



# COS 318: Operating Systems

## File Structure

Computer Science Department  
Princeton University

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



# Where Are We?

---

- Covered:
  - Management of CPU & concurrency
  - Management of main memory & virtual memory
  - Device drivers and storage devices
- Currently --- File Systems
  - This lecture: **File Structure**
- Then:
  - Naming and directories
  - Efficiency and performance
  - Reliability and protection



# The File System Abstraction

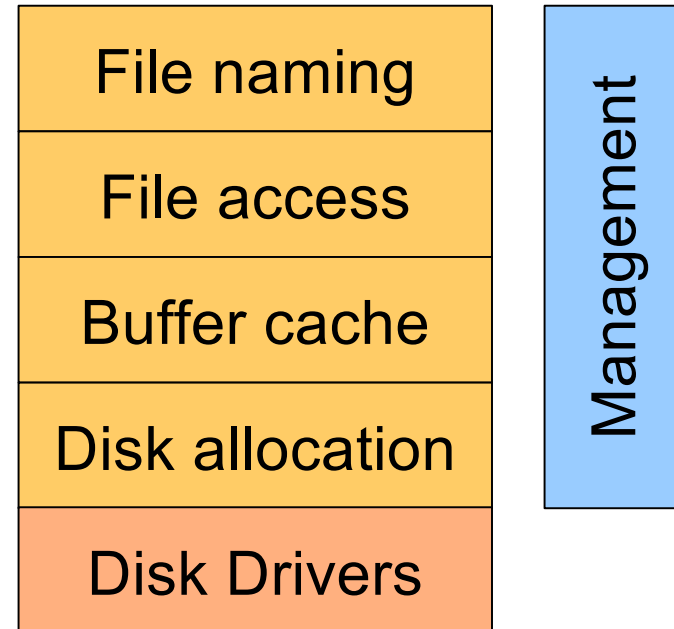
- Open, close, read, write ... named files, arranged in folders or directories

<b>Physical Reality</b>	<b>File System Abstraction</b>
block oriented	byte oriented (char stream)
physical sector #'s	named files
no protection	users protected from each other
data might be corrupted if machine crashes	robust to machine failures



# 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
- ◆ Security, protection, reliability, durability
- ◆ Management tools



# Topics

---



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



# Typical File Attributes

---

- **Name**
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device.
- **Size** – current file size.
- **Protection** – controls who can read, write, execute
- **Time, date, and user identification** – data for protection, security, and usage monitoring



# Master Boot Record

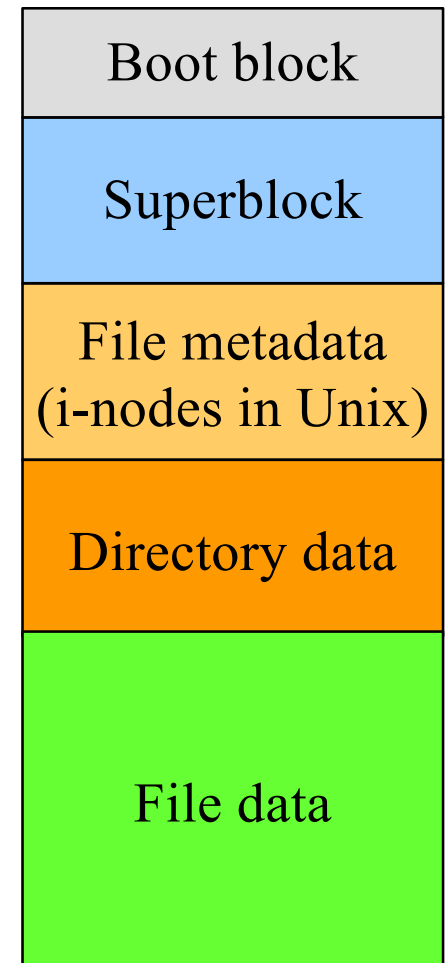
---

- 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 / ptr to free blocks
  - Location of descriptor of root directory
- ◆ File metadata
  - Each descriptor describes a file
- ◆ Directories
  - Directory data (directory and file names)
- ◆ File data
  - Data blocks





# File Types – Name, Extension

File Type	Usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	compiled, machine language, not linked
Source code	c, p, pas, 177, asm, a	source code in various languages
Batch	bat, sh	commands to the command interpreter
Text	txt, doc	textual data documents
Word processor	wp, tex, rrf, etc.	various word-processor formats
Library	lib, a	libraries of routines
Print or view	ps, dvi, gif	ASCII or binary file
Archive	arc, zip, tar	related files grouped into one file, sometimes compressed.



# Typical File Operations

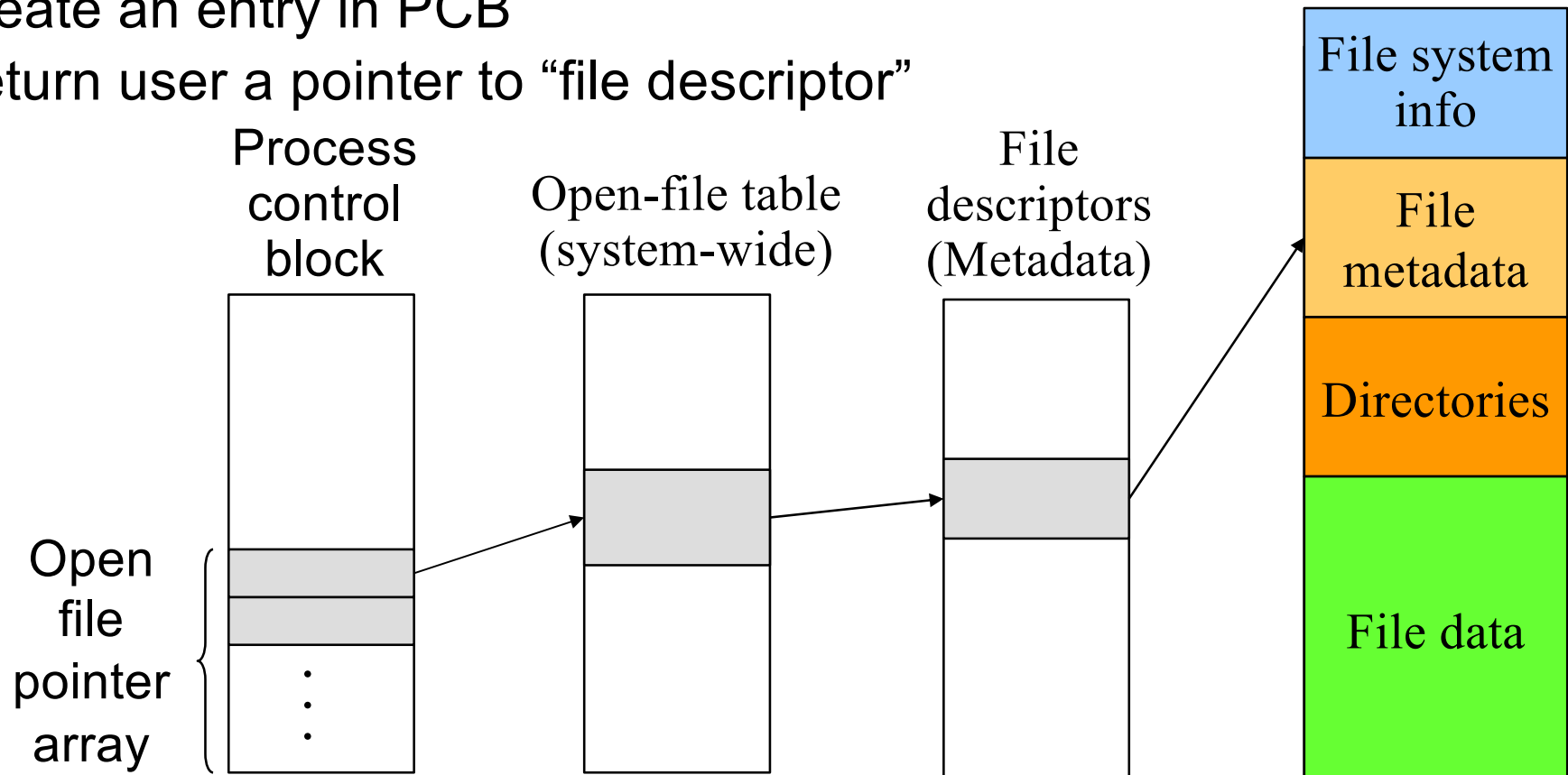
---

- Create
- Write
- Read
- Reposition within file – file seek
- Delete
- Truncate
- $\text{Open}(F_i)$  – search the directory structure on disk for entry  $F_i$ , and move the content of entry to memory.
- $\text{Close}(F_i)$  – move the content of entry  $F_i$  in memory to directory structure on disk.



# 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”



# Translating from user to system view

---

- User wants to read 10 bytes from file starting at byte 2?
  - Seek byte 2, fetch the block, read 10 bytes
- User wants to write 10 bytes to file starting at byte 2?
  - Seek byte 2, fetch the block, write 10 bytes, write out block
- Everything inside file system is in whole size blocks
  - Even getc and putc buffers 4096 bytes
- From now on, file is collection of blocks.



# File usage patterns

---

- How do users access files?
  - Sequential: bytes read in order
  - “Random”: read/write element out of middle of file
  - Content-based access: find me next byte starting with “COS318”
- How are files used?
  - Most files are small
  - Large files use up most of the disk space
  - Most transfers are small
  - Large files account for most of the bytes transferred
- Bad news
  - Need everything to be efficient



# File system design constraints

---

- For small files:
  - Small enough blocks for storage efficiency
  - Files used together should be stored together
- For large files:
  - Contiguous allocation for sequential access
  - Efficient lookup for random access
- May not know at file creation whether file will become small or large



# File system design

---

- Data structures
  - Directories: file name -> file metadata
    - Store directories as files
  - File metadata: used to find file data blocks of the file
  - Free map: list of free disk blocks
- How do we organize these data structures?



# Data structures for disk management

---

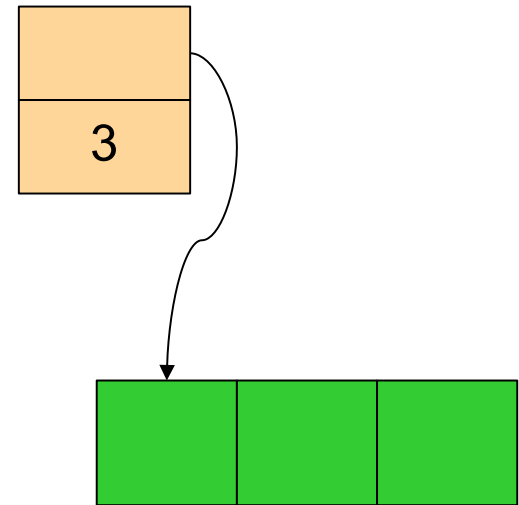
- A file header for each file (part of the file meta-data)
- A data structure to track free space on disk
  - Bit map
    - 1 bit per block (sector)
    - blocks numbered in cylinder-major order, why?
  - Linked list
  - Others?
- What about allocation for the blocks associated with a file?





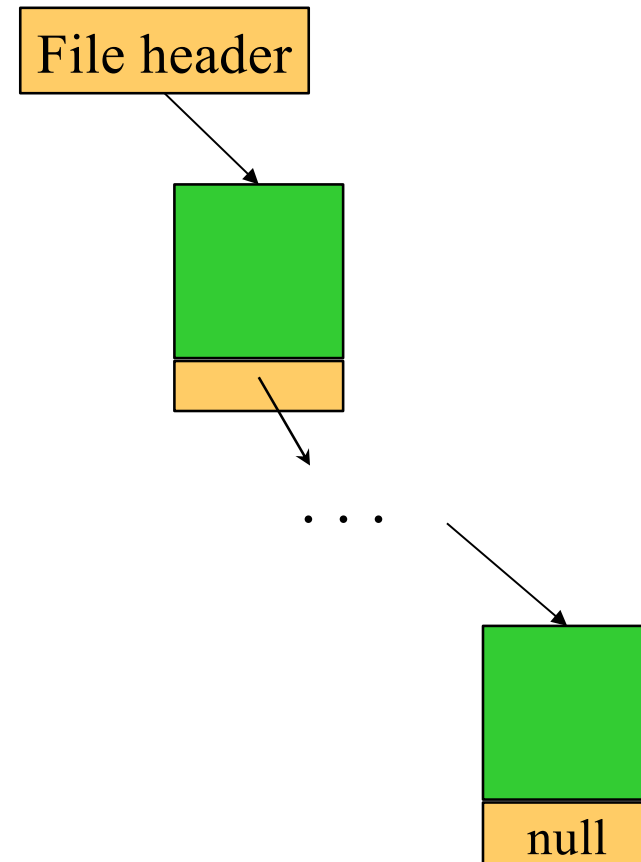
# Files: Contiguous Allocation

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

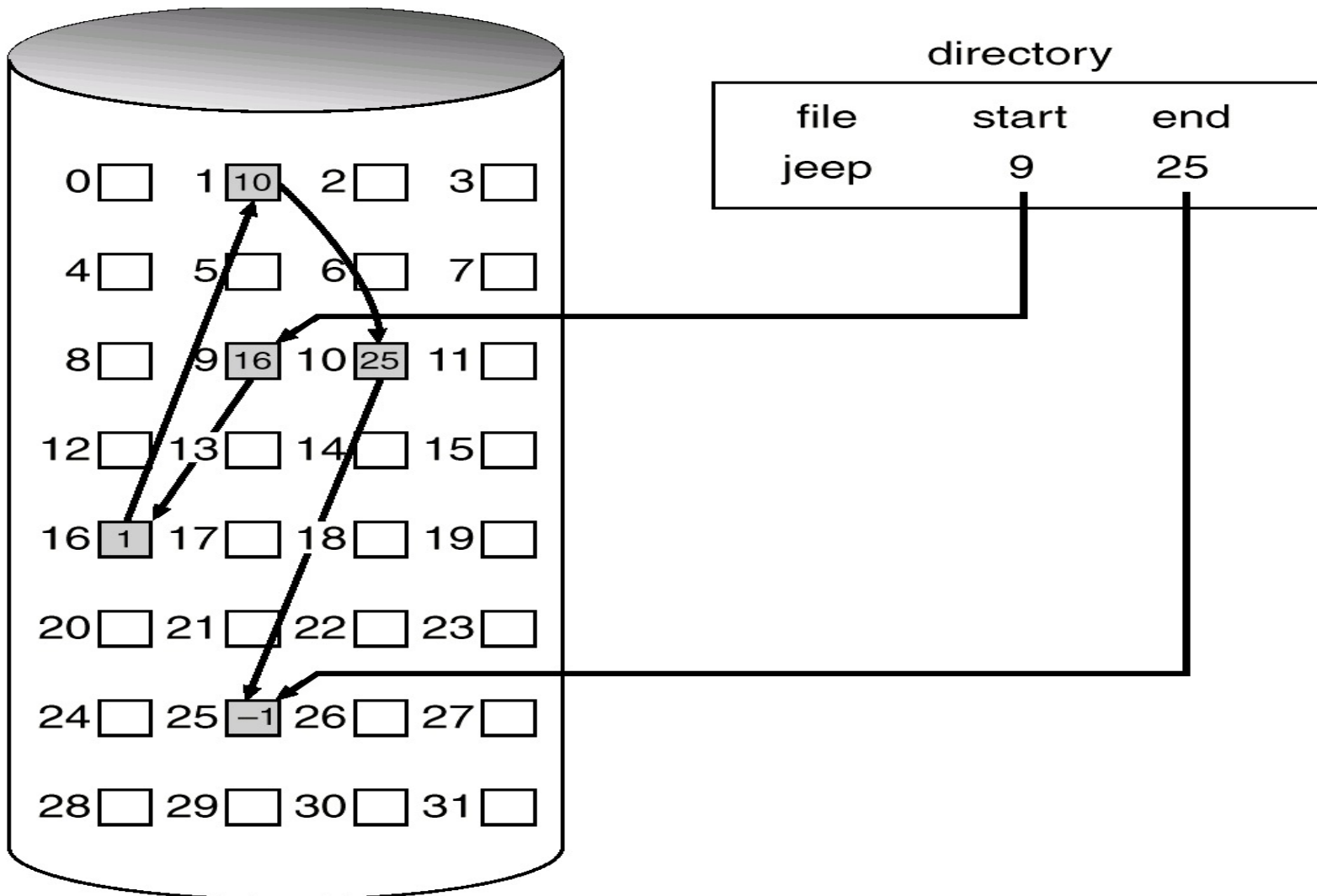


# Linked Files

- ◆ File structure (Alto)
  - 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
  - File data tracked similarly to free list of blocks
  - Doesn't waste space
- ◆ Cons
  - Random access: bad
  - Unreliable: losing a block means losing the rest

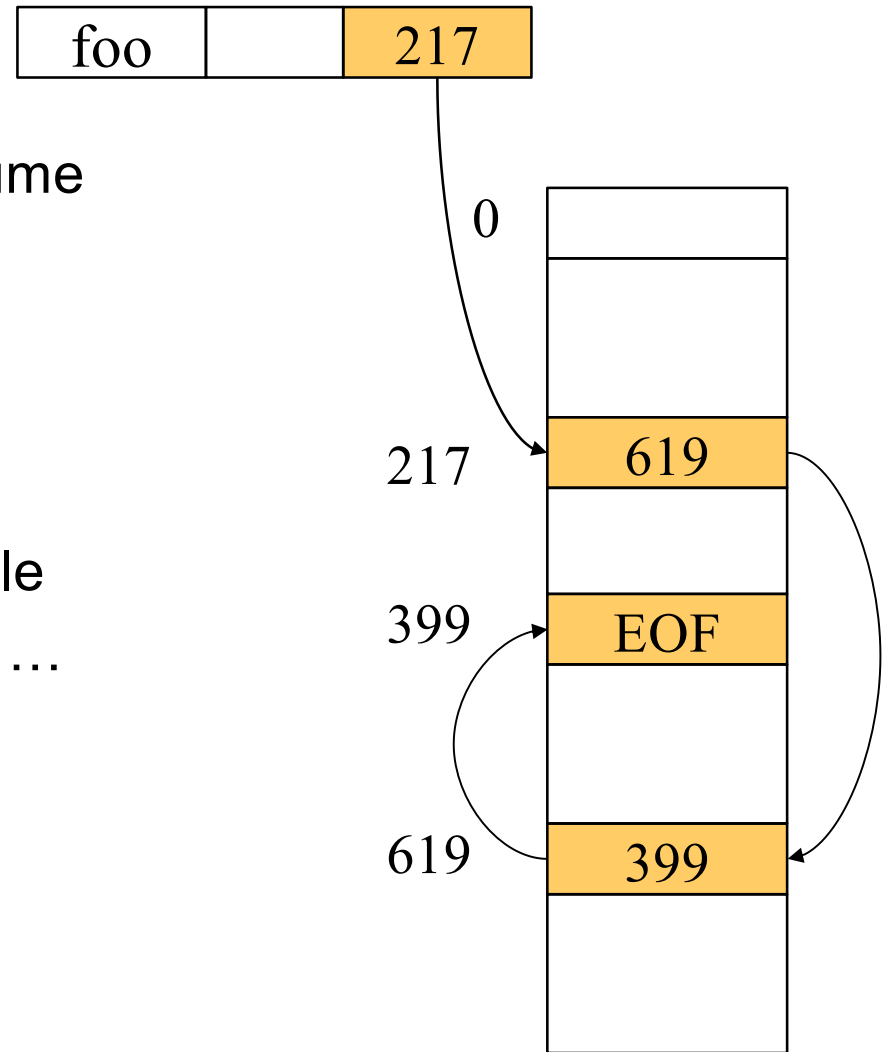


# Linked files (cont'd)



# File Allocation Table (FAT)

- Idea is to keep the linked list metadata (pointers) in memory, rather than on disk
- Allocation table at beginning of each volume
  - ◆ N entries for N blocks
  - ◆ Want to keep it in memory
- File structure (MS-DOS)
  - 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



# DEMOS (Cray-1)

## ◆ Idea

- Try contiguous allocation
- Allow non-contiguous

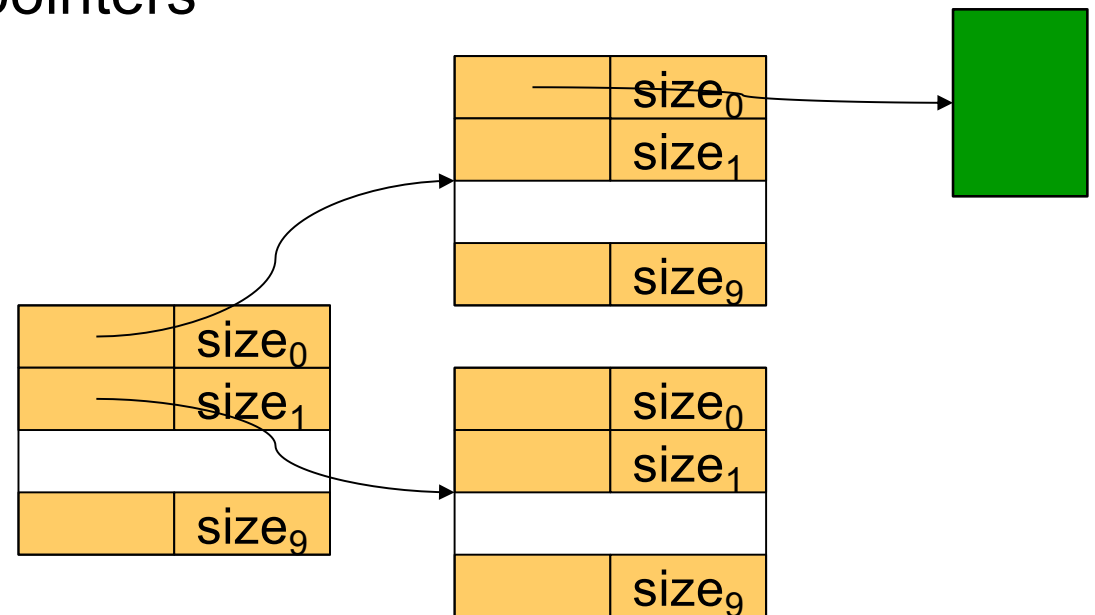
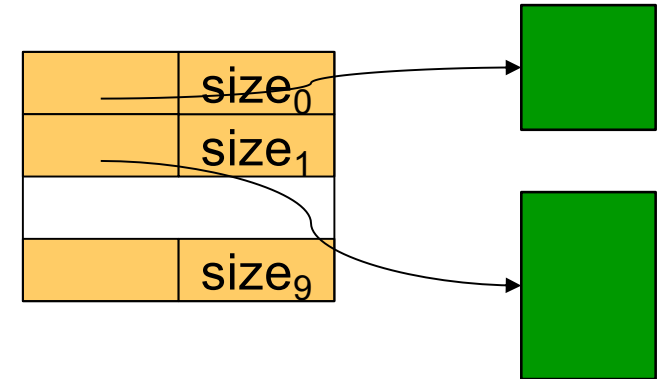
## ◆ File structure

- Small file metadata has 10 (base,size) pointers
- Big file has 10 indirect pointers

## ◆ Pros & Cons

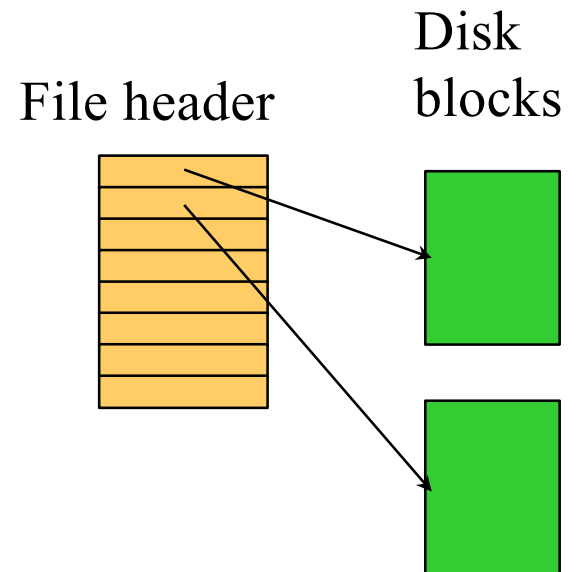
- Can grow
- Fragmentation

File metadata

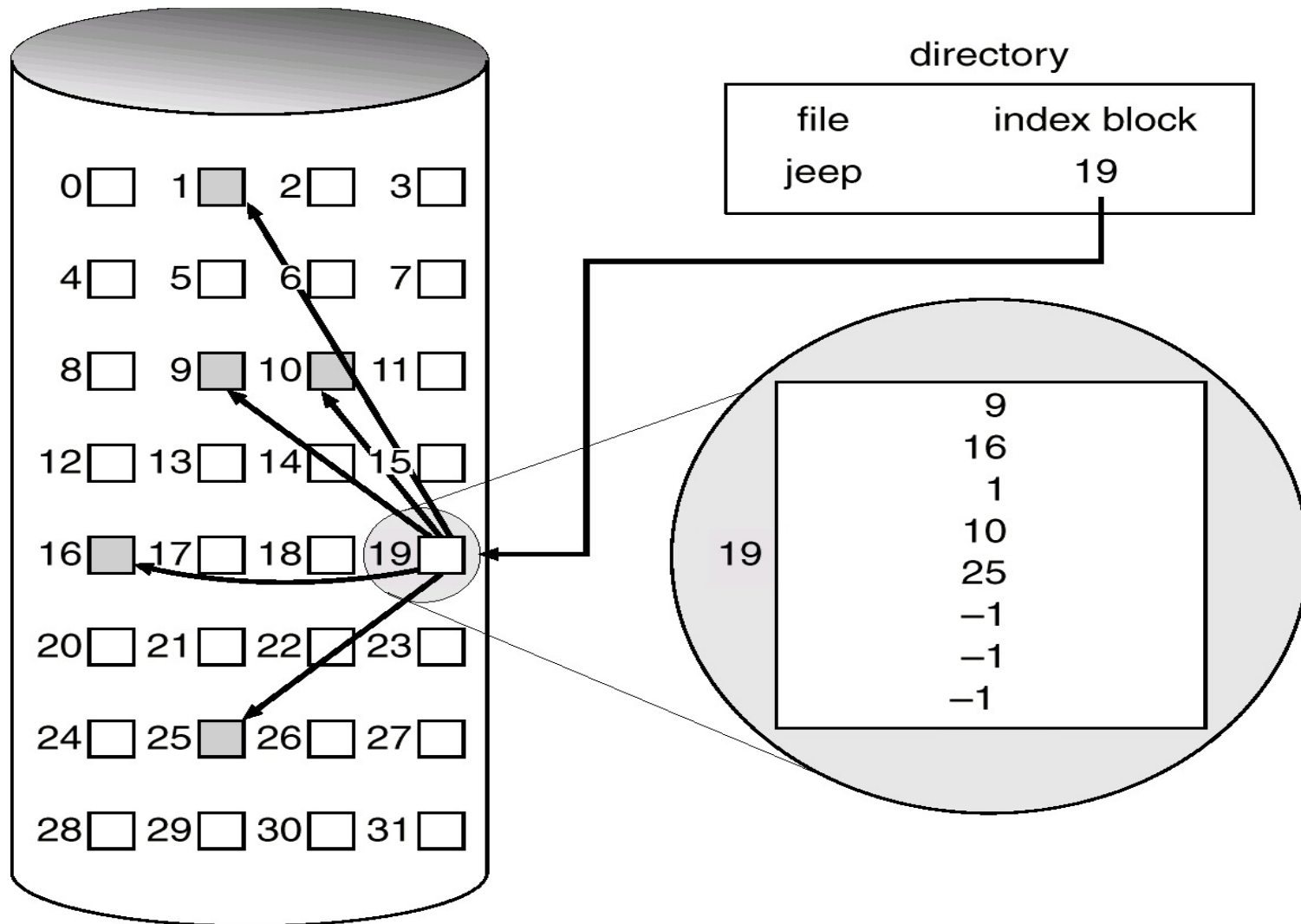


# Single-level Indexed File

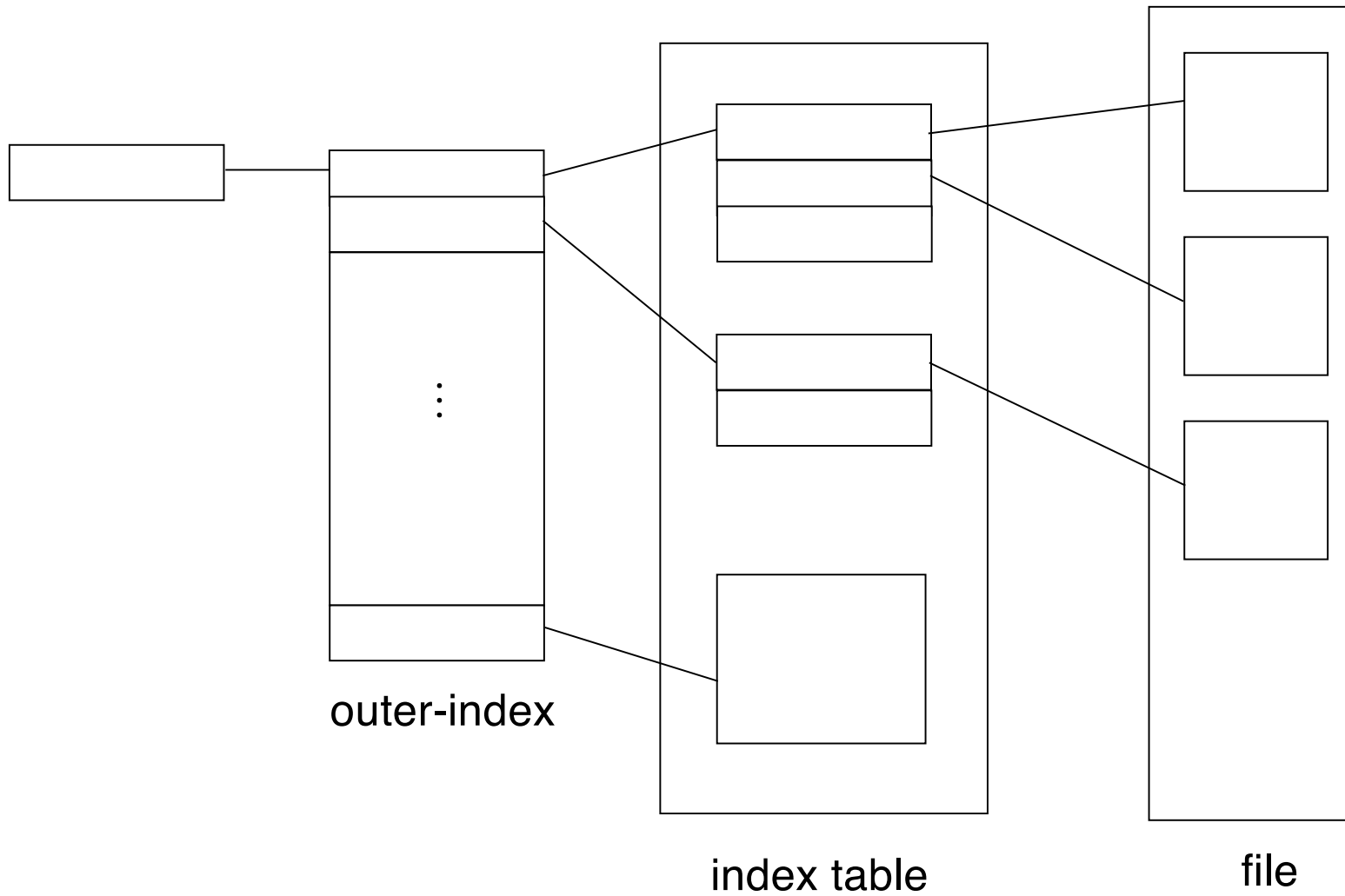
- User declares max size
- File header holds array of pointers to disk blocks
- Pros:
  - Can grow up to a limit
  - Random access is fast
  - No external fragmentation
- Cons:
  - Clumsy to grow beyond limit
  - Still lots of seeks



# Single-level indexed files (cont'd)



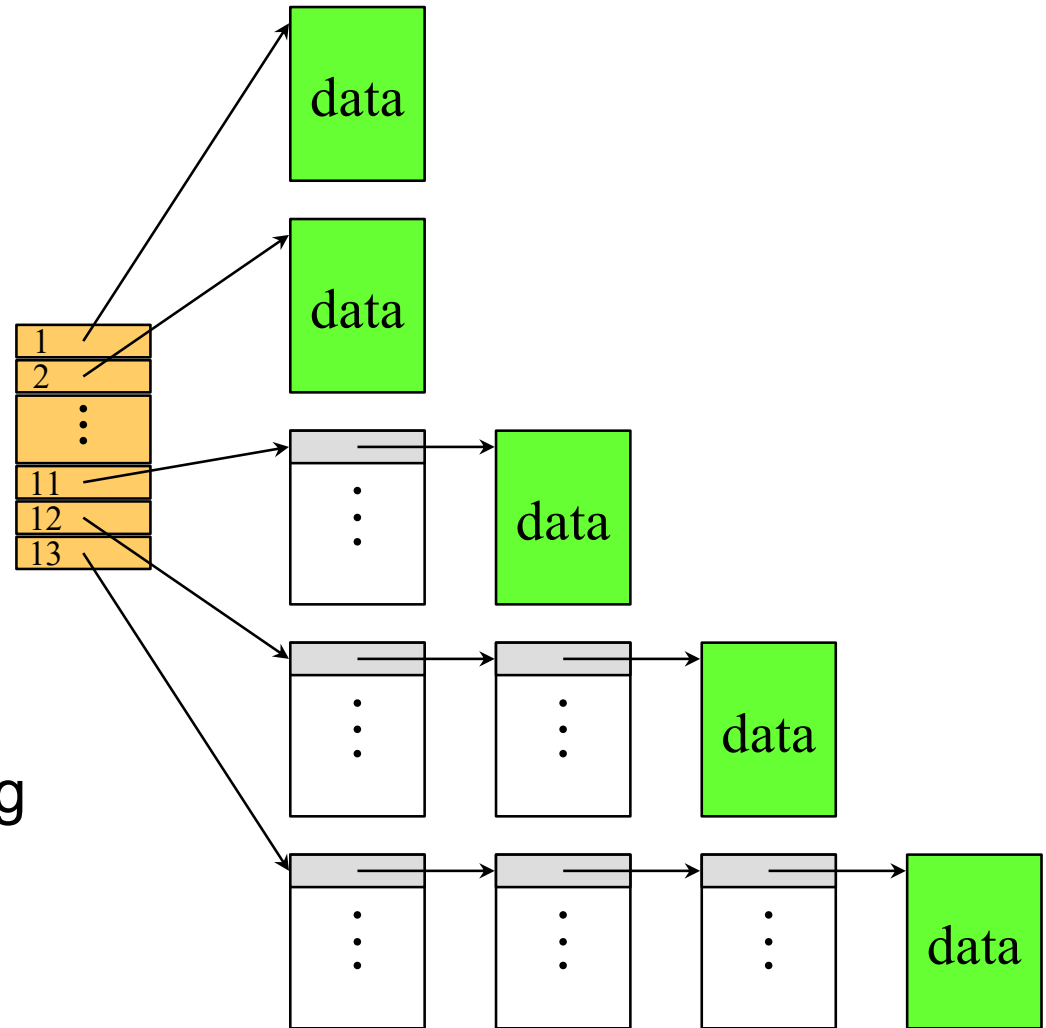
# Multi-level Indexed Files





# Hybrid 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

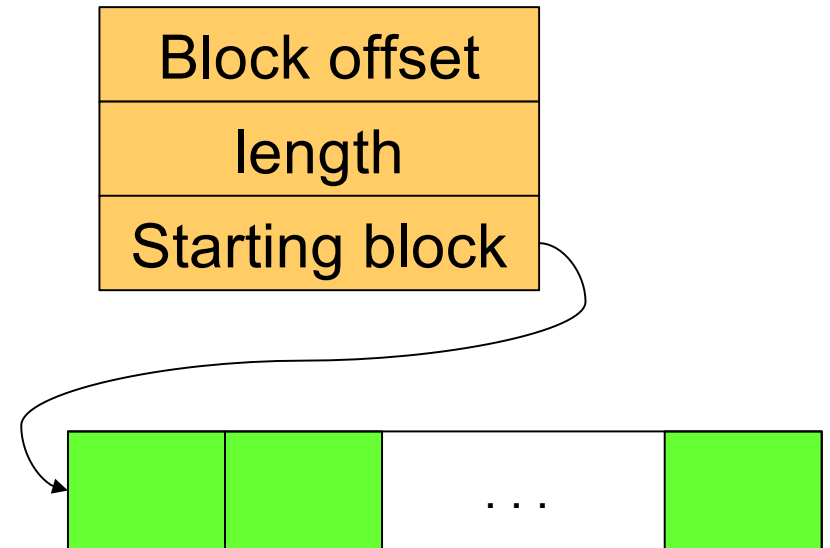
---

- ◆ Mode: file type, protection bits, setuid, setgid bits
- ◆ Link count: no. 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: little metadata, fast seq access, can grow over time, less fragmentation
- ◆ Cons: external fragmentation still problem



# Naming Files

---

Can name files via:

- ◆ Index (i-node number): Not easy for users to specify
- ◆ Text name: Need to map it to index
- ◆ Icon: Need to map it to index or to text and then to index
- ◆ Directories
  - ◆ Table of file name, file index pairs
  - ◆ Map name to file index (where to find the header)
  - ◆ A directory is itself stored as a file



# Naming Tricks

---

- Bootstrapping: Where do you start looking?
  - Root directory
  - inode #2 on the system
  - 0 and 1 used for other purposes
- Special names:
  - Root directory: “/” (bootstrap name system for users)
  - Current directory: “.”
  - Parent directory: “..”
  - user’s home directory: “~”
- Using the given names, only need two operations to navigate the entire name space:
  - cd ‘name’: move into (change context to) directory “name”
  - ls : enumerate all names in current directory (context)



# 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
- ◆ 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> ... <very  
LongFileName,  
4567>
```





# Tree Data Structure

## ◆ Method

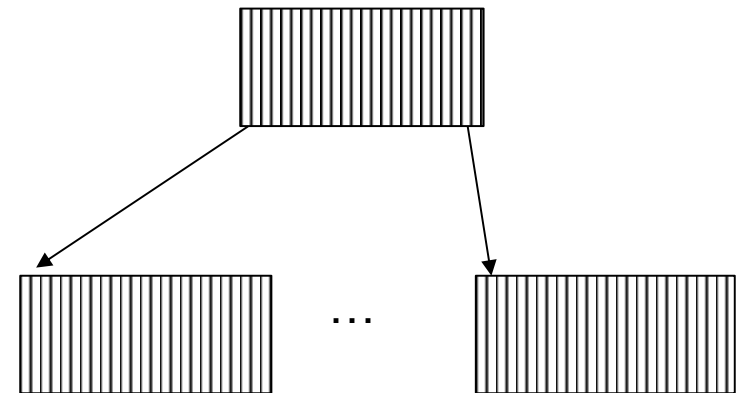
- Store <fileName, i-node> a tree data structure such as B-tree
- Create/delete/search in the tree data structure

## ◆ Pros

- Good for a large number of files

## ◆ Cons

- Inefficient for a small number of files
- More space
- Complex



# Hashing

## ◆ Method

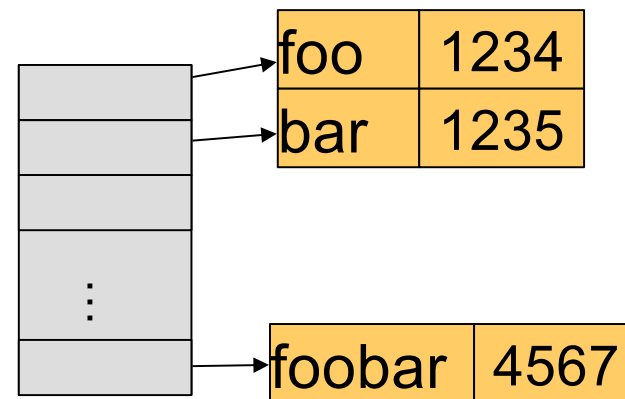
- Use a hash table to map FileName to i-node
- Space for name and metadata is variable sized
- Create/delete will trigger space allocation and free

## ◆ 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*** (e.g. `./foo`)

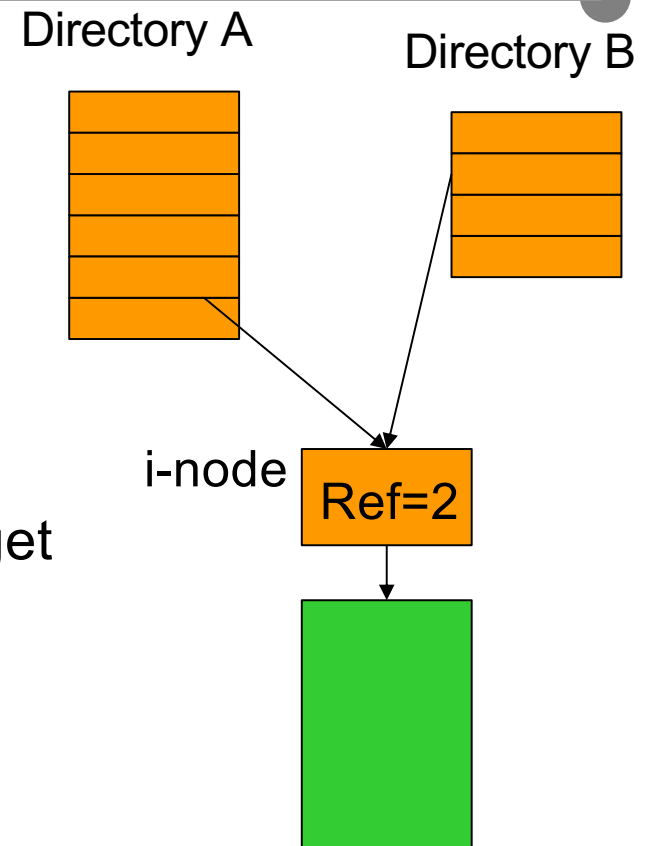


# Hard Links

## ◆ Approach

- A link to a file with the same i-node  
In source target
- i.e. the name points to the same i-node  
as that of the file being linked to
- Delete may or may not remove the target  
depending on whether it is the last one  
(link reference count)

## ◆ 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`
- Carries pathname of original file

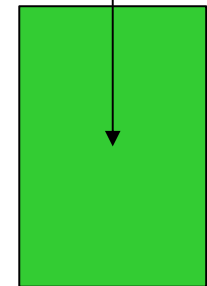
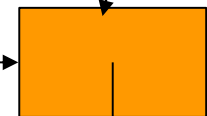
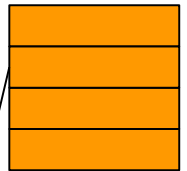
## ◆ Main issue with symbolic links?

- ◆ Performance?
- ◆ What if you delete the link?
- ◆ What if you delete the original file?

Directory B



Directory A



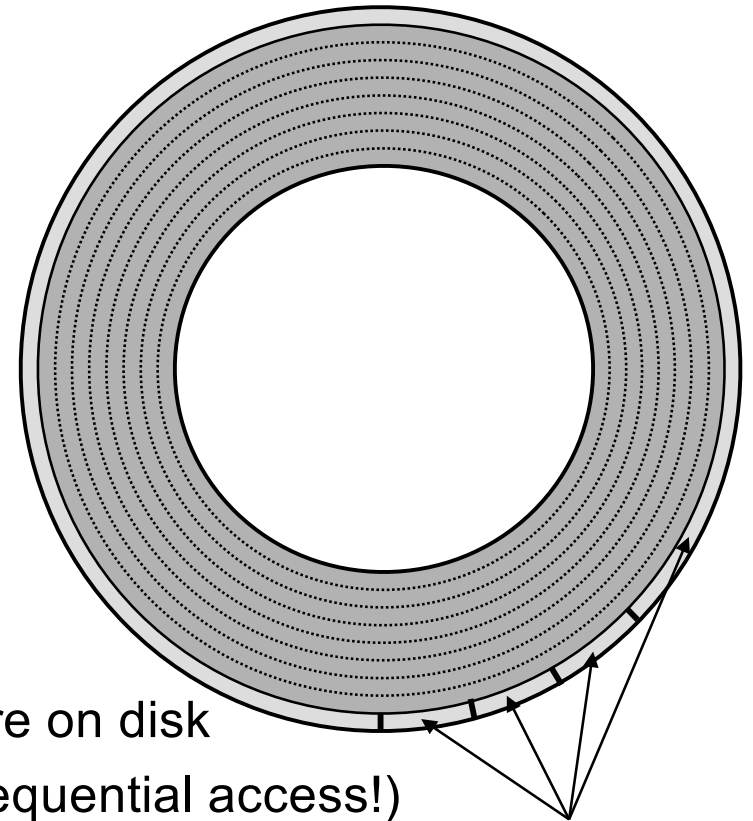
# Original Unix File System Disk Layout

## ◆ 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

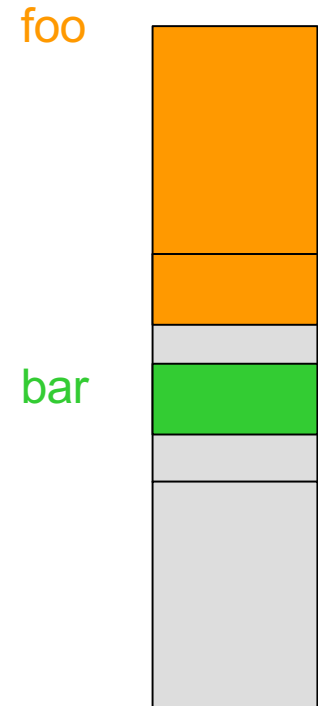
- Index is large due to small block size
- Fixed max number of files
- i-nodes far from data blocks
- i-nodes for directory not close together
- Consecutive blocks of file can be anywhere on disk
- Poor bandwidth (20Kbytes/sec even for sequential access!)



**i-node array**

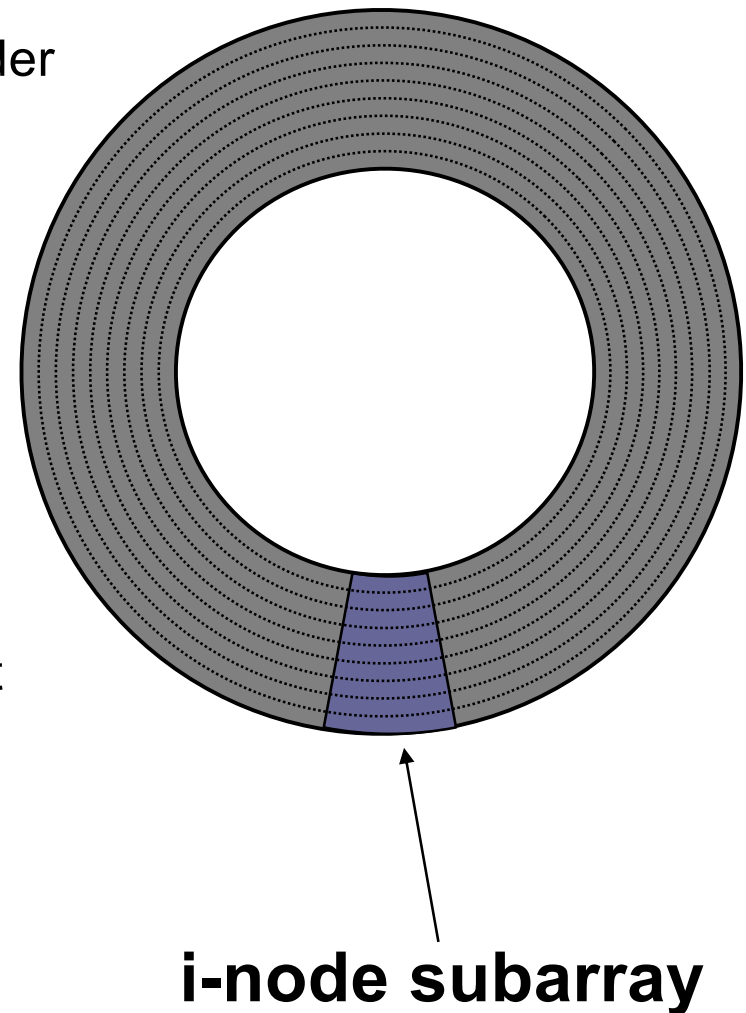
# BSD FFS (Fast File System)

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



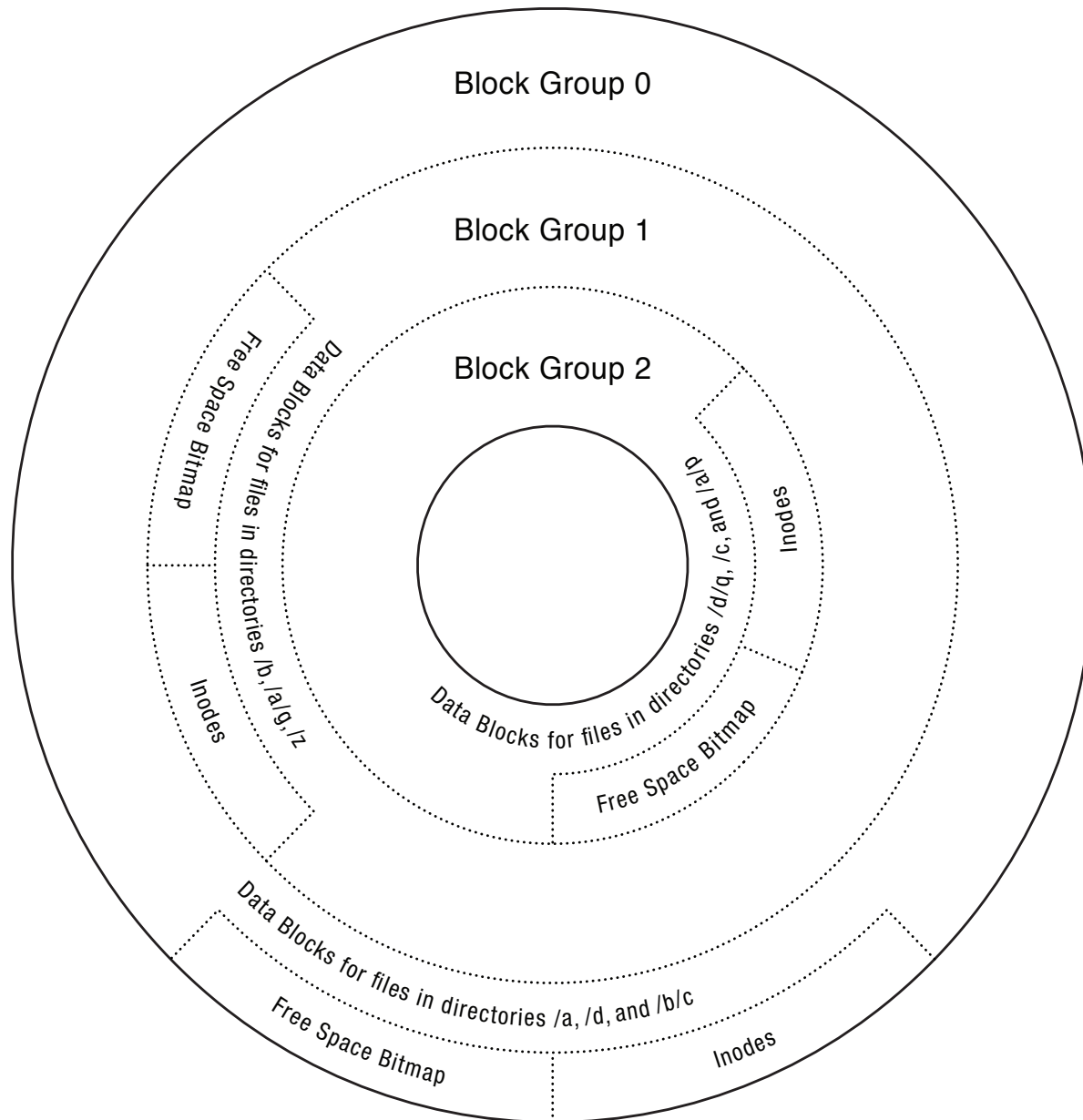
# FFS Disk Layout

- ◆ 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





# FFS block groups for better locality



# 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



# 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

