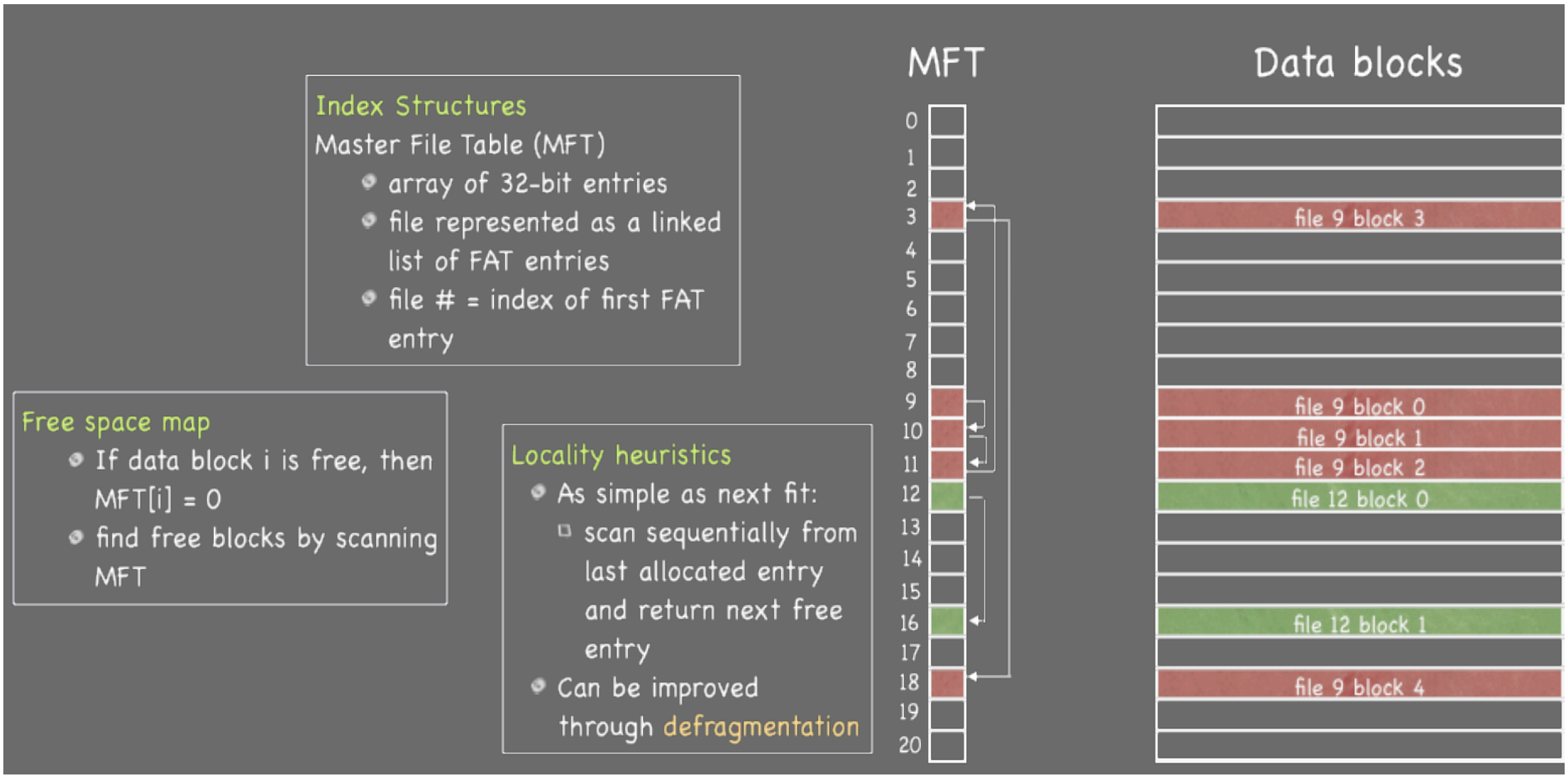
Linked Allocation: Text Description

- File stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks
  - Pointer to last block makes it easier to grow the file
- 2 implementations
  - Linked list of disk blocks, data blocks point to other blocks
  - Linked list in a table (FAT)

Example: FAT File System

File Allocation Table (FAT)

- started with MS-DOS (Microsoft, late 70s)
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}-1 bytes



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
  - FAT entry is 32 bits but top 4 are reserved
  - no more than 2^28 blocks
  - with 4KB blocks, at most 1 TB volume
  - file no bigger than 4 GB
- No support for transactional updates (more later)

FAT File System: Plain Text

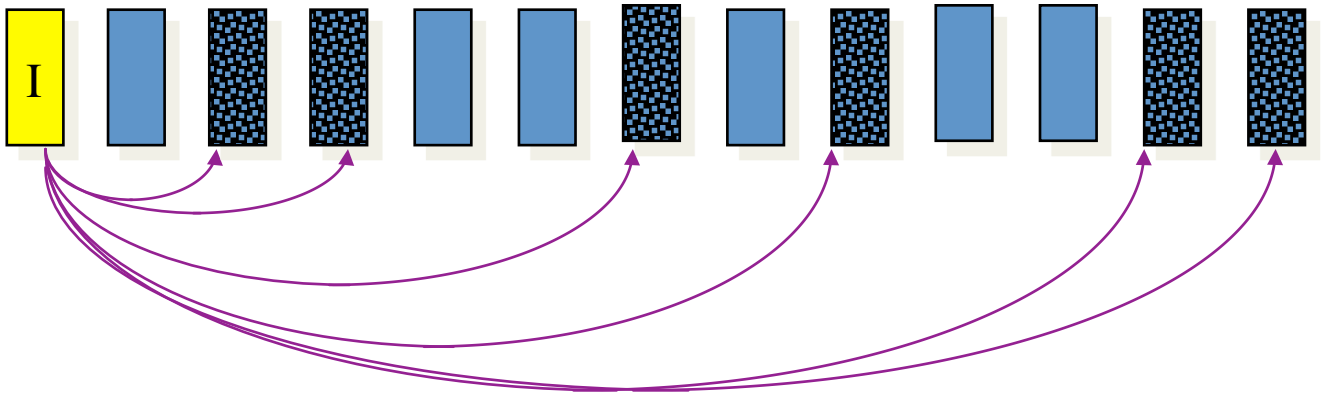
- Advantages
  - Simple!
- Disadvantages
  - Poor random access
    - Requires sequential traversal
  - Limited access control
    - No file owner or group ID metadata
    - Any user can read/write any file
  - No support for hard links
    - Metadata stored in directory entry
  - Volume and file size are limited
    - FAT entry is 32 bits but top 4 are reserved
    - no more than 2^28 blocks
    - with 4KB blocks, at most 1 TB volume
    - File no bigger than 4 GB
  - No support for transactional updates (more later)

Direct Allocation

- File header points to each data block
- Advantages: Easy to create, grow, and shrink files, little fragmentation, supports random access
- Disadvantages: Index node (*inode*) is big or variable sized

What should we do about large files?

Direct Allocation

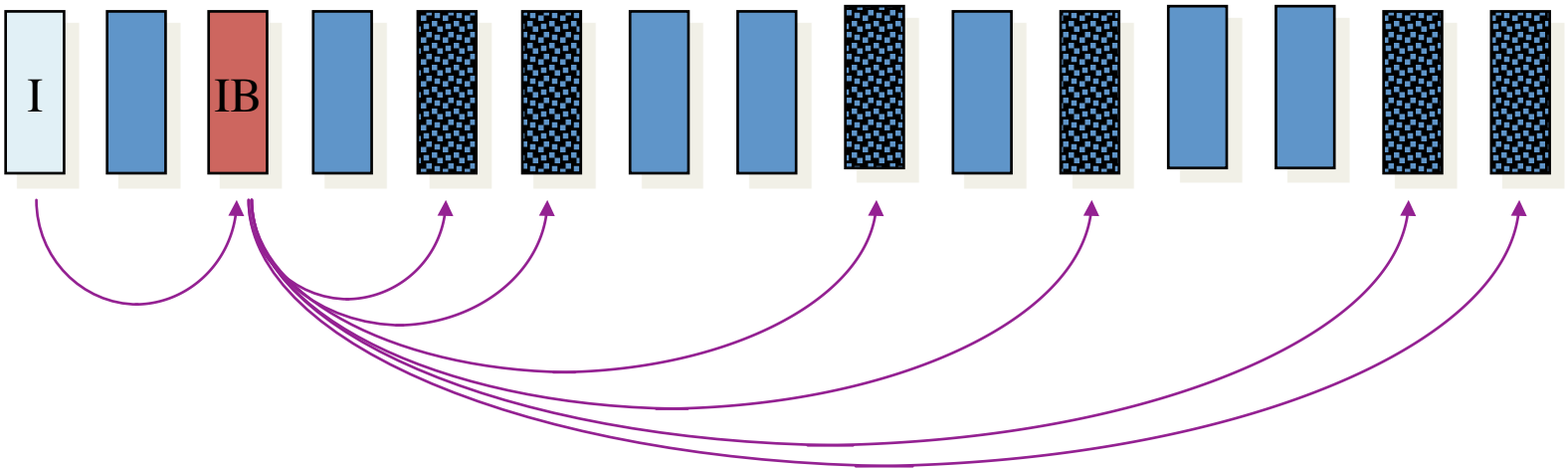


File header points to each data block

Indexed Allocation

- OS keeps an array of block pointers for each file in a non-data block called the *index block*
- OS allocates an array to hold the pointers to all the blocks when it creates a file, but allocates the blocks only on demand
- OS fills in the pointers as it allocates blocks
- Advantages: Supports both types of access, not much fragmentation
- Disadvantages: Maximum file size! Lots of seeks since data is not contiguous, what about large files?

Indexed Allocation



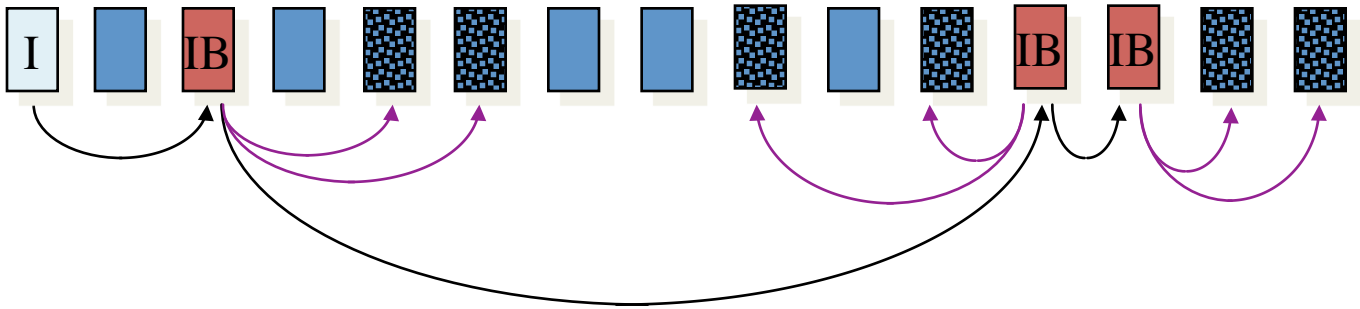
- Create a non-data block for each file called the *index block*
  - A list of pointers to file blocks
  - Number based on sixe of pointer and size of block
- File header contains the index block

Indexed Allocation: Text Description

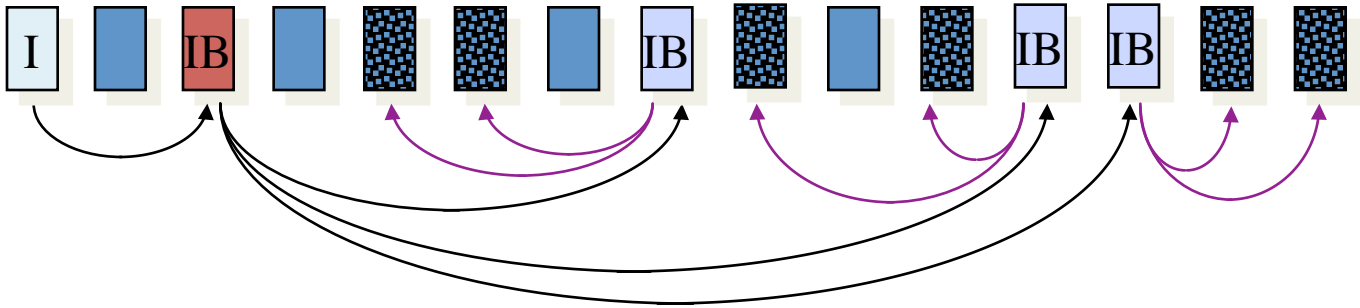
- Create a non-data block for each file called the index block (IB)
  - Contains a list of pointers to file blocks
  - Number of pointers based on size of pointer and size of block
- File header contains, or has a pointer to, the index block (but no longer points to data blocks)
  - So file header has no direct knowledge of where the file information is on disk

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

iClicker 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

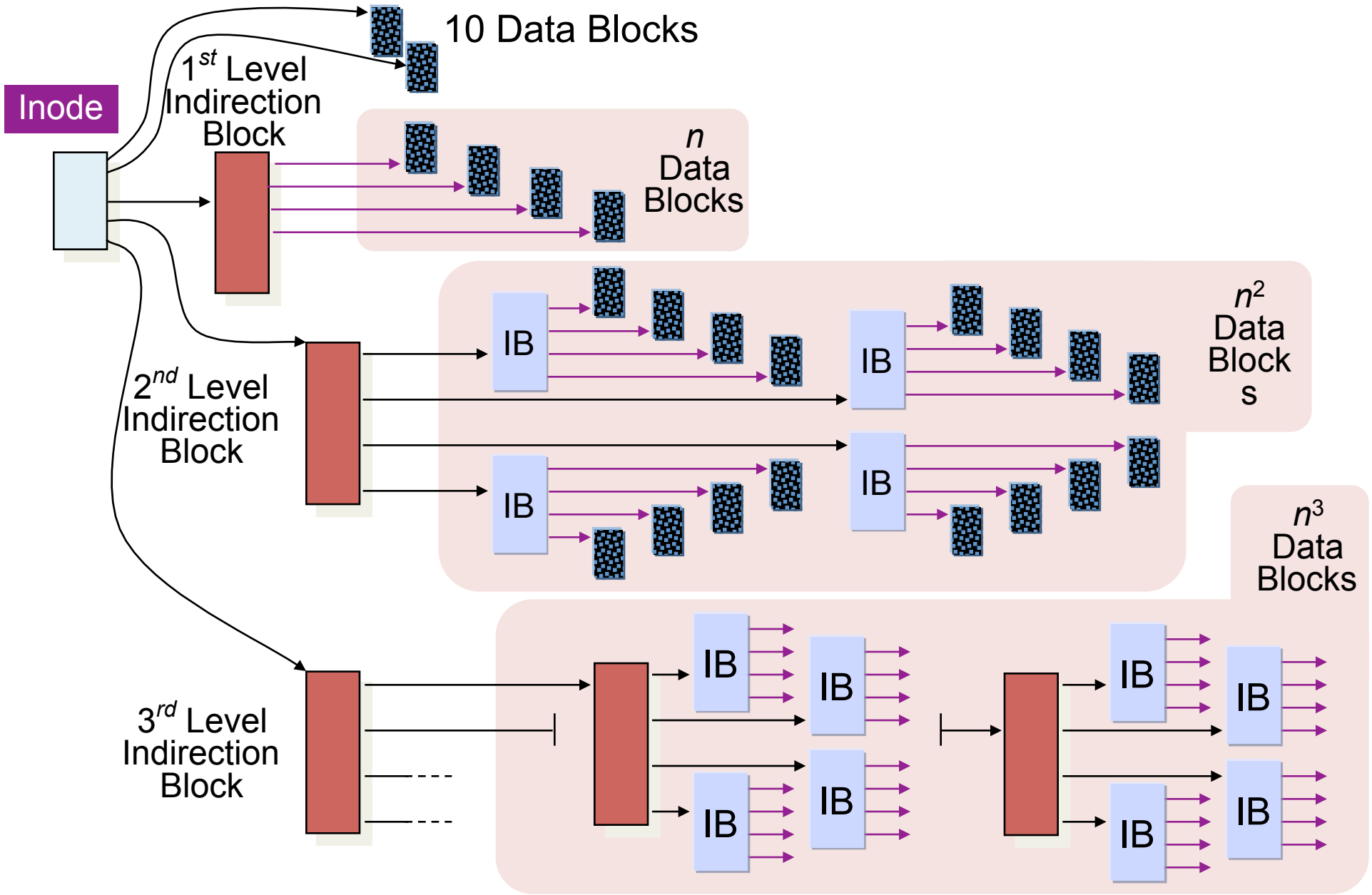
- Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves
- The root of the tree is the file's inode (UNIX/Linux specific name for file header)
  - contains file's metadata
    - owner, permissions (rwx for owner, group other), directory?, etc
    - setuid: file is always executed with owner's permission
      - add flexibility but can be dangerous
    - setgid: like setuid for groups
  - contains a set of pointers
    - typically 13
    - first 10 point to data block
    - but last three point to intermediate blocks

Multilevel Indexed Files

- 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).
- Advantages: simple to implement, supports incremental file growth, small files?
- Disadvantages: random access to very large files is inefficient, many seeks

Indexed Allocation in UNIX

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 $n^{\wedge 2}$ 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 $n^{\wedge 3}$ data blocks
- In total structure holds: $10 + n + n^{\wedge 2} + n^{\wedge 3}$ 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

Example: Fast File System (FFS)

- Implemented by UNIX in the 80s
- Smart index structure
  - multilevel index allows to locate all blocks of a file
    - efficient for both large and small files
- Smart locality heuristics
  - block group placement (more soon)
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space (more soon)
    - gives up about 10% of storage to allow flexibility needed to achieve locality

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 =

$12 \times 4KB + 512 \times 4KB + 512^2 \times 4KB + 512^3 \times 4KB + 512^4 \times 4KB$

= 256.5 TB

Summary

- File system is an abstraction
  - Provides order to linear bytes of data
- Saw file system API
  - create, link, unlink, read, write, …
- File header keeps file metadata
- File layout determines how we find the file on disk

Announcements

- Homework 8 due Friday 8:45a
- Exam next week (Wednesday 4/8)
  - UTC 2.122A 7p-9p
- Class performance formula will be posted to Piazza on Thursday
- Project 3 is posted due Friday, 11/14
  - NO hand-holding
  - Project 2 must be working
    - Except multi-oom
    - No, we will not give you solutions
- Instructions for how to get help with any remaining Project 2 tests are/will be posted to Piazza
  - You must show us a working Project 2

Summary

- File system is an abstraction
  - Provides order to linear bytes of data
- Directories provide a way to locate each of the files and map the OS number to the human-friendly name
- Finding a file from the root node can be expensive, so the current working directory is cached
- On disks, the inode structures and free map are kept in specific places
  - but where varies
- Locality heuristics group data to maximize access performance

Summary

Many of the concerns and implementation details of the file system are similar to those of memory management.

- Contiguous allocation is simple, but suffers from external fragmentation, need to compaction, and the need to move files as they grow
- Indexed allocation is very similar to page tables. A table maps from logical files to physical disk blocks.
- Free space can be managed using a bitmap or a linked list.