File System Design and Consistency

CS439: Principles of Computer Systems

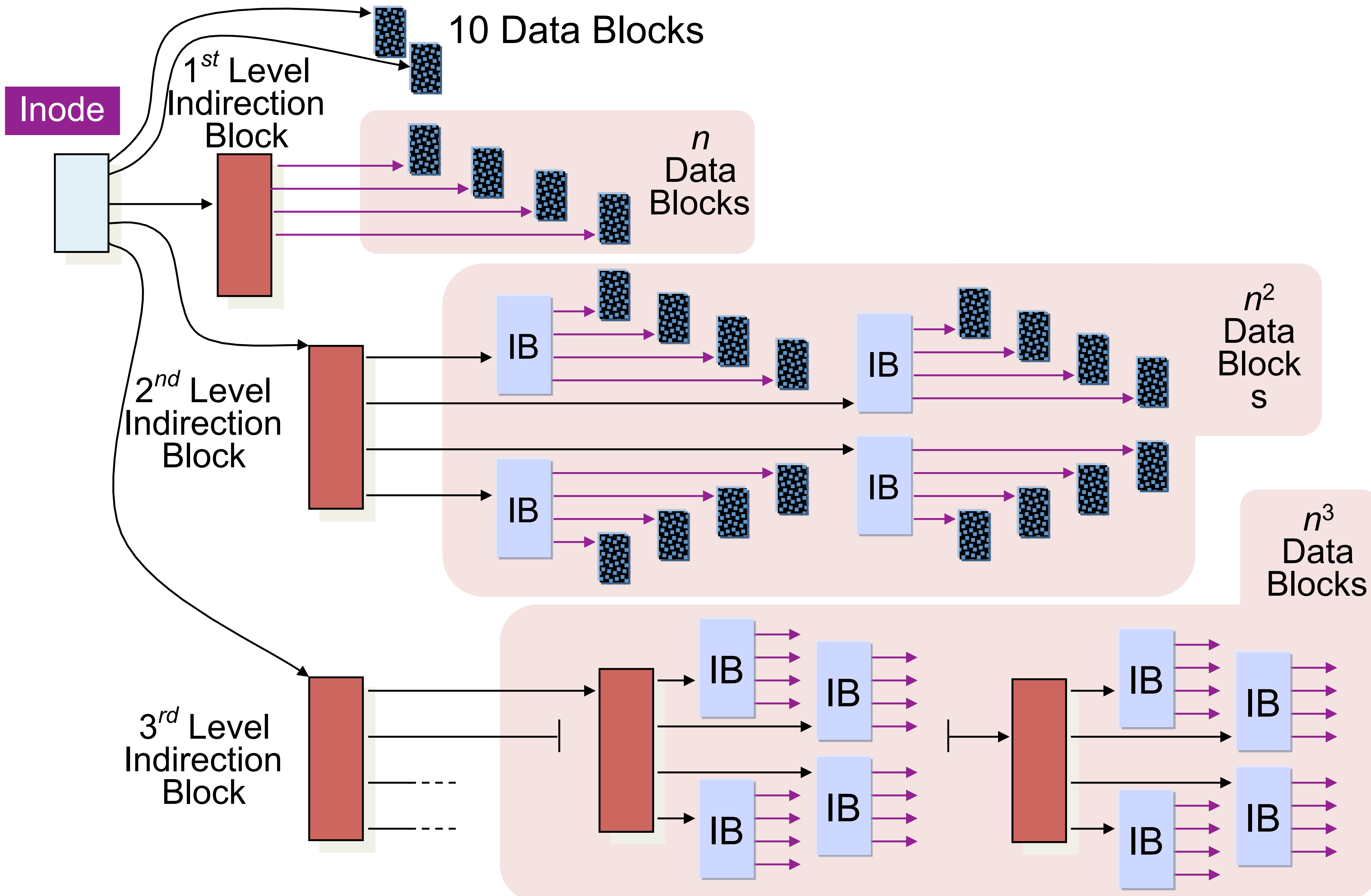
April 1, 2015

Last Time

- Disks
  - Flash storage
    - Read/write cost, durability
- File System Use
  - system calls (ish) to create, remove, open, read, write, and close files
- File System Implementation
  - How are files organized?
    - Contiguous, Linked, Direct, Indexed, Multilevel Indexed

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^^{2} data blocks
  - 3rd level indirection block
    - inode points to block full of pointers to 2nd 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

Today's Agenda

- File System Implementation
  - Directories
    - Designs
    - How they work
  - Finding files on disk (FFS)
  - Disk Layout
- NTFS
- File System Consistency
  - Sources of Inconsistency
  - Maintaining Consistency/Fixing Inconsistencies
- File System Fault Tolerance
  - Transactions
  - Journaling File Systems
  - Copy-on-Write File Systems

Movin' on up… Directories

Directories

- A file that contains a collection of mappings from file name to file number
  - Those mappings are directory entries

  <name, file number>

  The *file number* is an *inumber* (inode number).
- Only OS can modify directories
  - Ensure integrity of mapping
  - Application programs *can* read directories
- Directories create a name space for the files

Directory Strategies: Simple and Stupid

One name space for the entire disk.

- Use a special area of the disk to hold the directory
- Directory contains <name, index> pairs
- If one user uses a name, no one else can

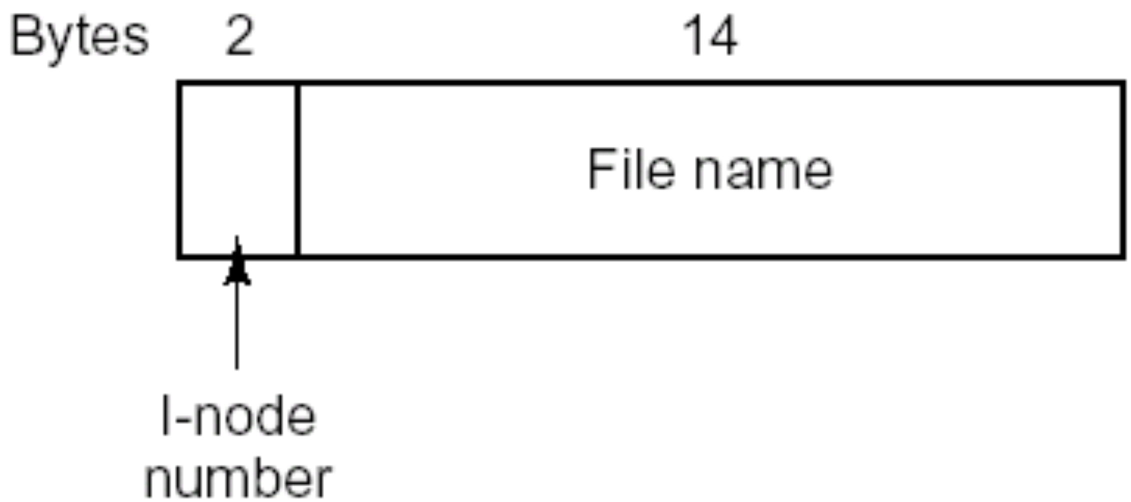
Directory Strategies: Simple User-Based Strategy

- Each user has a separate directory
- BUT all of each user's files must still have unique names

Directory Strategies: Multi-level Directories

Tree-structured hierarchical name space (all modern OSs)

- Store directories on disk, just like files except the file header for directories has a special flag bit
- User programs read directories just like any other file, but only special system calls can write directories
- Each directory contains <name, file number> pairs in no particular order
  - The file referred to by <name> may be another directory
- There is one special root directory



A simple UNIX directory entry

Simple Unix Directory Entry: Text Description

- Total 16 bytes
  - First 2 bytes hold i-node number
  - Next 14 bytes hold the file name

iClicker Question

Every directory has a file header.

A. True
B. False

Given only the inode number (*inumber*) the OS can find the inode on disk.

A. True
B. False

How do you find the blocks of a file?

- Find the file header (inode); it contains pointers to file blocks
- To find file header (inode), we need its inumber
- To find inumber, read the directory that contains the file
- To find the directory…
- But wait! The directory is a file…

Example: Read file `/Users/ans/wisdom.txt`

How do we find it?

- `wisdom.txt` is a file
- `ans/` is a directory that contains the inumber for `wisdom.txt`
  - Locate `ans/`, read directory
- `Users/` is a directory that contains the inumber for `ans`
  - Locate `Users/`, read directory
- How do you find the inumber for `Users/`?
  - `/` is a directory that contains the inumber for `Users/`
  - In Unix, `/`'s inumber is 2
    - (whew! At least we know how to find that! Or do we? … disk layout up soon!)

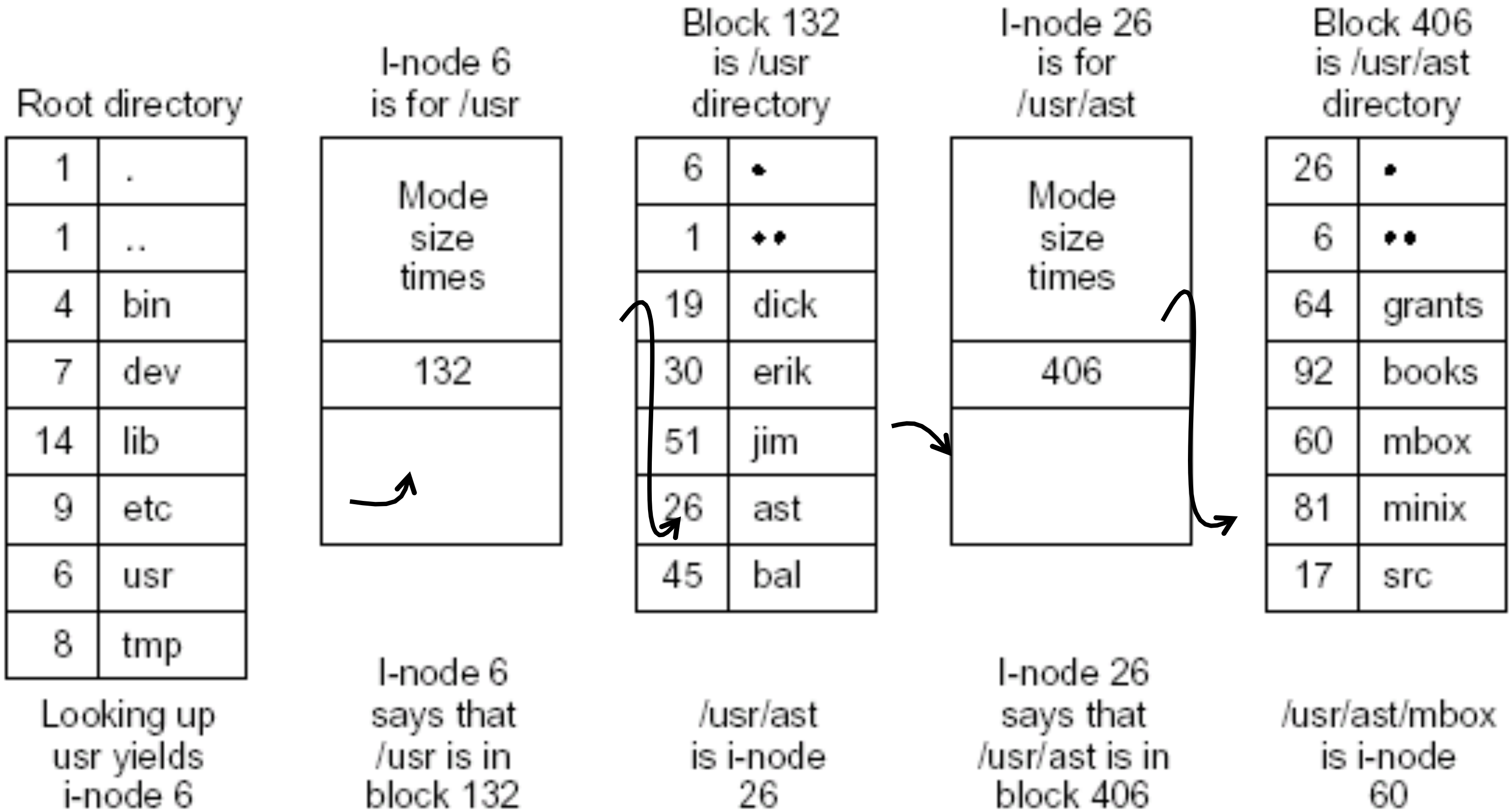
How much work was that?

How many disk accesses are needed to access file `/Users/ans/wisdom.txt`?

1. Read the inode for `/` from a fixed location
2. Read the first data block for root
3. Read the inode for `Users/`
4. Read the first data block for `Users/`
5. Read the inode for `ans/`
6. Read the first data block for `ans/`
7. Read the inode for `wisdom.txt`
8. Read the first data block for `wisdom.txt`

`“A cache is a (wo)man’s best friend!”`

Another Example: Reading a File



The steps in looking up */usr/ast/mbox*.

Root directory is a file itself and its inode is at a fixed location on
disk.

Another Example: Reading a File Text Description

The steps in looking up /usr/ast/mbox on the disk

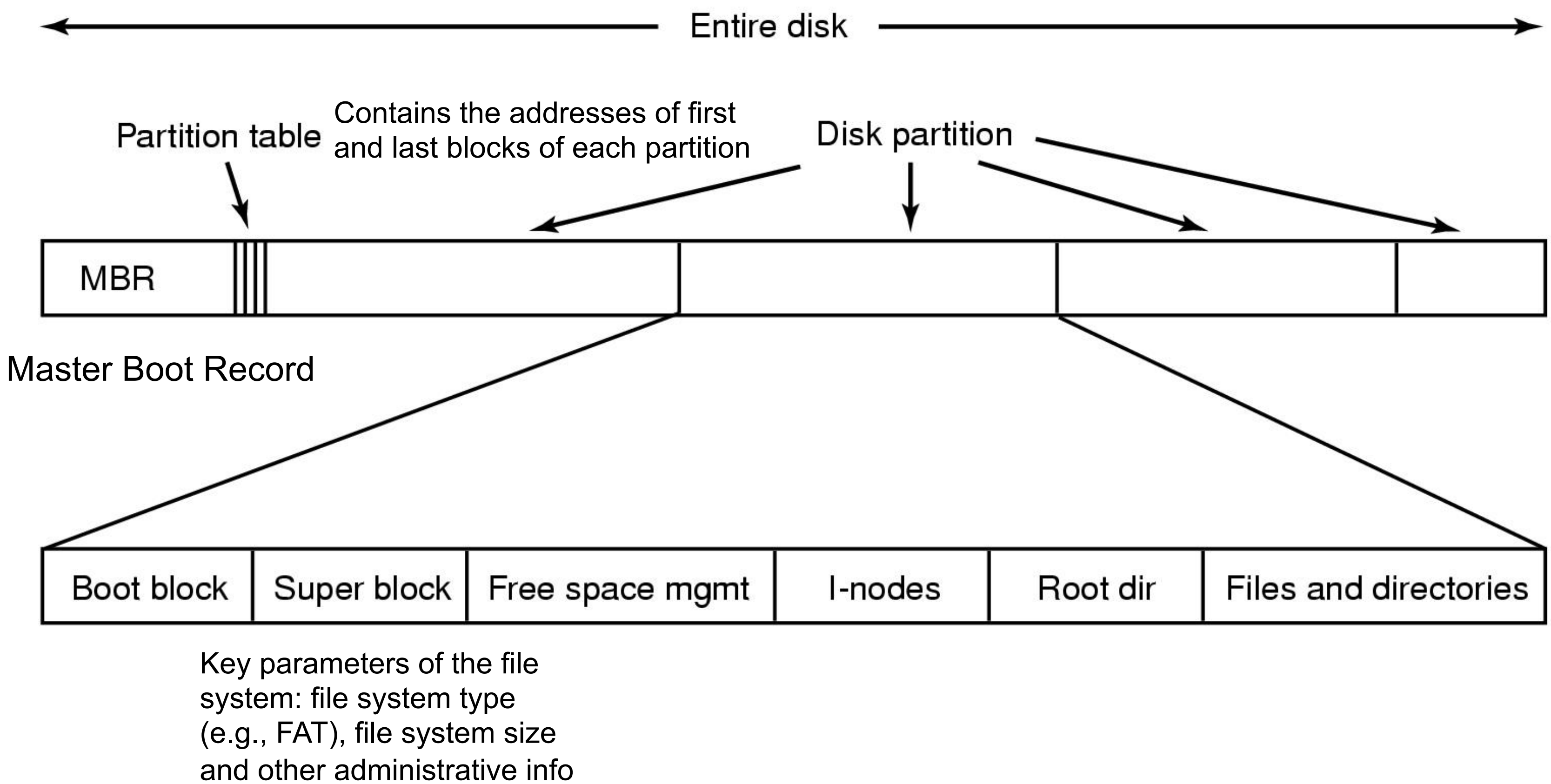
1. Retrieve root directory inode from fixed location on disk
2. Retrieve root directory data block from location given in inode.
3. Look up usr in root directory data block; its information is in inode 6
4. Go to inode 6 and determine which block holds the data for /usr
   - This is block 132
5. Get block 132, the /usr directory data, and lookup ast; its information is in inode 26
6. Go to inode 26 and determine which block holds the data for /usr/ast
   - This is block 406
7. Get block 406, the /usr/ast data, and lookup mbox; its information is in inode 60
   - Woot! We have found our email!

Optimize

- Maintain the notion of *current working directory* (CWD)
- Users can now specify relative file names
- OS can cache the data blocks of CWD

File System Layout

File System Layout on Disk



File System Layout on Disk: Text Description

- Components of the entire disk
  - MBR - Master Boot Record
  - Partition table: contains the addresses of first and last blocks of each partition
  - Disk partitions
- Components of each partition
  - Boot block
  - Super block
  - Free space management
  - I-nodes
  - Root directory
  - Files and directories
- Components of Super Block
  - File system type (eg FAT)
  - File system size
  - Key parameters of system
  - Other administrative info

Finding Free Space

- Need a data block? Consult list of free data blocks!
- Need an inode? Consult list of free inodes!
- Represent the list of free blocks as a bit vector

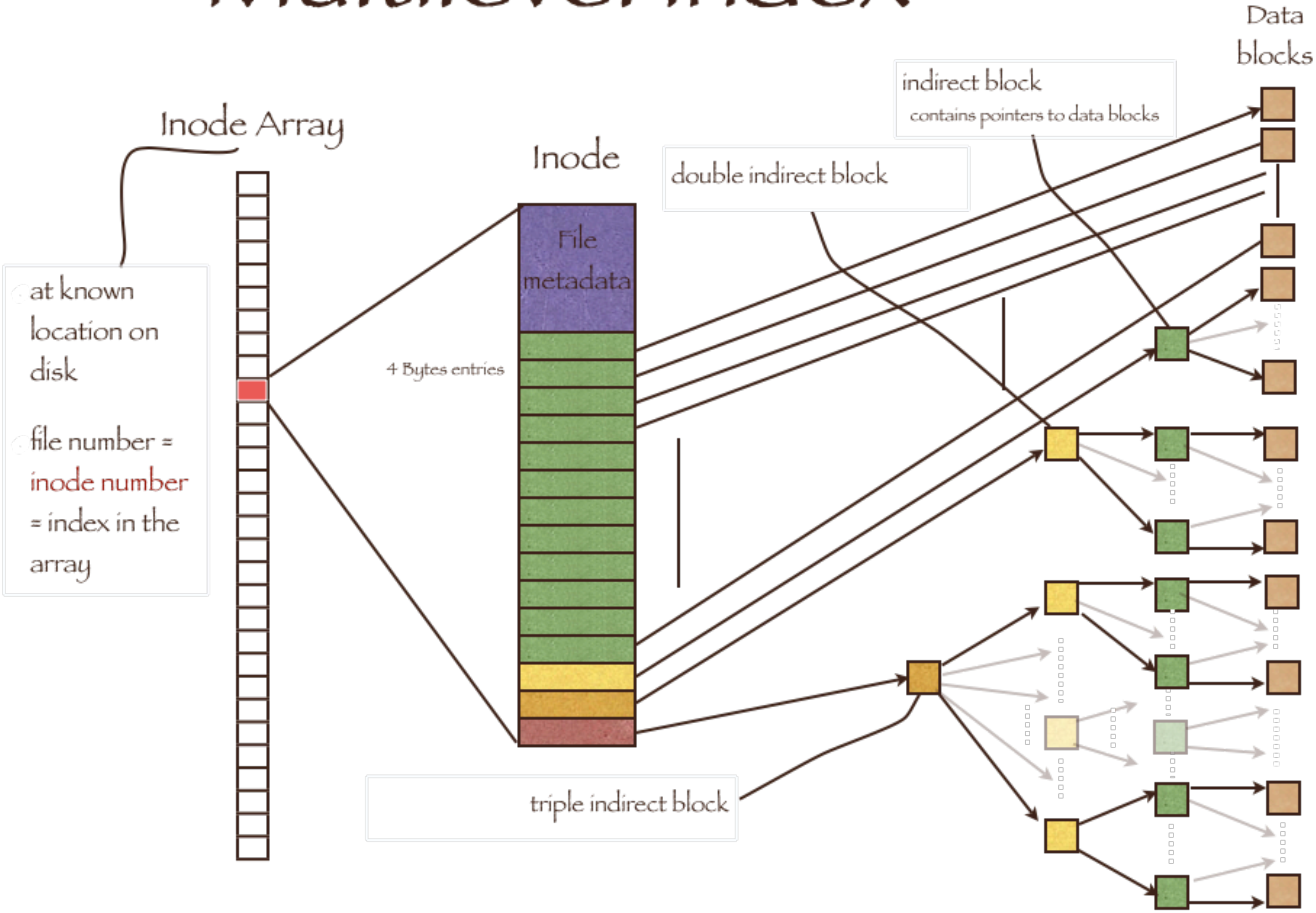
  111110000111111001110101111110011

  - one bit for each block on the disk
  - if bit is 1 then in is allocated, if 0 it is free
- Or represent them as a linked list

More About the Bitmap

- One bit per storage block
- Bitmap location fixed at formatting time
- i-th bit indicates whether i-th block is used or free

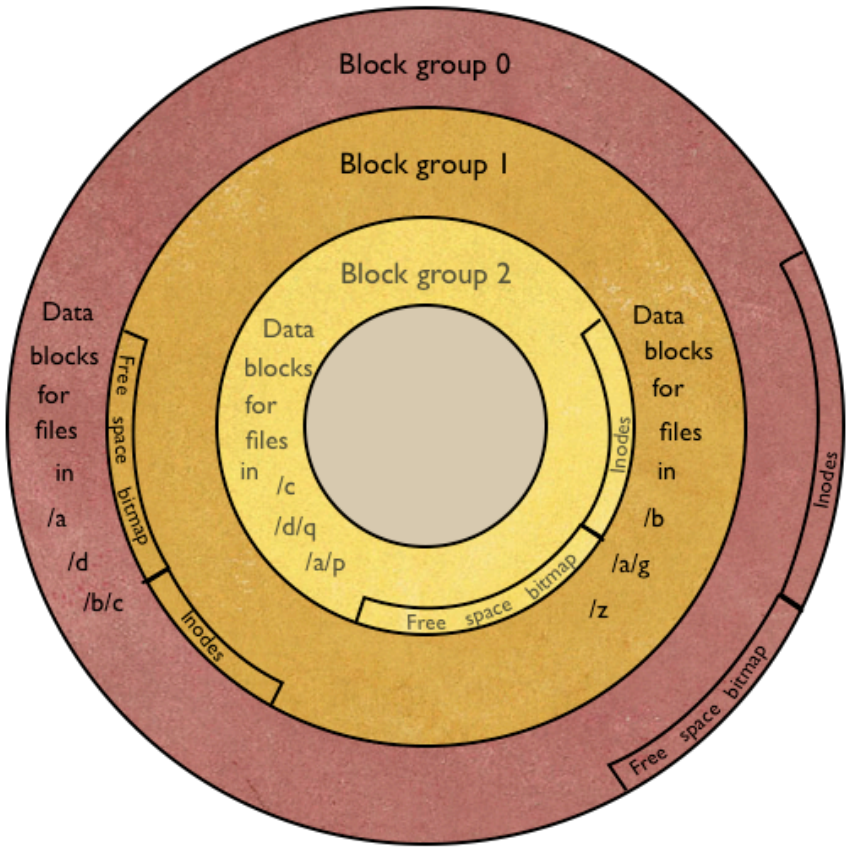
Multilevel index



Multilevel Index: Text Description

- Inode array is located at a known location on disk
  - file number = inode number = index in the array
  - contains all the inodes in the file system
- Inode contains:
  - File metadata
  - A certain number of direct pointers to data blocks
    - This example has 12
  - A pointer to an index block, which contains pointers to data blocks
  - A pointer to a double indirect block, which contains pointers to index blocks
  - A pointer to a triple indirect block, which contains pointers to double indirect blocks

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

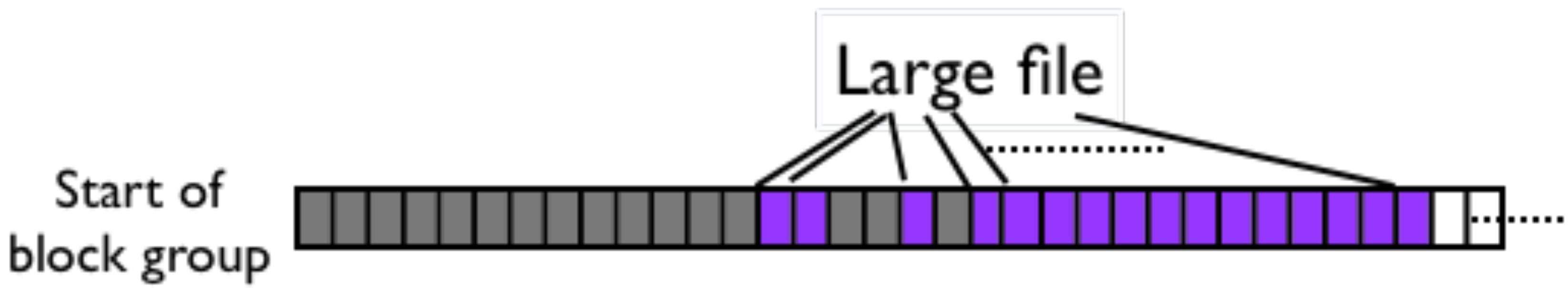
FFS Locality Heuristics: Block Groups
Plain Text

- Divide partition into block groups
  - Group = set of nearby tracks
  - Disk group placement
    - Block group 0 on outer edge
    - Block group 1 between groups 0 and 2
    - Block group 2 closest to the center of the surface
- 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
    - Idea: if you are in a directory you will be accessing other files within that 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

FFS Locality Heuristics: Block Group File Placement

FFS Locality Heuristics: Block Group File Placement

FFS Locality Heuristics: Block Group File Placement



FFS Locality Heuristics: Block Group File Placement Text Description

- We have a block group that is partially filled
  - Blocks 0-6 are taken
  - Block 7 is free
  - Block 8 is taken
  - Block 9 is free
  - Blocks 10-12 are taken
  - Blocks 13-14 are free
  - Blocks 15-16 are taken
  - Block 17 is free
  - Block 18 is taken
  - The rest of the blocks are free
- We want to allocate a small file that is 2 blocks long
  - We are using a first-free scheme
  - The file will be split between block 7 and block 9, since they are the first 2 free blocks in the group
- Now we want to allocate a large file
  - It will be split into 3 pieces:
    - Blocks 13-14
    - Block 17
    - The long string of free blocks at the end of the group

FFS Locality Heuristics: Reserved Space

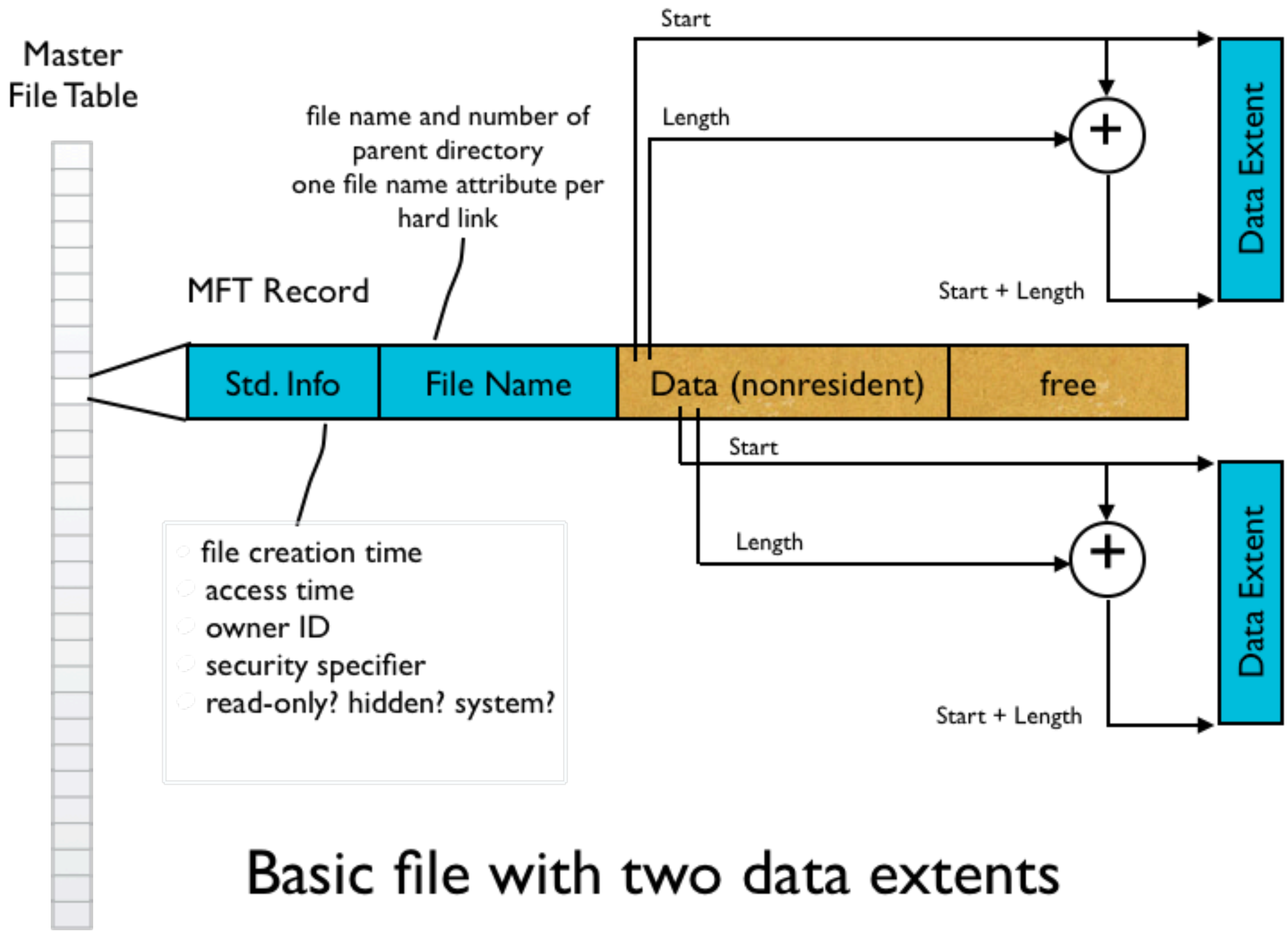
- When a disk is close to full, hard to optimize locality
  - file may end up scattered through disk
- FFS presents applications with a smaller disk
  - about 10% smaller
  - user write that encroaches on reserved space fails
  - super user still able to allocate inodes

NTFS: Flexible Tree with Extents

Index structure: extents and flexible tree

- Extents
  - Track ranges of contiguous blocks rather than single blocks
- Flexible tree
  - File represented by variable depth tree
    - Large file with few extents can be stored in a shallow tree
  - MFT (Master File Table)
    - Array of 1 KB records holding the trees' roots
    - Similar to inode table
    - Each record stores sequence of variable-sized attribute records
- Microsoft 1993

Example of NTFS

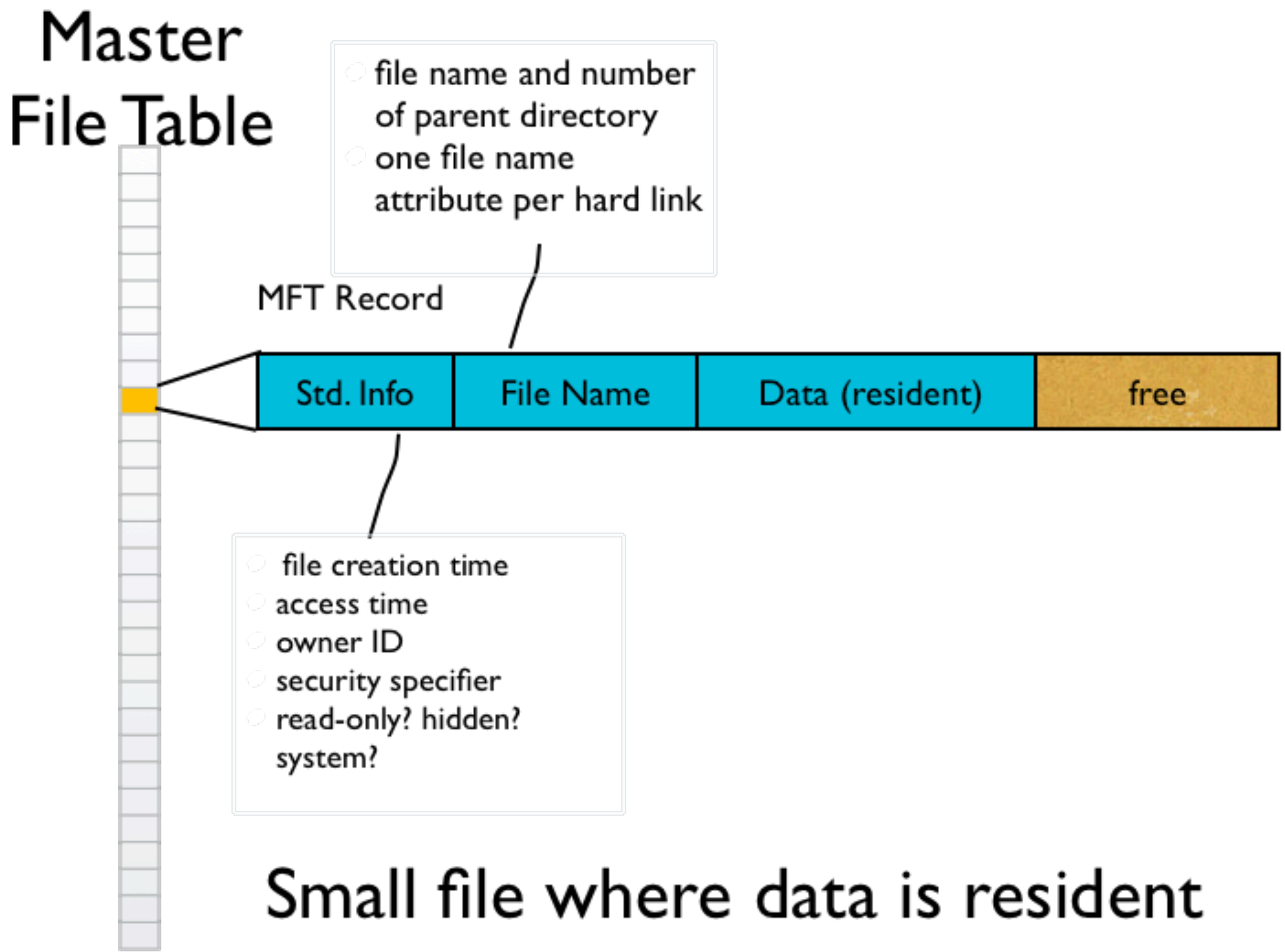


Basic file with two data extents

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

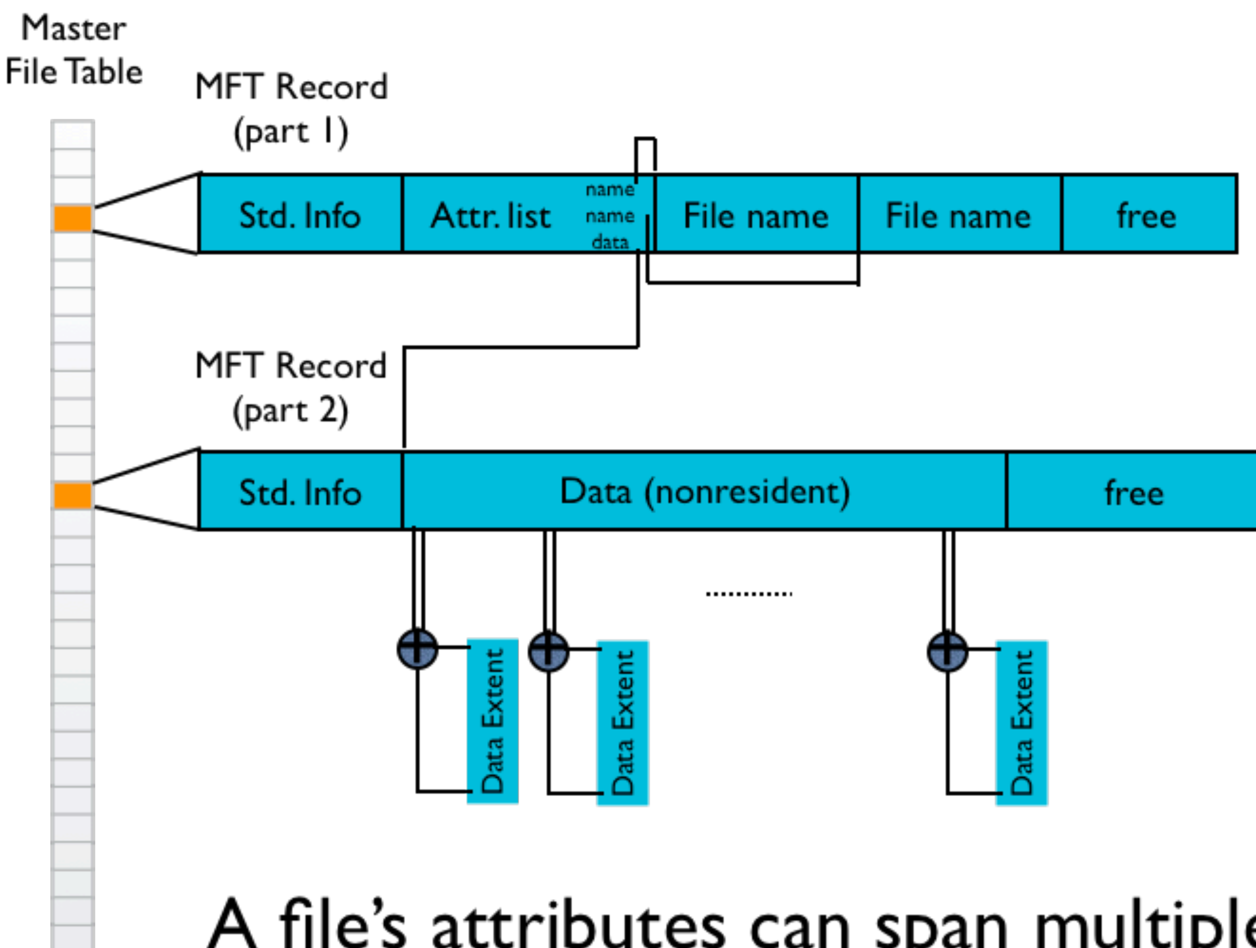


Small file where data is resident

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

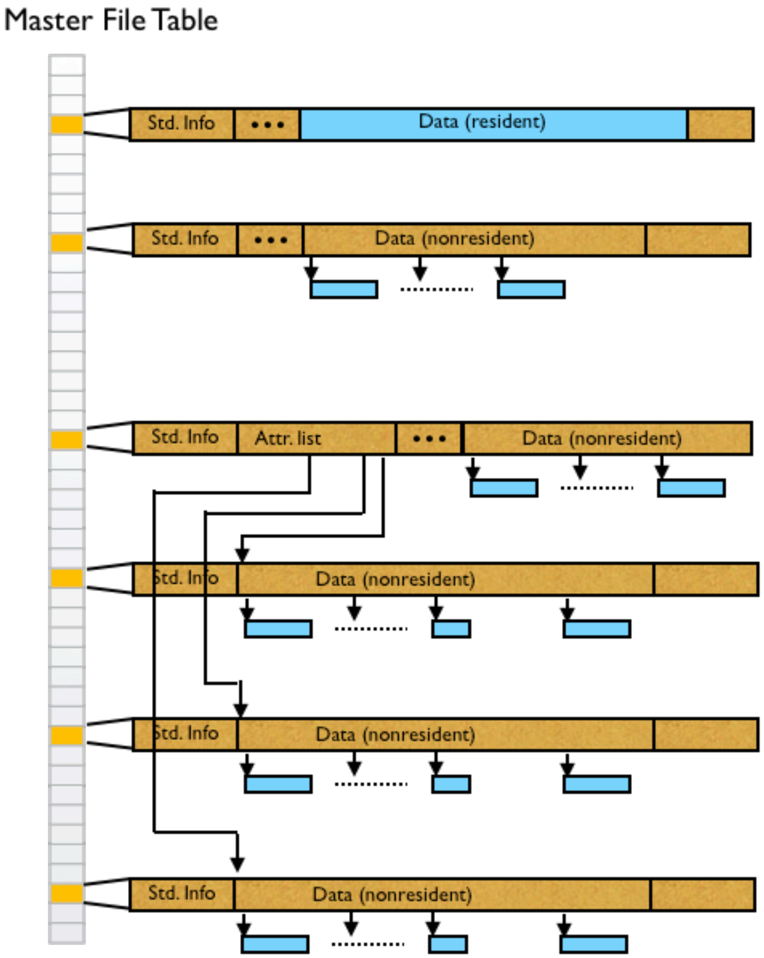


A file's attributes can span multiple records

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

The File System Abstraction (A Few More Things)

- path: string that identifies a file or directory
  - absolute (if it starts with "/", the root directory)
  - relative (with respect to the current working directory)
- mount: allows multiple file systems to form a single logical hierarchy
  - a mapping from some path in existing file system to the root directory of the mounted file system

Stepping Back (FS and Disks)

From the user's point of view, what is important?

• *Persistence*: data preserved between jobs, power cycles, crashes

• *Speed*: can get to data quickly

• *Size*: can store lots of data

• *Sharing/Protection*: Users can share data where appropriate or keep it private when appropriate

• *Ease of use*: user can easily find, examine, modify data

Hardware provides:

- *Persistence*: Disks provide non-volatile memory
- *Speed*: Speed gained though random access
- *Size*: Disks are getting bigger and bigger

OS provides:

- *Persistence*: Redundancy allows recovery from some additional failures
- *Sharing/Protection*: Unix allows the owner to control privileges
- *Ease of Use*

Stepping Back (FS and Disks): Plain Text

- From the user's point of view, what is important?
  - Persistence: data preserved between jobs, power cycles, crashes
  - Speed: can get to data quickly
  - Size: can store lots of data
  - Sharing/Protection: Users can share data where appropriate or keep it private when appropriate
  - Ease of use: user can easily find, examine, modify data
- Hardware provides:
  - Persistence: Disks provide non-volatile memory
  - Speed: Speed gained though random access
  - Size: Disks are getting bigger and bigger
- OS provides:
  - Persistence: Redundancy allows recovery from some additional failures
  - Sharing/Protection: Unix allows the owner to control privileges
  - Ease of Use

How Persistent Storage Affects OS Design

Goal	Physical Characteristics	Design Implication
High Performance	*Large cost to initiate I/O	*Organize storage to access data in large sequential units *Use caching
Named Data	*Large capacity *Survives crashes *Shared across programs	*Support files and directories with meaningful names
Controlled Sharing	*Device may store data from many users	*Include metadata for access control
Reliability	*Crash can occur during updates *Storage devices can fail	*Use transactions *Use redundancy to detect and correct failures

How Persistent Storage Affects OS Design: Plain Text

- Goal: High Performance
  - Physical characteristic of persistent storage:
    - Large cost to initiate I/O
  - OS design implications:
    - Organize storage to access data in large sequential units
    - Use caching
- Goal: Named Data
  - Physical characteristics of persistent storage:
    - Large capacity
    - Survives crashes
    - Shared across programs
  - OS design implication:
    - Support files and directories with meaningful names
- Goal: Controlled Sharing
  - Physical characteristic of persistent storage:
    - Device may store data from many users
  - OS design implications:
    - Include with files metadata for access control
    - Use transactions

File System Types and Consistency

Cached Data Structures

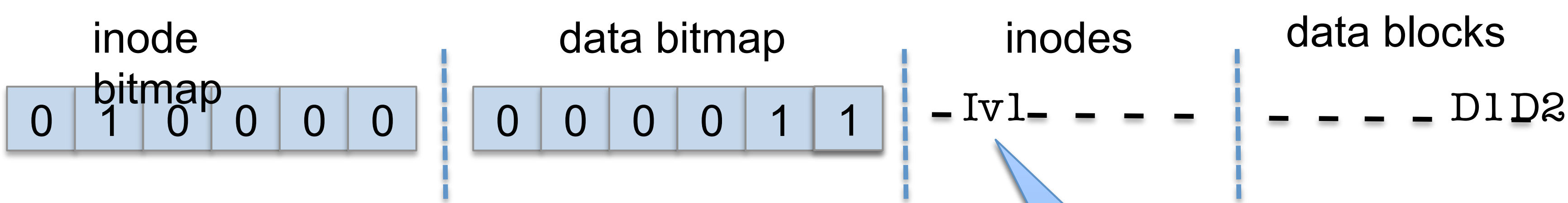
- File systems have many data structures that the OS caches to get good performance
  - Bit map/list of free blocks
  - Directories
  - File headers
  - Indirect blocks
  - Data blocks
- Keeping them accurate is easy if we read
- But what if we write?

Writing to Caches

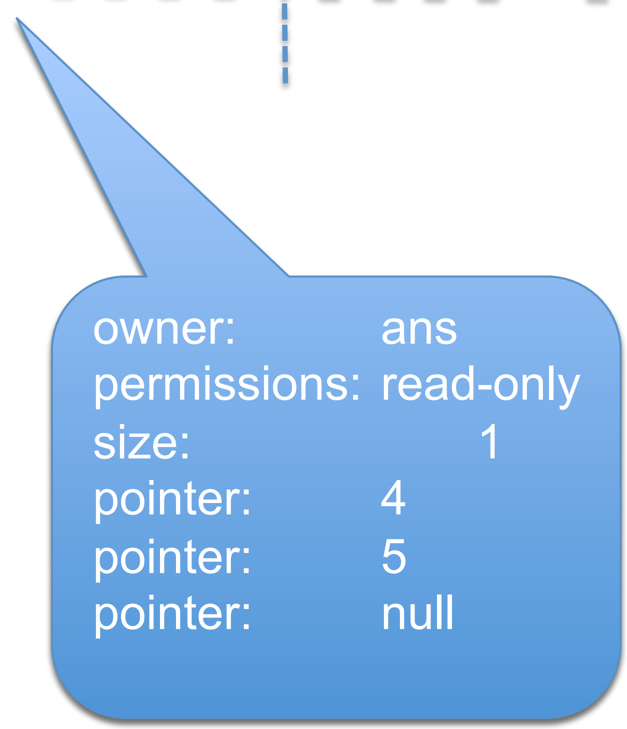
- Modified data kept in memory can be lost!
- *Write-through (now)*: write changes immediately back to disk
  - Consistent
  - *Slow!* We have to wait for the write to hit the disk and generate an interrupt
- *Write-back (later)*: delay writing the modified data until, for example, the page is replaced in memory
  - Better performance
  - Can cause inconsistencies since the data can be lost in a crash

Example: A Tiny ext2

- 6 blocks, 6 inodes



- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap



What if a crash or power outage occurs between writes?

Example: A Tiny ext2 Plain Text

- 6 blocks, 6 indoes
- Original data for structures
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 0
  - Inodes: _ [v] _ _ _ _
    - _ means that that spot is empty
    - Information for [v]
      - Owner: ans
      - Permissions: read-only
      - Size: 1
      - Pointer: 4
      - Pointer: null
      - Pointer: null
  - Data blocks: _ _ _ _ D1 _
- Suppose we append a data block to the file
  - Add new data block D2
  - Update inode
  - Update data bitmap
- Updated data for structures
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 1
  - Inodes: _ [v] _ _ _ _
    - _ means that that spot is empty
    - Information for [v]
      - Owner: ans
      - Permissions: read-only
      - Size: 1
      - Pointer: 4
      - Pointer: 5
      - Pointer: null
  - Data blocks: _ _ _ _ D1 D2
- What if a crash or power outage occurs between writes?

What If Only a Single Write Succeeds?

- Just the data block (D2) is written to disk
  - Data is written, but no way to get to it---in fact, D2 still appears as a free block
  - As if write never occurred
- Just the updated inode (Iv2) is written to disk
  - If we follow the pointer, we read garbage
  - *file system inconsistency*: data bitmap says block is free, while inode says it is used. Must be fixed
- Just the updated bitmap is written to disk
  - *file system inconsistency*: data bitmap says data block is used, but no inode points to it.
  - No idea which file should contain the data block!

What If Two Writes Succeed?

- Inode and data bitmap updates succeed
  - File system is consistent
  - But reading new block returns garbage
- Inode and data block updates succeed
  - File system inconsistency
  - Must be fixed
- Data bitmap and data block succeed
  - File system inconsistency
  - No idea which file should contain the data block!

Consistency Issues: Multiple Updates

- Several file system operations update multiple data structures
  - Move a file between directories
    - Delete file from old directory
    - Add file to new directory
  - Create a new file
    - Allocate space on disk for header and data
    - Write a new header to disk
    - Add new file to a directory
    - Write data to disk
- What if the system crashes in the middle?

iClicker Question

Which is a metadata consistency problem?

A. Null double indirect pointer
B. File created before a crash is missing
C. Free block list contains a file data block that is pointed to by an inode
D. Directory contains a corrupt file name

UNIX Approach to Consistency: Metadata

- To keep metadata consistent, UNIX uses synchronous write-through
- If multiple updates are needed, they are performed in a specific order
- If a crash occurs:
  - Run `fsck` to scan entire disk for consistency
  - Check for "in progress" operations and fix up problems

Example: File Create

- Write data block
- Update inode
- Update inode bitmap
- Update data bitmap
- Update directory
- If directory needed another data block:
  - update data bitmap
  - update directory inode

On Crash: fsck

- Scans entire disk for inconsistencies
- Prior to update of inode bitmap: writes disappear
- Data block referenced in inode, but not in data bitmap: update data bitmap
- File created but not in any directory: delete file

UNIX Approach to Consistency: Regular Data

- UNIX uses asynchronous write-back for user data
  - Write-back forced after fixed time intervals (30 sec)
  - Can lose data written within time interval
  - User can also issue a `sync` command to force the OS to send all outstanding writes to disk
- Does not guarantee blocks are written to disk in any particular order
- User programs that care about consistency and reliability store new versions of data in temporary files and replace older version only when user commits
  - Rely on metadata consistency

UNIX Approach: Issues

- Ad hoc approach
- Need to get reasoning exactly right
- Synchronous writes lead to poor performance
- Recovery is sloooow: must scan entire disk

UNIX Approach: Another Problem

- What if we need multiple file operations to occur as a unit?
  - If you transfer money from one account to another, you need to update the two account files as a unit!
- What if we need *atomicity*?

Solution: *Transactions*

Transactions (Review)

- *Transactions* group actions together so that they are:
  - *atomic*: they all happen or they all don't
  - *serializable*: transactions appear to happen one after the other
  - *durable*: once it happens, it sticks
- Critical sections give us atomicity and serializability, but not durability

Achieving Durability (Review)

To get durability, we need to be able to:

- *Commit*: indicate when a transaction is finished
- *Roll back*: recover from an *aborted* transaction
  - If we have a failure in the middle of a transaction, we need to be able to undo what we have done so far
- In other words, we do a set of operations tentatively.
  - If we get to the commit stage, we are okay.
  - If not, roll back operations as if the transaction

Implementing Transactions (Review)

- Key idea: Turn multiple disk updates into a single disk write!

```
begin transaction
    x = x + 100
    y = y – 100
Commit
```

- Keep *write-ahead* (or *redo*) log on disk of all changes in the transaction
- The log records everything the OS does (or tries!) to do
- Once the OS writes both changes on the log, the transaction is committed
- Then *write-behind* changes to the disk, logging all writes
- If the crash comes after a commit, the log is replayed

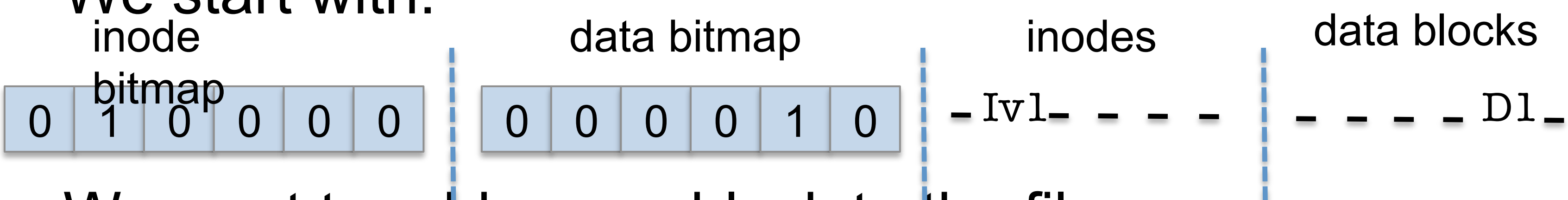
Transactions in File Systems

Most file systems now use *write-ahead logging*

- known as *journaling file systems*
- write all metadata changes to a transaction log before sending any changes to disk
  - file changes are: update directory, allocate blocks, etc.
  - transactions are: create directory, delete file, etc.
- eliminates the need for `fsck` after a crash
- In the event of a crash, read the log.
  - If no log, then all updates made it to disk, do nothing
  - If the log is not complete (no commit), do nothing
  - If the log is completely written (committed), apply any changes that are left to disk

Data Journaling: An Example

- We start with:



- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks: `TxBegin` | `Iv2` | `B2` | `D2` | `TxEnd`
    - Write each record to a block, so it is atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What happens if we crash before the log is updated?
  - no commit, nothing to disk---ignore changes!
- What happens if we crash after the log is updated?
  - replay changes in log back to disk

Data Journaling: An Example Plain Text

- We start with:
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 0
  - Inodes: _ [v] _ _ _ _
  - Data blocks: _ _ _ _ D1 _
- We want to add a new block to the file
- 3 easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | B2 | D2 | TxEnd
    - Write each record to a block, so it's atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What happens if we crash before the log is updated?
  - No commit, nothing to disk---ignore changes!
- What happens if we crash after the log is updated?
  - Replay changes in log back to disk

Journaling and Write Order

- Issuing the 5 writes to the log `TxBegin | Iv2 | B2 | D2 | TxEnd` sequentially is slow
- Issue at once and transform in a single sequential write
- Problem: disk can schedule writes out of order
  - First write `TxBegin`, `Iv2`, `B2`, `TxEnd`
  - Then write `D2`
- Syntactically, transaction log looks fine, even with nonsense in place of `D2`!
- Set a Barrier before `TxEnd`
  - `TxEnd` must block until data on disk

Transactions in File Systems

- Advantages:
  - Reliability
  - Asynchronous write-behind
- Disadvantages:
  - All data is written twice!

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
  - Small writes are expensive on RAID (more soon)
    - Expensive to update a single block (4 disk I/O) but efficient for entire stripes
  - 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

iClicker Question

Where on disk would you put the journal for a journaling file system?

A. Anywhere
B. Outer rim
C. Inner rim
D. Middle
E. Wherever the inodes are

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 file 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.

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 2 help information is posted to Piazza
  - You must show us a working Project 2
- Project 3 is posted due Friday, 4/17