COS 318: Operating Systems

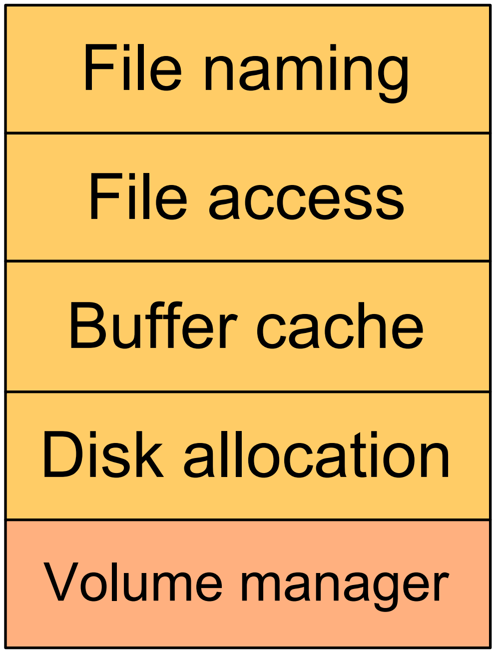
File Layout and Directories

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

File System

- Naming
  - File name and directory
- File access
  - Read, write, other operations
- Buffer cache
  - Reduce client/server disk I/Os
- Disk allocation
  - Layout, mapping files to blocks
- Volume manager
  - Storage layer including RAID
  - Block storage interface


File naming

File access

Buffer cache

Disk allocation

Volume manager

Topics

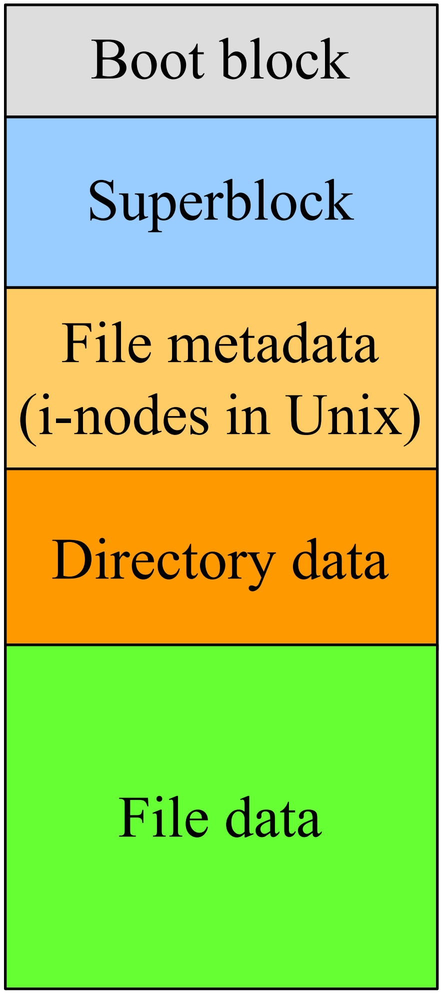
- File system structure
- Disk allocation and i-nodes
- Directory and link implementations
- Physical layout for performance

Master Boot Record

- Starts at first sector of disk
- End of record lists the partitions on the disk
  - Every partition can have a different file system
- Upon boot:
  - BIOS reads in and executes MBR
  - Finds active disk partition from MBR
  - First block of active partition (boot block) is loaded and executed
  - That loads in the OS from that partition
- What does partition and file layout on it look like?

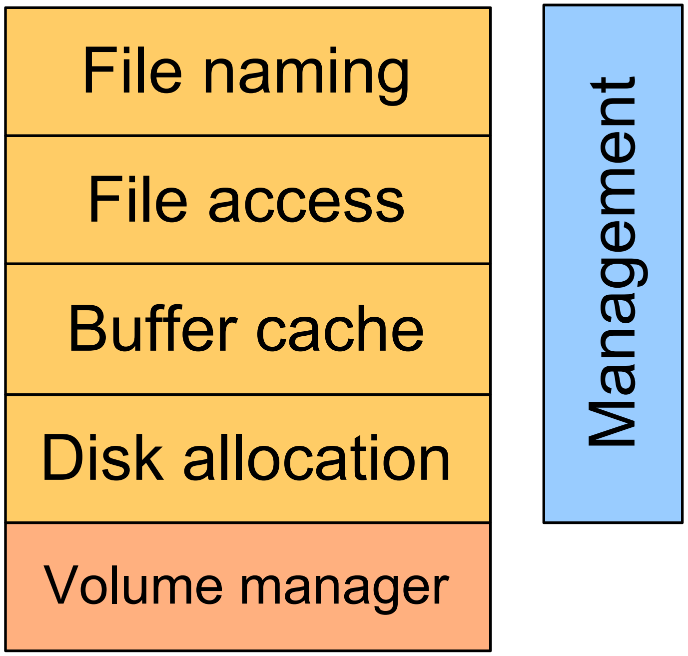
Typical Layout of a Disk Partition

- Boot block
  - Code to load and boot OS
- Super-block defines a file system
  - File system info: type, no of blocks, ...
  - File metadata area
  - Information about free blocks
- File metadata
  - Each descriptor describes a file
- Directories
  - Directory data (directory and file names)
- File data
  - Data blocks



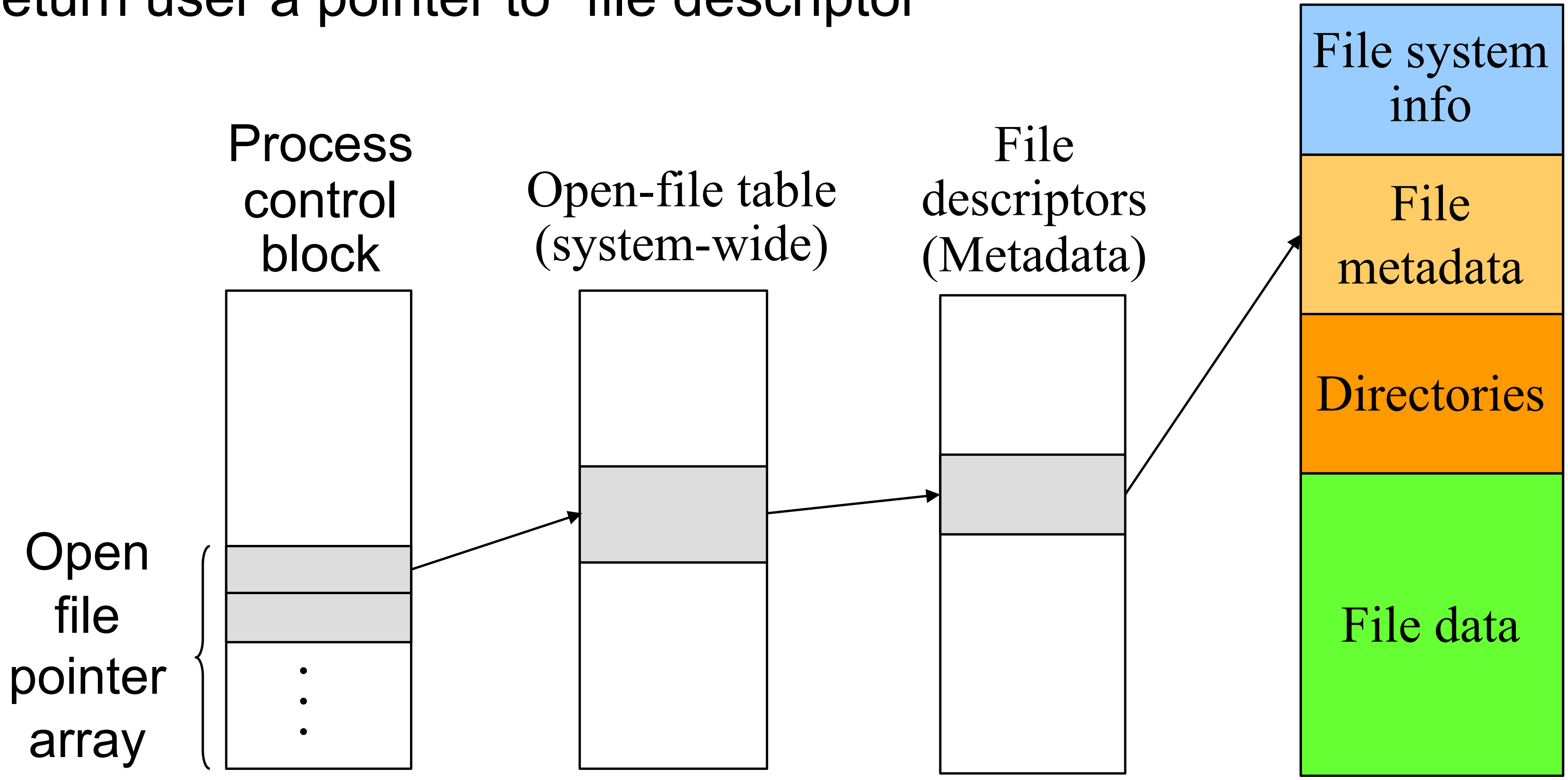
Software Components

- Naming
  - File name and directory
- File access
  - Read, write, other operations
- Buffer cache
  - Reduce client/server disk I/Os
- Disk allocation
  - Layout, mapping files to blocks
- Volume manager
  - Storage layer including RAID
  - Block storage interface
- Management
  - Tools for system administrators to manage file systems



Open A File: Open(fd, name, access)

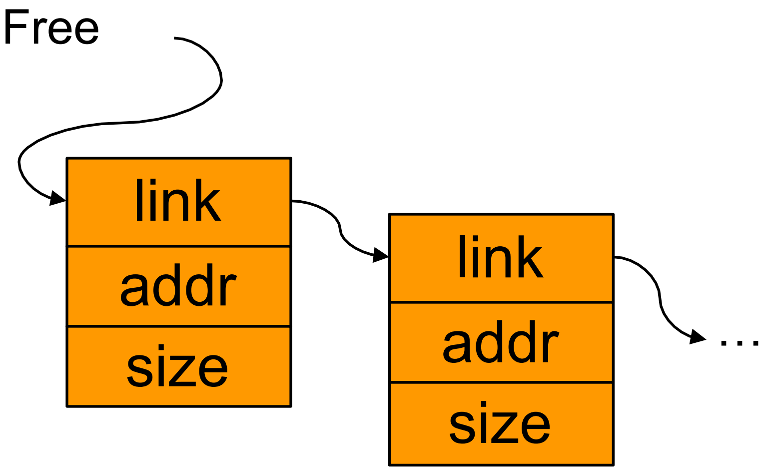
- Various checking (directory and file name lookup, authenticate)
- Copy the file descriptors into the in-memory data structure
- Create an entry in the open file table (system wide)
- Create an entry in PCB
- Return user a pointer to “file descriptor”



Data Structures for Storage Allocation

- A File
  - Metadata
  - A list of data blocks
- Free space data structure
  - Bit map indicating the status of disk blocks
  - Linked list that chains free blocks together
  - Buddy system
  - …

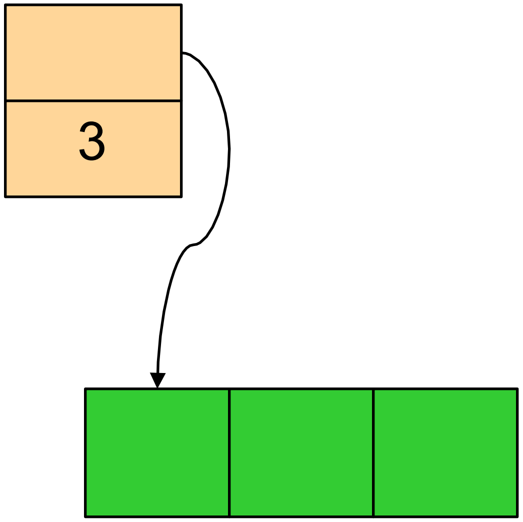
11111111111111111000000000000000
00000111111110000000000111111111
⋮
11000000111100011110000000000000

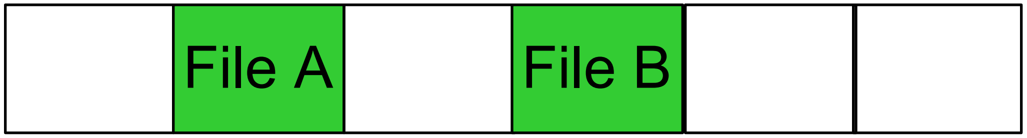


- Let's look at some ways of keeping track of file data

Contiguous Allocation

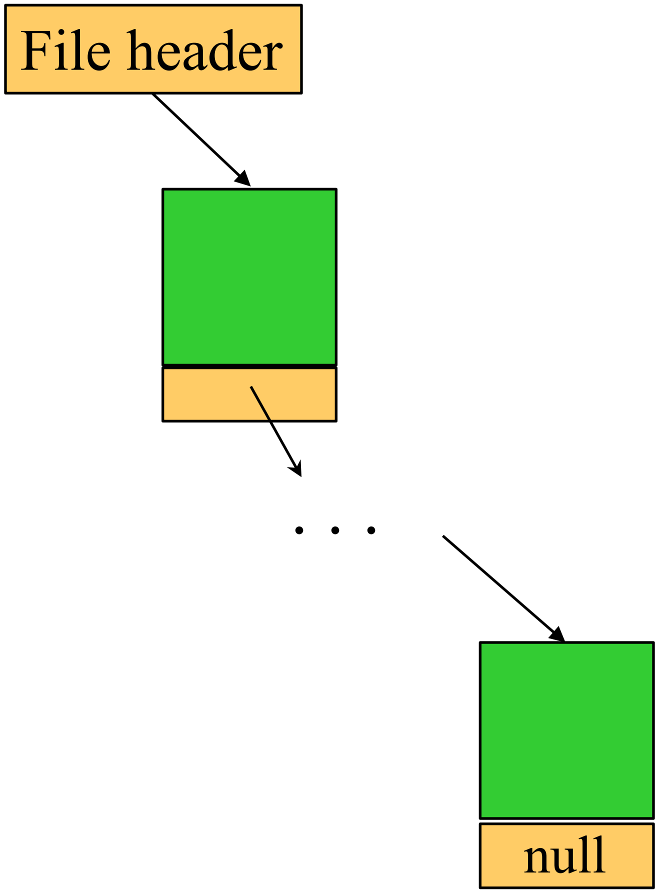
- Allocate contiguous blocks on storage
  - Bitmap: find N contiguous 0's
  - Linked list: find a region (size >= N)
- File metadata
  - First block in file
  - Number of blocks
- Pros
  - Fast sequential access to a file
  - Easy random access to locations in file
- Cons
  - External fragmentation
    (what if file C needs 4 blocks)
  - Hard to grow files





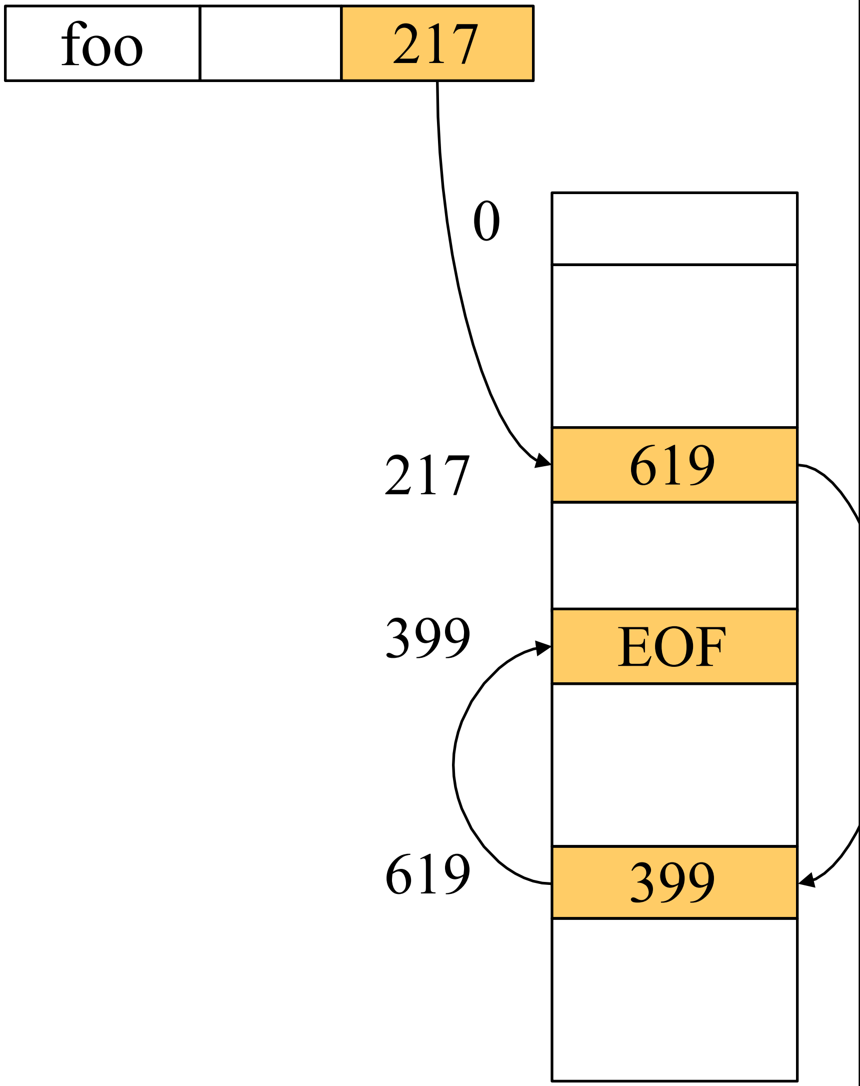
Linked Files (Alto)

- File structure
  - File metadata points to 1st block on storage
  - A block points to the next
  - Last block has a NULL pointer
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
- Cons
  - Random access: bad
  - Unreliable: losing a block means losing the rest



File Allocation Table (FAT)

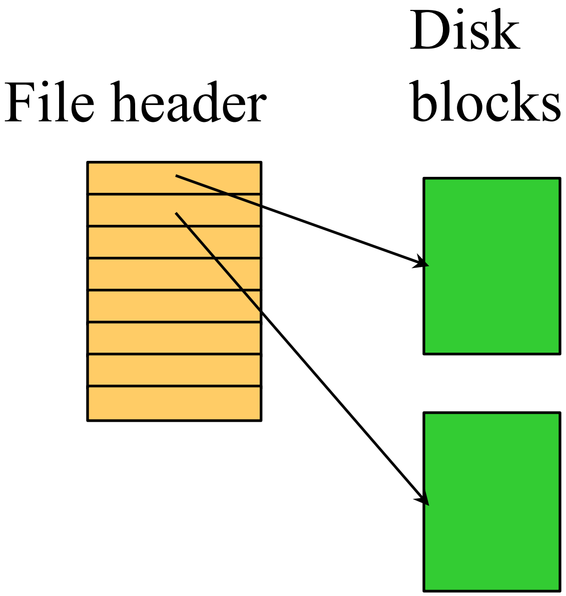
- Idea is to keep the linked list metadata (pointers) in memory
- Allocation table at beginning of each volume
  - N entries for N blocks
  - Want to keep it in memory
- File structure
  - A file is a linked list of blocks
  - File metadata points to first block of file
  - The entry of first block points to next, …
- Pros
  - Simple
- Cons
  - Random access: still not good
  - Wastes space - table for each file expensive to keep in memory



FAT Allocation Table

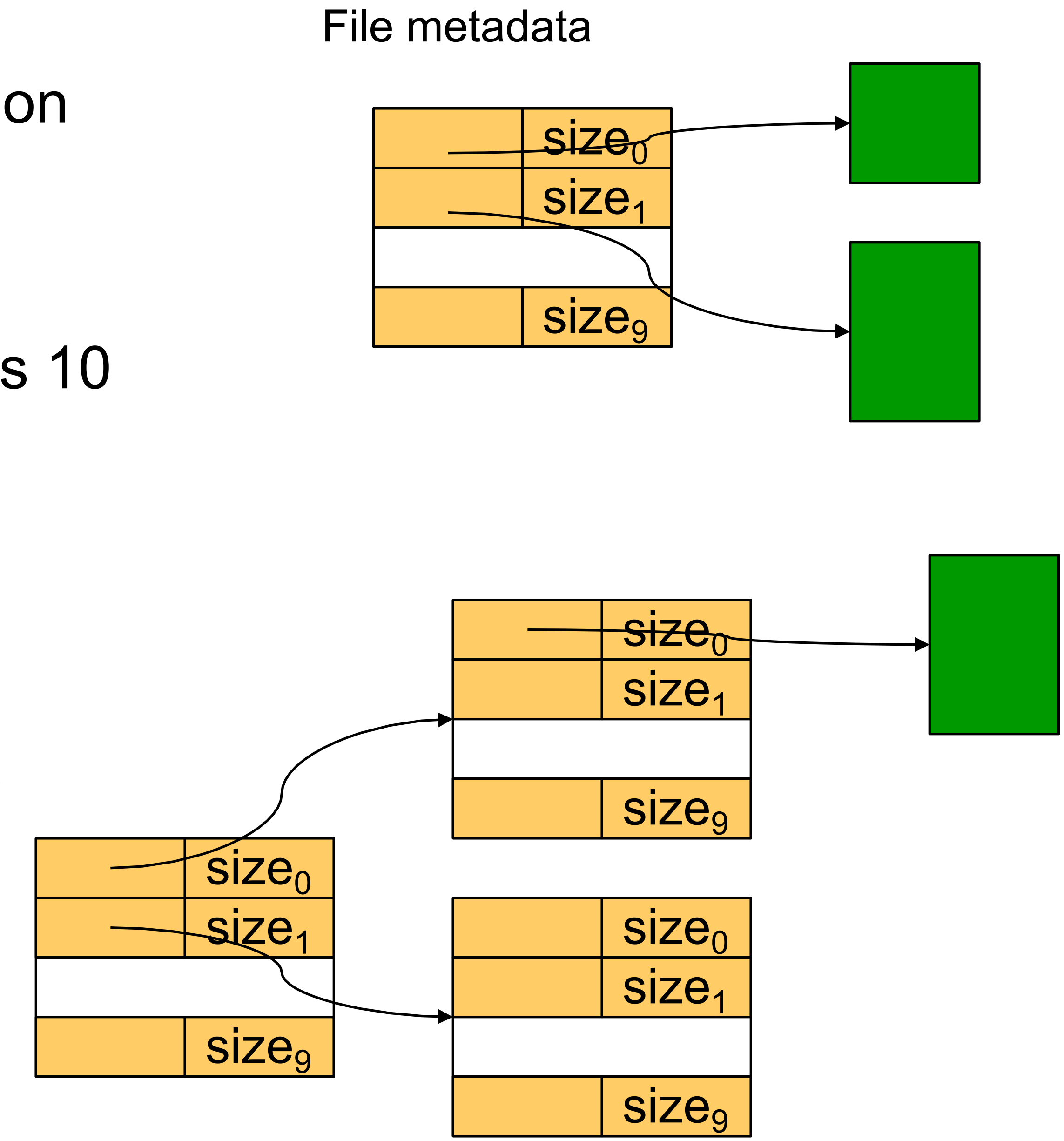
Single-Level Indexed Files

- File structure
  - User declares max size
  - A file header holds an array of pointers to point to disk blocks
- Pros
  - Can grow up to a limit
  - Random access is fast
- Cons
  - Difficult to grow beyond the limit



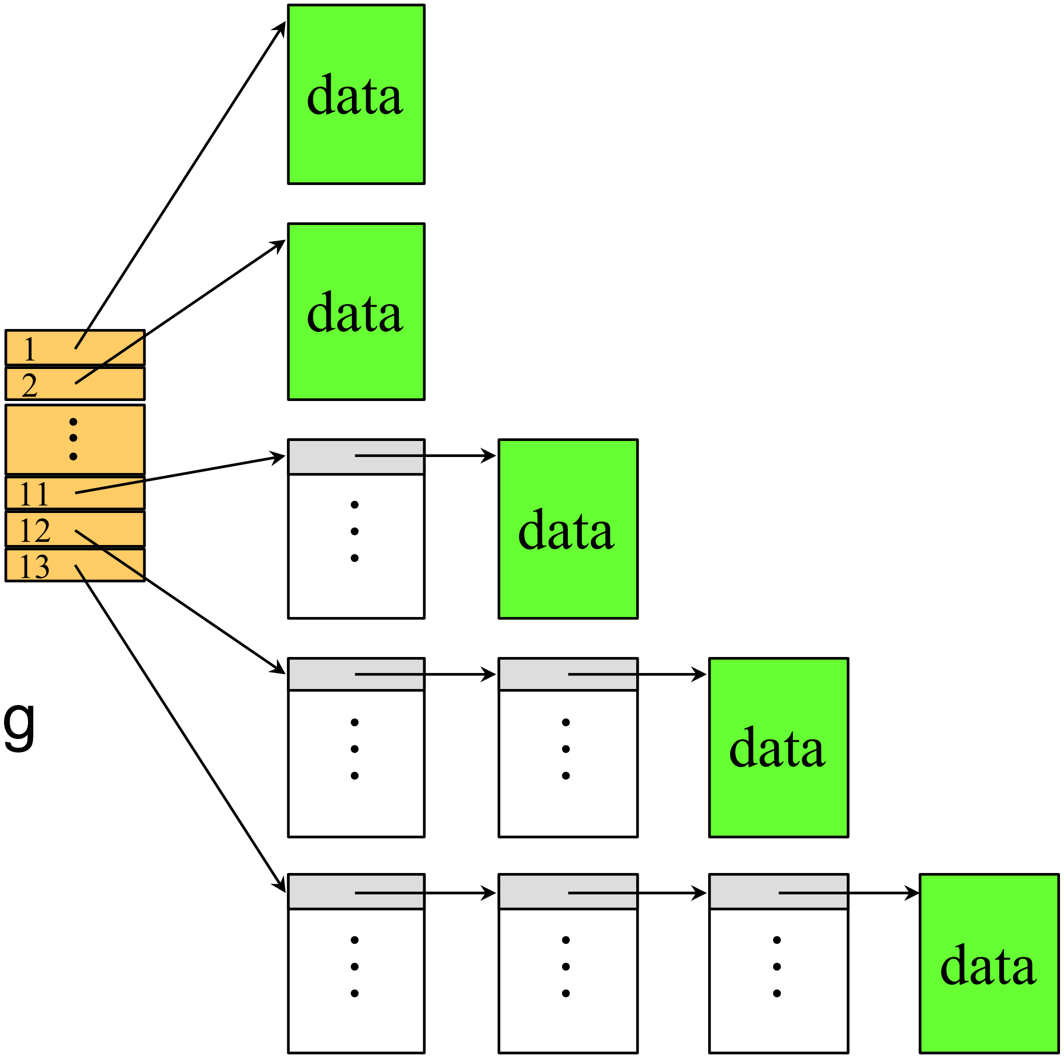
DEMOS (Cray-1)

- Idea
  - Try contiguous allocation
  - Allow non-contiguous
- File structure
  - Small file metadata has 10 (base,size) pointers
  - Big file has 10 indirect pointers
- Pros & cons
  - Can grow (max 10GB)
  - fragmentation



Multi-Level Indexed Files (Unix)

- 13 Pointers in a header
  - 10 direct pointers
  - 11: 1-level indirect
  - 12: 2-level indirect
  - 13: 3-level indirect
- Pros & Cons
  - In favor of small files
  - Can grow
  - Limit is 16G
  - Can have lots of seeking



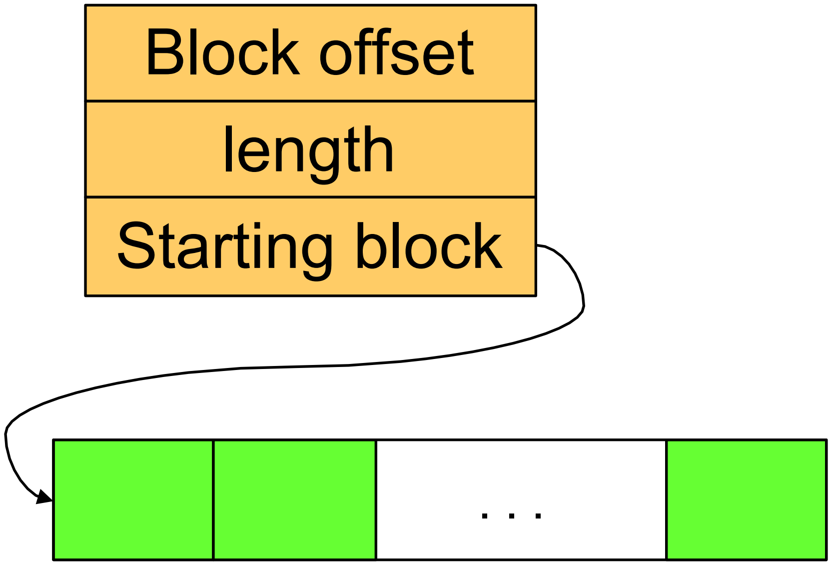
Original Unix i-node

- Mode: file type, protection bits, setuid, setgid bits
- Link count: number of directory entries pointing to this file
- Uid: uid of the file owner
- Gid: gid of the file owner
- File size
- Times (access, modify, change)

- 10 pointers to data blocks
- Single indirect pointer
- Double indirect pointer
- Triple indirect pointer

Extents

- An extent is a variable number of blocks
- Main idea
  - A file is a number of extents
  - XFS uses 8Kbyte blocks
  - Max extent size is 2M blocks
- Index nodes need to have
  - Block offset
  - Length
  - Starting block
- Microsoft NTFS, Linux EXT4, …
- Pros and Cons?



Block offset

length

Starting block

Naming

Can name files via:

- Text name
  - Need to map it to index
- Index (i-node number)
  - Ask users to specify i-node number
- Icon
  - Need to map it to index or map it to text then to index

Directory Organization Examples

- Flat (CP/M)
  - All files are in one directory
- Hierarchical (Unix)
  - /u/cos318/foo
  - Directory is stored in a file containing (name, i-node) pairs
  - The name can be either a file or a directory
- Hierarchical (Windows)
  - C:\windows\temp\foo
  - File extensions have meaning (unlike in Unix). Use the extension to indicate whether the entry is a directory

Mapping File Names to i-nodes

Need to support the following types of operations:

- Create/delete
  - Create/delete a directory
- Open/close
  - Open/close a directory for read and write
  - Should this be the same or different from file open/close?
- Link/unlink
  - Link/unlink a file
- Rename
  - Rename the directory

Linear List

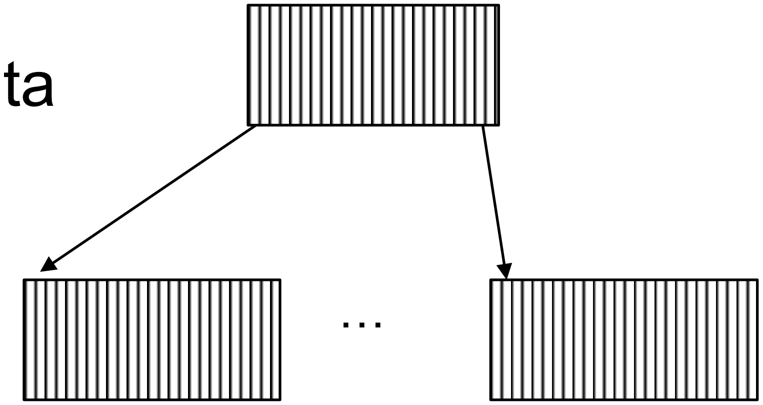
- Method
  - <FileName, i-node> pairs are linearly stored in a file
  - Create a file
    - Append <FileName, i-node>
  - Delete a file
    - Search for FileName
    - Remove its pair from the directory
    - Compact by moving the rest
- Pros
  - Space efficient
- Cons
  - Linear search
  - Need to deal with fragmentation

/u/jps
foo bar …
veryLongFileName

<foo,1234> <bar, 1235> … <veryLongFileName, 4567>

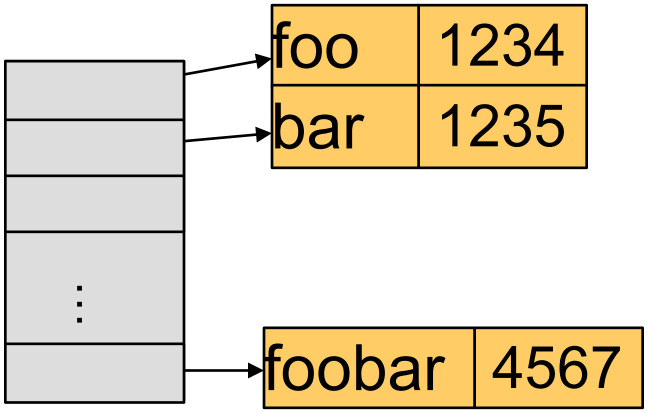
Tree Data Structure

- Method
  - Store <fileName, i-node> a tree data structure such as B-tree
  - Create/delete/search in the tree data structure
- Pros
  - Good for a large number of files
- Cons
  - Inefficient for a small number of files
  - More space
  - Complex



Hashing

- Method
  - Use a hash table to map FileName to i-node
  - Space for name and metadata is variable sized
  - Create/delete will trigger space allocation and free
- Pros
  - Fast searching and relatively simple
- Cons
  - Not as efficient as trees for very large directory (wasting space for the hash table)

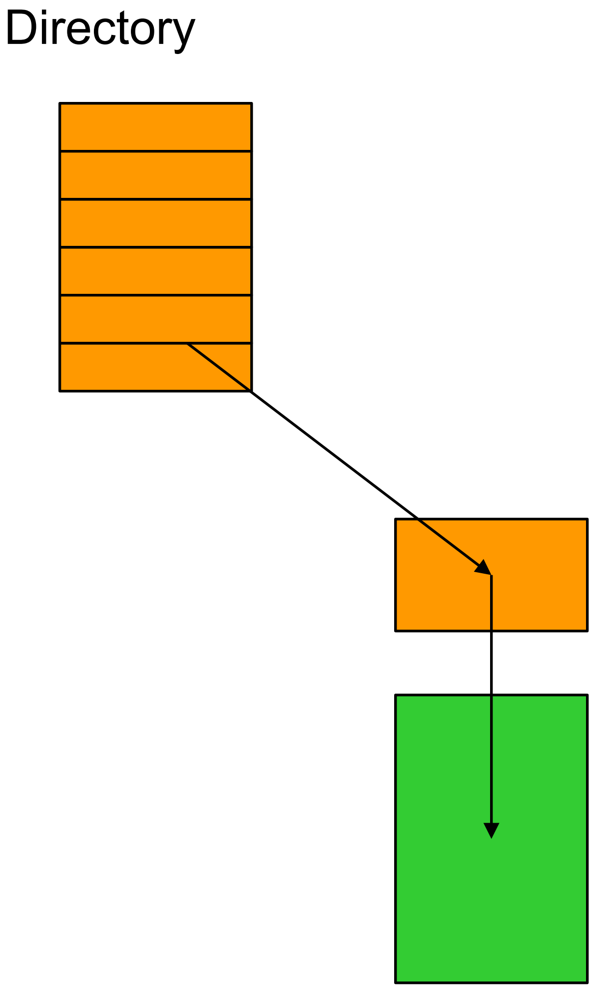


Number of I/O operations

- I/Os to access a byte of /u/cos318/foo
  - Read the i-node and first data block of “/”
  - Read the i-node and first data block of “u”
  - Read the i-node and first data block of “cos318”
  - Read the i-node and first data block of “foo”
- I/Os to write a file
  - Read the i-node of the directory and the directory file (as above)
  - Read or create the i-node of the file
  - Read or create the file itself
  - Write back the directory and the file
- Too many I/Os to traverse the directory
  - Solution is to use ***Current Working Directory***

Hard Links

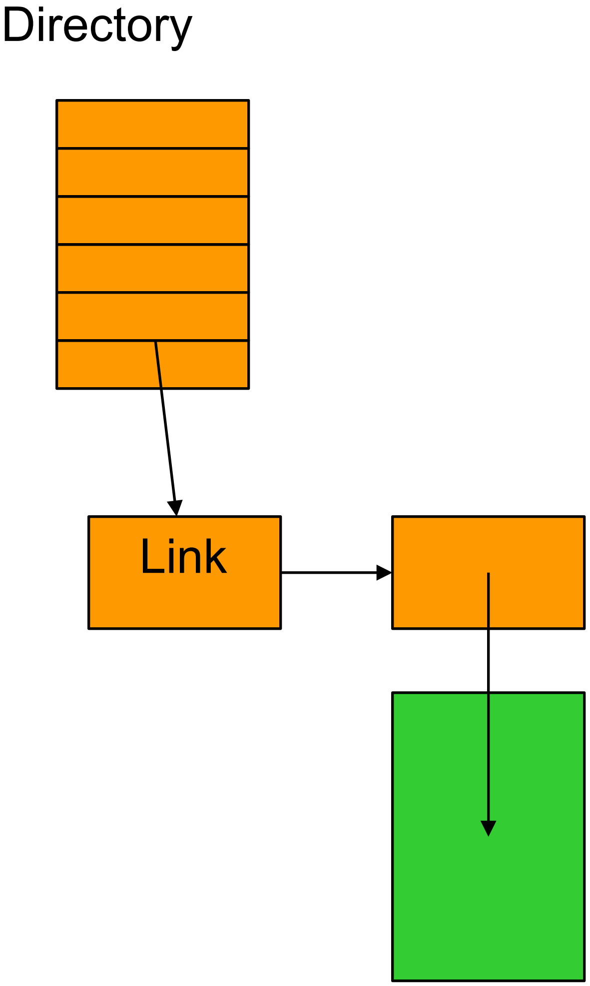
- Approach
  - A link to a file with the same i-node
    `ln source target`
  - Delete may or may not remove the target depending on whether it is the last one (link reference count)
- Benefits of hard links?
- Main issue with hard links?



Symbolic Links

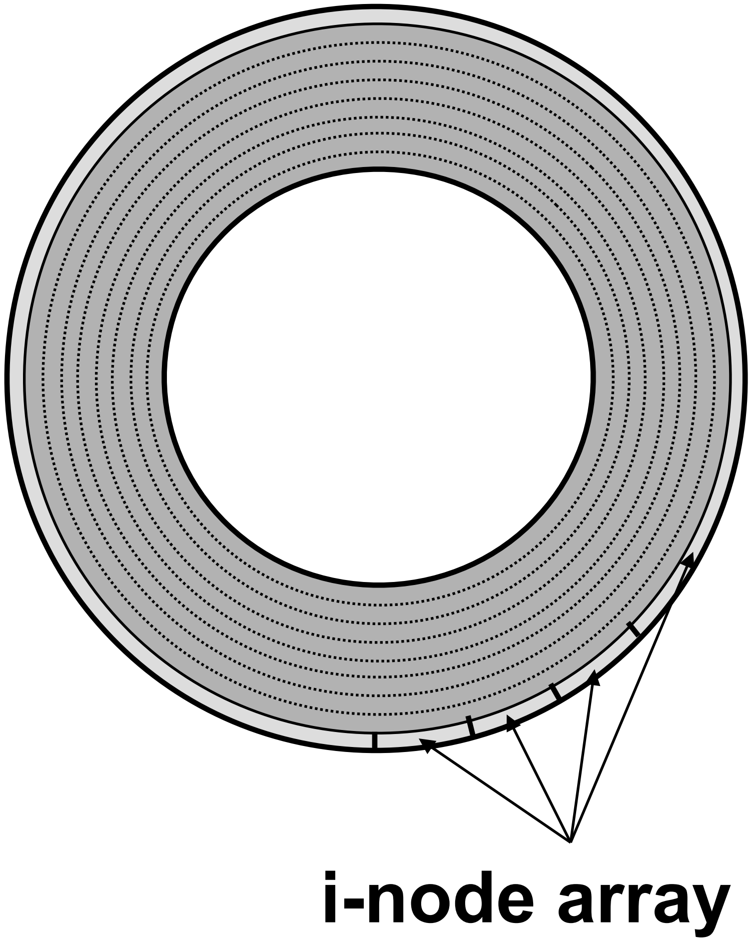
- Approach
  - A symbolic link is a pointer to a file
  - Use a new i-node for the link

    `ln –s source target`
- Benefit of symbolic links?
- Main issue with symbolic links?



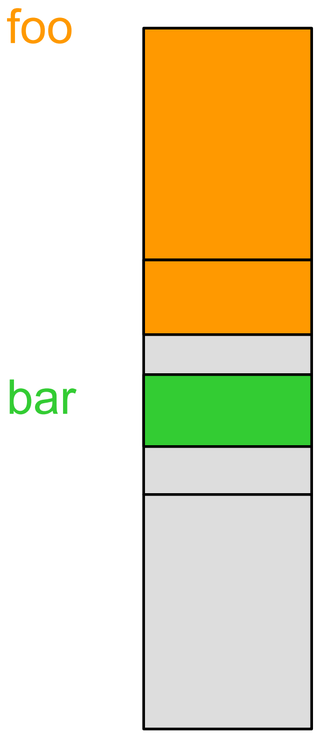
Original Unix File System

- Simple disk layout
  - Block size is sector size (512 bytes)
  - i-nodes are on outermost cylinders
  - Data blocks are on inner cylinders
  - Use linked list for free blocks
- Issues
  - Index is large
  - Fixed max number of files
  - i-nodes far from data blocks
  - i-nodes for directory not close together
  - Consecutive blocks can be anywhere
  - Poor bandwidth (20Kbytes/sec even for sequential access!)



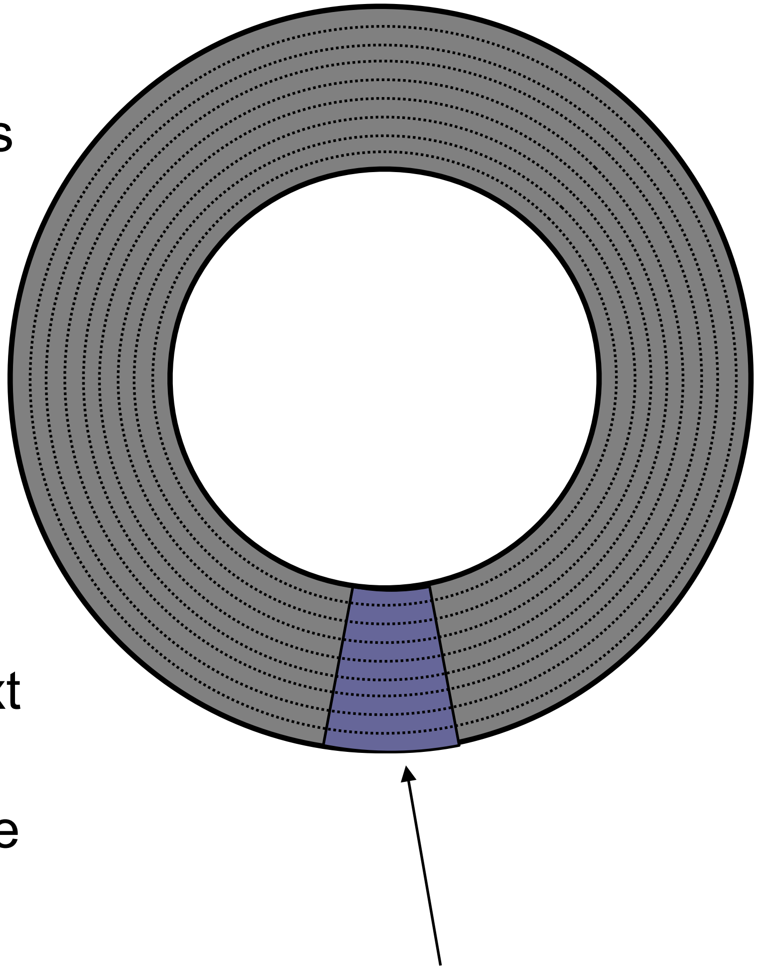
BSD FFS (Fast File System)

- Use a larger block size: 4KB or 8KB
  - Allow large blocks to be chopped into fragments
  - Used for little files and pieces at the ends of files
- Use bitmap instead of a free list
  - Try to allocate contiguously



FFS Disk Layout

- i-nodes are grouped together
  - A portion of the i-node array on each cylinder
  - In same cylinder group as data for the files
  - 10% reserved disk space, to keep room
- Do you ever read i-nodes without reading any file blocks?
  - 4 times more often than reading together
  - examples: ls, make
- Overcome rotational delays
  - Skip sector positioning to avoid the context switch delay
  - Read ahead: read next block right after the first



What Has FFS Achieved?

- Performance improvements
  - 20-40% of disk bandwidth for large files (10-20x original)
  - Better small file performance (why?)
- We can do better
  - Extent based instead of block based
    - Use a pointer and size for all contiguous blocks (XFS, Veritas file system, etc)
  - Synchronous metadata writes hurt small file performance
    - Asynchronous writes with certain ordering (“soft updates”)
    - Logging (talk about this later)
    - Play with semantics (/tmp file systems)

Summary

- File system structure
  - Boot block, super block, file metadata, file data
- File metadata
  - Consider efficiency, space and fragmentation
- Directories
  - Consider the number of files
- Links
  - Soft vs. hard
- Physical layout
  - Where to put metadata and data