CS 370: OPERATING SYSTEMS
[FILE SYSTEMS]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- How are files recovered if the drive is accidentally formatted?
- Are file systems optimized for the storage type?
- BIOS and UEFI: BIOS will disappear in 2020
  - Unified Extensible Firmware Interface (UEFI) Spec in 2007
- Difference between data and information
- Does the OS manage fragmentation when reading/writing data?
- Who manages the actual writes at the end? OS or device?

Topics covered in this lecture

- Block Allocations
  - Contiguous allocations
  - Linked allocations
  - Indexed allocations
    - iNodes
- Free space management
- Memory mapped files

Allocation methods: Objective and approaches

- How to allocate space for files such that:
  - Disk space is utilized effectively
  - File is accessed **quickly**
- Major Methods
  - Contiguous
  - Linked
  - Indexed

What's in a name? That which we call a rose
By any other name would smell as sweet.
—Juliet
Romeo and Juliet (II, ii, 1-2)
(Shakespeare)

NOMENCLATURE

Terminology

- Storage hardware arranges data in **sectors** (for magnetic disk) or **pages** (for flash)
- File systems often group together a *fixed number* of disk sectors or flash pages into a larger allocation unit called a **block**.
  - E.g.: format file system to run on a disk with 512b sectors to use 4 KB blocks
- Windows FAT and NTFS refer to blocks as clusters
- **File Control Block** (FCBs) organize info about blocks comprising a file
  - iNode in UFS and MFT Record in NTFS; Master File Table (MFT)

CONTIGUOUS ALLOCATIONS

Contiguous Allocation

- Each file occupies a set of **contiguous** blocks on the disk
  - If file is of size **n** blocks and starts at location **b**
    - Occupies blocks **b**, **b+1**, …, **b+n-1**
- Disk head movements
  - None for moving from block **b** to (**b+1**)
  - Only when moving to a different track

Sequential and direct access in contiguous allocations

- Sequential accesses
  - Remember *disk address* of the last referenced block
  - When needed, read the next block
- **Direct access** to block **i** of file that starts at block **b** ?

  b + i

Contiguous allocations suffer from external fragmentation

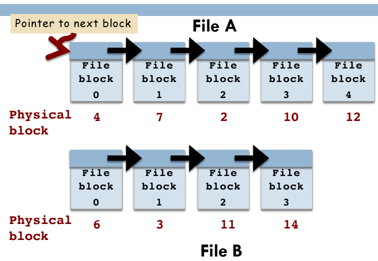
- Free space is broken up into chunks
  - Space is **fragmented**
- Largest continuous chunk may be insufficient for meeting request
- **Compaction** is very slow on large disks
  - Needs several hours

Determining how much space is needed for a file is another problem

- **Preallocate** if eventual size of file is known?
  - Inefficient if the file grows very slowly
    - Much of the allocated space is unused for a long time
- Modified contiguous allocation scheme
  - Allocate space in a continuous chunk initially
  - When space runs out allocate another set of chunks (**extent**)

LINKED ALLOCATIONS

Linked Allocation: Each file is a linked list of disk blocks



Linked List Allocations: Advantages

- **Every** disk block can be used
  - No space is lost in external fragmentation
- Sufficient for directory entry to merely store *disk address of first block*
  - Rest can be found starting there

Linked List Allocation: Disadvantages

- Used effectively only for sequential accesses
  - Extremely **slow random access**
- Space in each block set aside for pointers
  - Each file requires *slightly more space*
- Reliability
  - What if a pointer is lost or damaged?

Linked List Allocations: Reading and writing files is much less efficient

- Amount of data storage in block is no longer a **power of two**
  - Pointer takes up some space
- **Peculiar size** is less efficient
  - Programs read/write in blocks that is a power of two

Linked list allocation: Take pointers from disk block and put in table

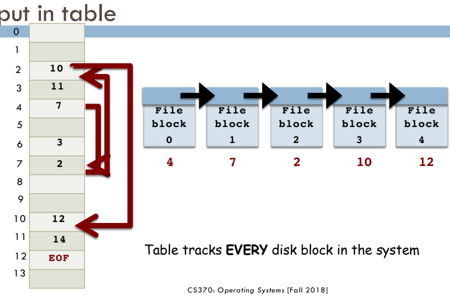


Table tracks **EVERY** disk block in the system

Linked list allocation using an index

- **Entire** disk block is available for data
- Random access is much easier
  - Chain must still be followed
    - But this chain could be *cached in memory*
- MS-DOS and OS/2 operating systems
  - Use such a file allocation table (FAT)

Linked list allocation using an index: Disadvantages

- Table must be cached **in memory** for efficient access
- A large disk will have a large number of data blocks
  - Table consumes a large amount of physical memory

Indexed Allocations

Indexed allocations

- Bring all pointers together into one location
  - **index block**
- Each file has its ***own*** index block
  - Directory contains address of the index block

Indexed allocation supports direct access without external fragmentation

- Every disk block can be utilized
  - **No external fragmentation**
- Space wasted by pointers *is generally higher* than linked listed allocations
  - Example: File has two blocks
    - Linked listed allocations: 2 pointers are utilized
    - Indexed allocations: Entire index block must be allocated

iNodes

inode

- **Fixed-length** data structure
  - One per file
- Contains information about
  - File **attributes**
    - Size, owner, creation/modification time etc.
  - **Disk addresses**
    - File blocks that comprise file

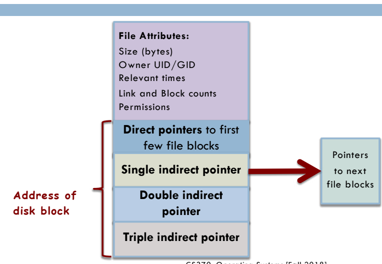
inode

- The inode is used to encapsulate information about a large number of file blocks.
- For e.g.
  - Block size = 8 KB, and file size = 8 GB
  - There would be a million file-blocks
    - inode will store info about the **pointers to these blocks**
  - inode allows us to access info for *all* these blocks
    - Storing pointers to these file blocks also takes up storage

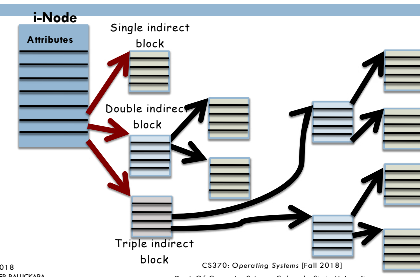
Managing information about data blocks in the inode:

- The first **few** pointers to the data blocks of the file stored in the inode
- If the file is large: **Indirect** pointer
  - To a block of pointers that point to additional data blocks
- If the file is larger: **Double indirect** pointer
  - Pointer to a block of indirect pointers
- If the file is huge: **Triple indirect** pointer
  - Pointer to a block of double-indirect pointers

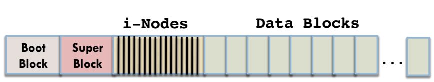
Schematic structure of the inode



i-Node: How the pointers to the file blocks are organized



Disk Layout in traditional UNIX systems



An integral number of inodes fit in a single data block

Super Block describes the state of the file system

- Total size of partition
- Block size and number of disk blocks
- Number of inodes
- List of free blocks
- inode number of the root directory

- Destruction of super block?
  - Will render file system unreadable

A linear array of inodes follows the data block

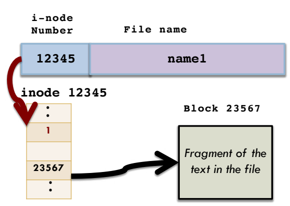
- inodes are numbered from **1** to some **max**
- Each inode is identified by its inode number
  - inode number contains info needed to **locate** inode on the disk
  - Users think of files as filenames
  - UNIX thinks of files in terms of inodes

UNIX directory structure

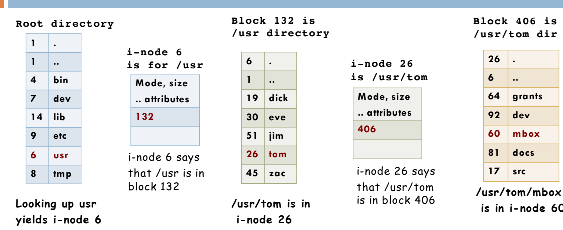
- Contains only file names and the corresponding inode numbers

- Use **ls –i** to retrieve inode numbers of the files in the directory

Directory entry, inode and data block for a simple file



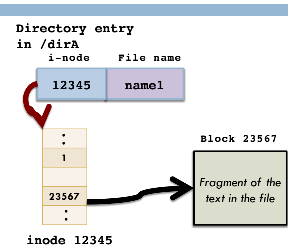
Looking up path names in UNIX Example: /usr/tom/mbox



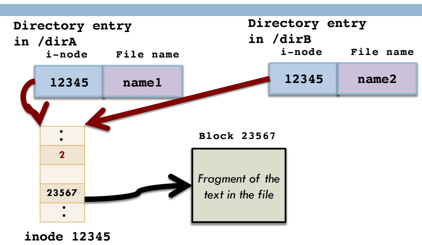
Advantages of directory entries that have name and inode information

- Changing filename only requires changing the directory entry
- Only 1 physical copy of file needs to be on disk
  - File may have several names (or the same name) in different directories
- Directory entries are small
  - Most file info is kept in the inode

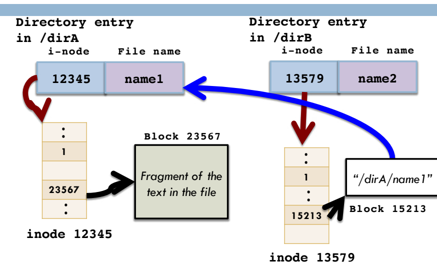
Two hard links to the same file



Two hard links to the same file



File with a symbolic link



Maximum size of your hard disk (8 KB blocks and 32-bit pointers)

- 32-bit pointers can address 2^{32} blocks
- At 8 KB per-block
  - Hard disk can be $2^{13} \times 2^{32} = 2^{45}$ bytes (32 TB)

The case for larger block sizes

- Larger partitions for a fixed pointer size
- Retrieval is more efficient
  - Better system throughput
- Problem
  - Internal fragmentation

Limitations of a file system based on inodes

- File **must fit** in a single disk partition
- Partition size and number of files are **fixed** when system is set up

inode preallocation and distribution

- inodes are **preallocated** on a volume
  - Even on empty disks % of space lost to inodes
- Preallocating inodes and spreading them
  - Improves performance
- Keep file's data block **close** to its inode
  - Reduce seek times

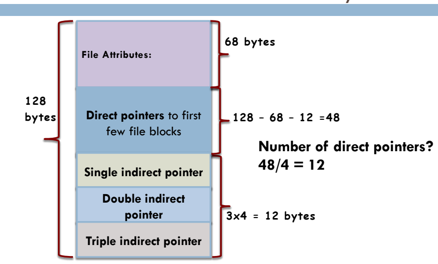
Checking up on the iNodes: The df –i command (*disk free*)

- inode statistics for a given set of file systems
  - Total, free and used inodes

df -i /s/bach/*

Filesystem	Inodes	IUsed	IFree	IUse%
/dev/cciss/c0d1p1	12746752	948362	11798390	8%
/dev/cciss/c0d2p1	10240000	150436	10089564	2%
/dev/cciss/c0d3p1	10240000	812727	9427273	8%
/dev/cciss/c0d4p1	10240000	930080	9309920	10%
/dev/cciss/c0d5p1	10240000	496744	9743256	5%
/dev/cciss/c0d6p1	10240000	167900	10072100	2%
/dev/cciss/c0d7p1	10240000	748709	9491291	8%
/dev/cciss/c0d8p1	12681216	760002	11921214	6%
/dev/cciss/c0d9p1	12681216	394165	12287051	4%

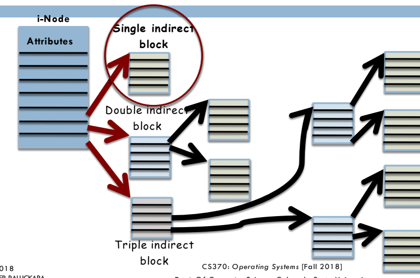
inode: A quantitative look
BLOCK Size = 8 KB and Pointers = 4 bytes



inode: A quantitative look
BLOCK Size = 8 KB and Pointers = 4 bytes

- 12 **direct** pointers to file blocks
- Each file block = 8KB
- Size of file that can be represented with direct pointers
  - 12 x 8 KB = 96 KB

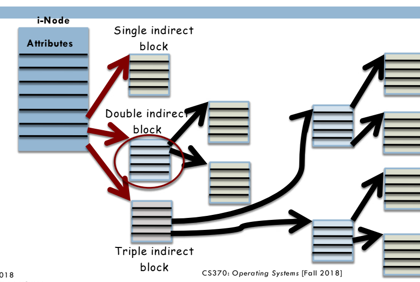
inode



inode: A quantitative look
BLOCK Size = 8 KB and Pointers = 4 bytes

- Block size = 8 KB
- Single indirect block =block size = 8 KB (8192 bytes)
- Number of pointers held in a single-indirect-block
  - Block-size/Pointer-size
  - 8192/4 = 2048
- With single-indirect pointer
  - Additional 2048 x 8 KB = $2^{11} \times 2^{3} \times 2^{10} = 2^{24}$ (16 MB) of a file can be addressed

inode



inode: A quantitative look
BLOCK Size = 8 KB and Pointers = 4 bytes

- With a **double indirect pointer** in the inode
  - The double-indirect block has 2048 pointers
    - Each pointer points to a different single-indirect-block
    - So, there are 2048 single-indirect blocks
  - Each single-indirect block has 2048 pointers to file blocks
- Double indirect addressing allows the file to have an additional size of
  - $2048 \times 2048 \times 8 \text{ KB} = 2^{11} \times 2^{24} = 2^{35}$ .... (32 GB)

inode: A quantitative look
BLOCK Size = 8 KB and Pointers = 4 bytes

- **Triple indirect addressing**
  - Triple indirect block points to 2048 double indirect blocks
  - Each double indirect block points to 2048 single indirect block
  - Each single direct block points to 2048 file blocks
- Allows the file to have an additional size of
  - $2048 \times 2048 \times 2048 \times 8 \text{ KB} = 2^{11} \times 2^{35} = 2^{46}$ (64 TB)

Limits of triple indirect addressing

- In our example:
  - There can be 2048 x 2048 x 2048 data blocks
  - *i.e.*, $2^{11} \times 2^{11} \times 2^{11} = 2^{33}$
  - Pointers would need to be longer than 32-bits to fully address this storage

What if we increase the size of the pointers to 64-bits (data block is still 8 KB) ?

- What is the maximum size of the file that we can hold?
- 8 KB data block can hold (8192/8) = 1024 pointers
- **Single indirect** can add
  - $1024 \times 8 \text{ KB} = 2^{10} \times 2^{3} \times 2^{10} = 2^{23}$ (8MB) of additional bytes to the file

What if we increase the size of the pointers to 64-bits (data block is still 8 KB)?

- **Double indirect** addressing allows the file to have an additional size of
  - $1024 \times 1024 \times 8 \text{ KB} = 2^{10} \times 2^{23} = 2^{33}$ .... (8 GB)
- **Triple indirect** addressing allows the file to have an additional size of
  - $1024 \times 1024 \times 1024 \times 8 \text{ KB} = 2^{10} \times 2^{33} = 2^{43}$ (8 TB)

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 11]
- *Andrew S Tanenbaum amd Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 4]*
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 4]
- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. Recursive Books. ISBN: 978-0985673529.* [Chapter 12]