COS 318: Operating Systems

File Layout and Directories

Andy Bavier
Computer Science Department
Princeton University

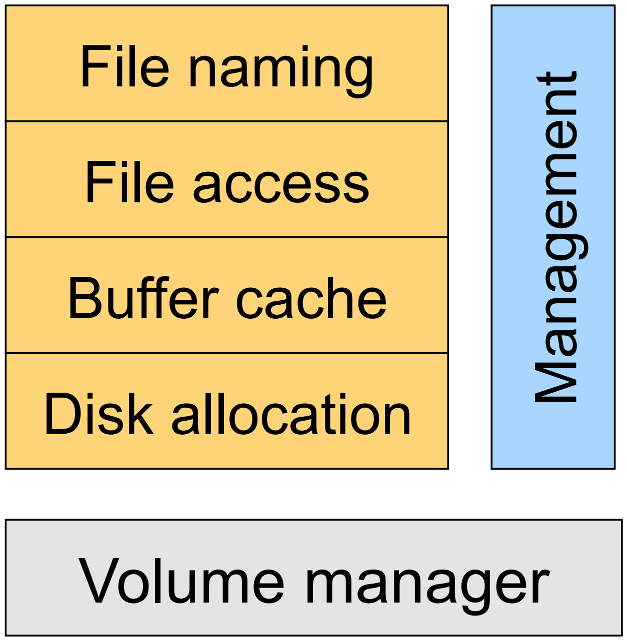
http://www.cs.princeton.edu/courses/archive/fall10/cos318/

Topics

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

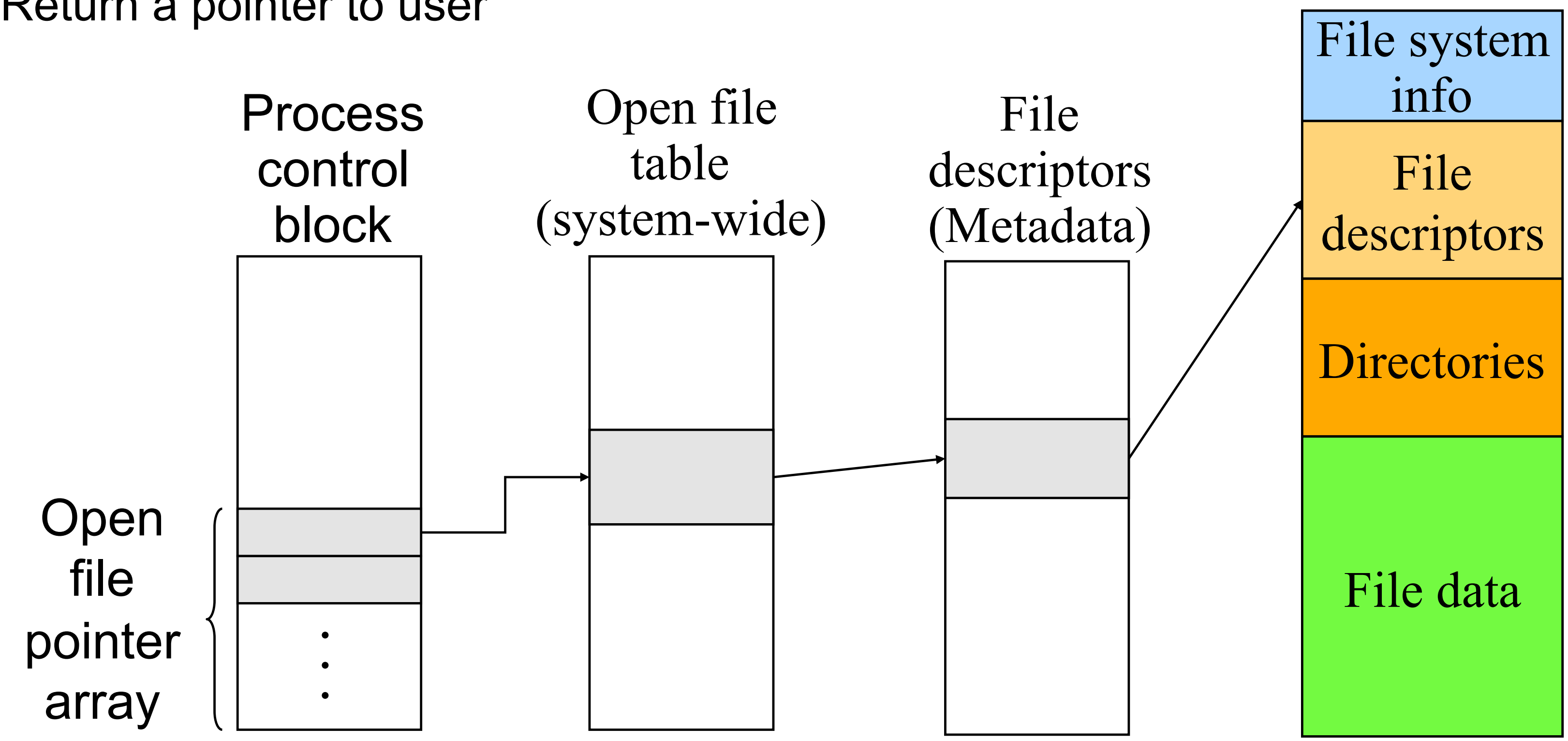
File System Components

- Naming
  - File and directory naming
  - Local and remote operations
- File access
  - Implement read/write and other functionalities
- Buffer cache
  - Reduce client/server disk I/Os
- Disk allocation
  - File data layout
  - Mapping files to disk blocks
- Management
  - Tools for system administrators to manage file systems



Steps to Open A File

- File name lookup and authenticate
- Copy the file descriptors into the in-memory data structure, if it is not in yet
- Create an entry in the open file table (system wide) if there isn't one
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



File Read and Write

- Read 10 bytes from a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - read 10 bytes
- Write 10 bytes to a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - write 10 bytes in memory
  - write out the block

Disk Layout

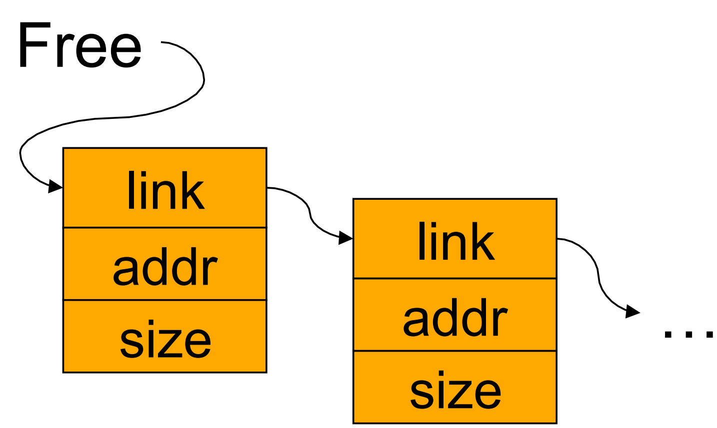
- Boot block
  - Code to bootstrap the operating system
- Super-block defines a file system
  - Size of the file system
  - Size of the file descriptor area
  - Free list pointer, or pointer to bitmap
  - Location of the file descriptor of the root directory
  - Other meta-data such as permission and various times
- File descriptors
  - Each describes a file
- File data blocks
  - Data for the files, the largest portion on disk
- Where should we put the boot image?

Boot block	Super block	File descriptors (i-node in Unix)	File data blocks

Data Structures for Disk Allocation

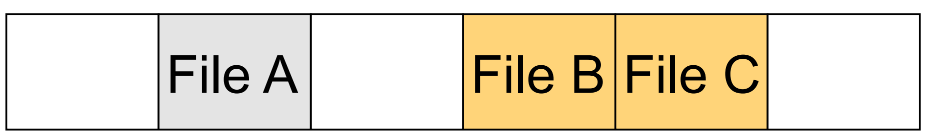
- The goal is to manage the allocation of a volume
- A file header for each file
  - Disk blocks associated with each file
- A data structure to represent free space on disk
  - Bit map that uses 1 bit per block (sector)
  - Linked list that chains free blocks together
  - Buddy system
  - …

11111111111111111000000000000000
00000111111110000000000111111111
⋮
11000001111000111100000000000000



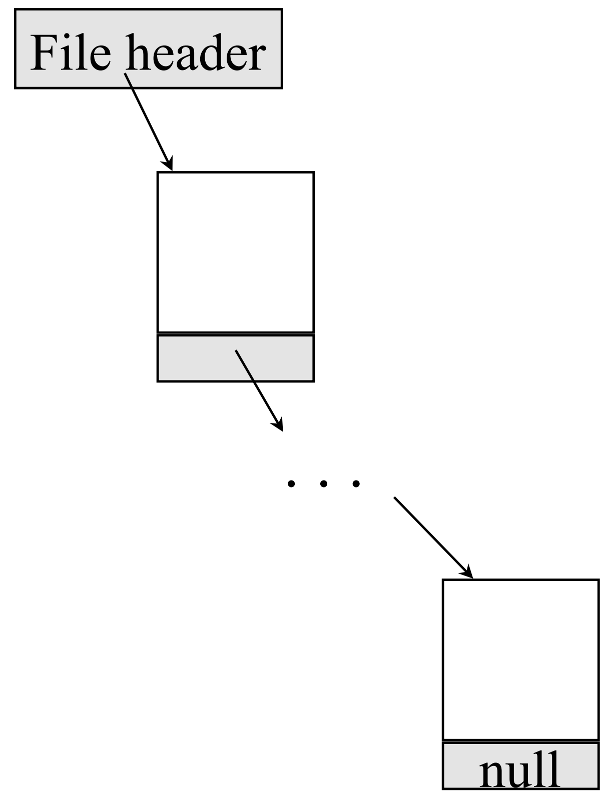
Contiguous Allocation

- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header
  - First block in file
  - Number of blocks
- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation (what if file C needs 3 blocks)
  - Hard to grow files



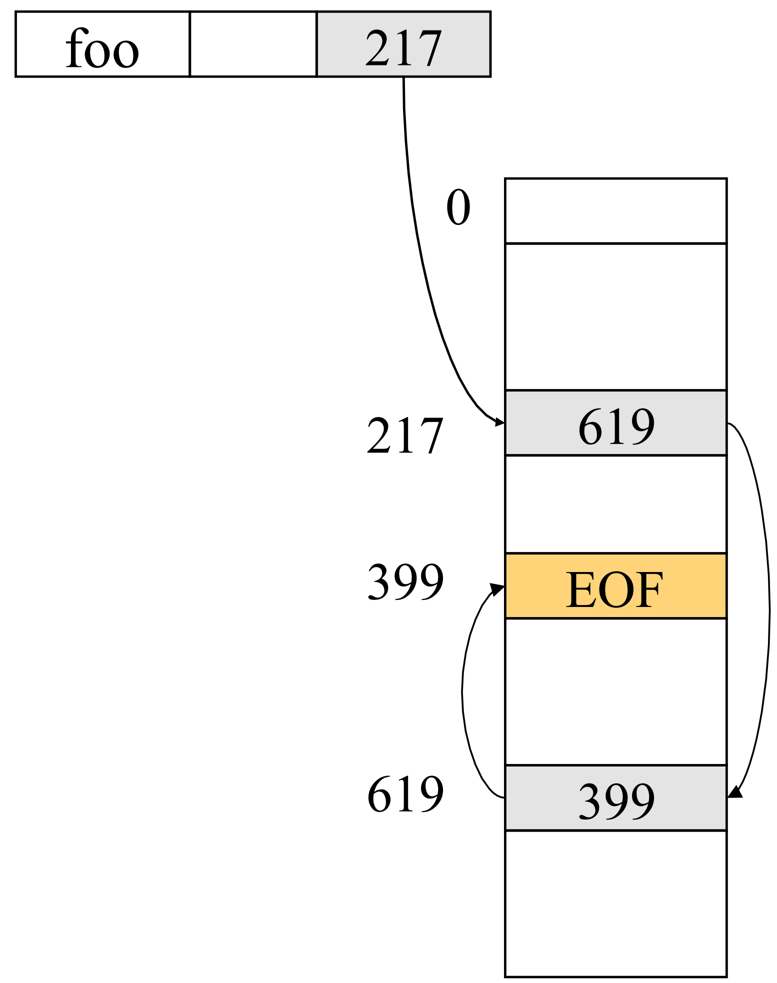
Linked Files (Alto)

- File header points to 1st block on disk
- A block points to the next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
- Cons
  - Random access: horrible
  - Unreliable: losing a block means losing the rest



File Allocation Table (FAT)

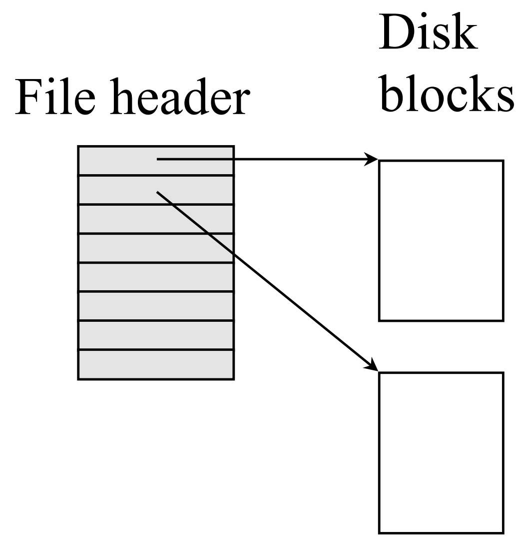
- Approach
  - A section of disk for each partition is reserved
  - One entry for each block
  - A file is a linked list of blocks
  - A directory entry points to the 1st block of the file
- Pros
  - Simple
- Cons
  - Always go to FAT
  - Wasting space



FAT Allocation Table

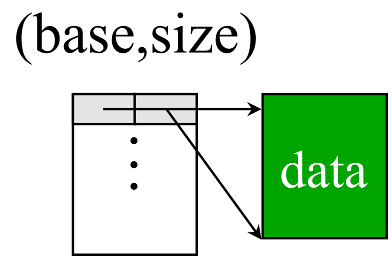
Single-Level Indexed Files

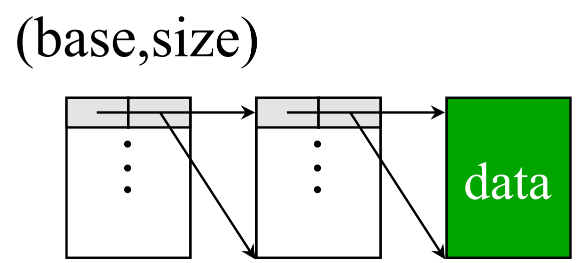
- A user declares max size
- A file header holds an array of pointers to point to disk blocks
- Pros
  - Can grow up to a limit
  - Random access is fast
- Cons
  - Clumsy to grow beyond the limit
  - Still lots of seeks



DEMOS (Cray-1)

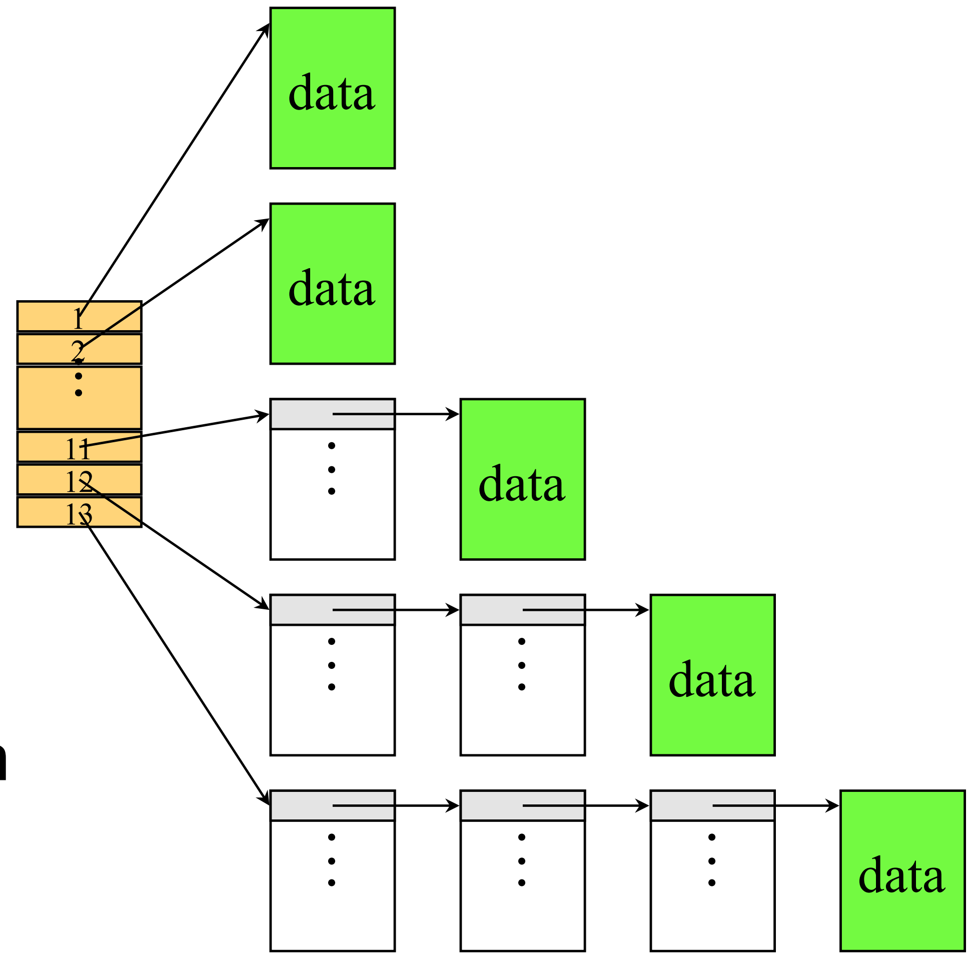
- Idea
  - Using contiguous allocation
  - Allow non-contiguous
- Approach
  - 10 (base,size) pointers
  - Indirect for big files
- Pros & cons
  - Can grow (max 10GB)
  - fragmentation
  - find free blocks





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 and lots of seek
- What happens to reach block 23, 5, 340?



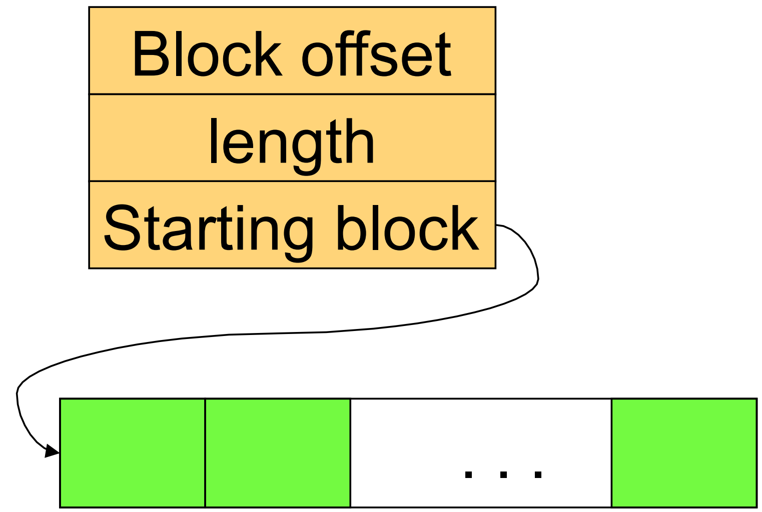
What’s in Original Unix i-node?

- Mode: file type, protection bits, setuid, setgid bits
- Link count: number of directory entries pointing to this
- 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

- Instead of using a fix-sized block, use a number of blocks
  - XFS uses 8Kbyte block
  - Max extent size is 2M blocks
- Index nodes need to have
  - Block offset
  - Length
  - Starting block
- Is this approach better than the Unix i-node approach?



Naming

- 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
  - Use the extension to indicate whether the entry is a directory

Mapping File Names to i-nodes

- 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

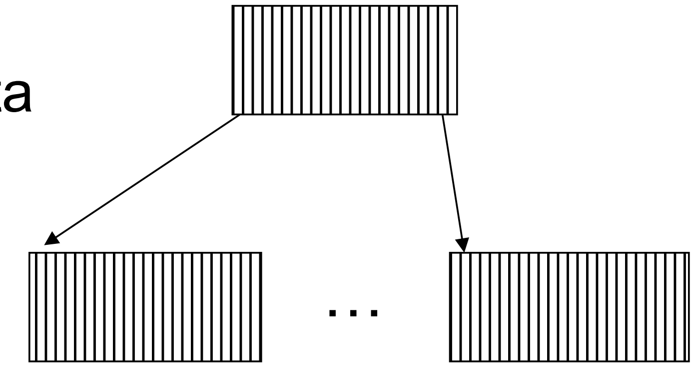
- Method
  - <FileName, i-node> pairs are linearly stored in a file
  - Create a file
    - Append <FileName, i-node>
  - Delete a file
    - Search for FileName
    - Remove its pair from the directory
    - Compact by moving the rest
- Pros
  - Space efficient
- Cons
  - Linear search
  - Need to deal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

/u/li/
foo bar …
veryLongFileName

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

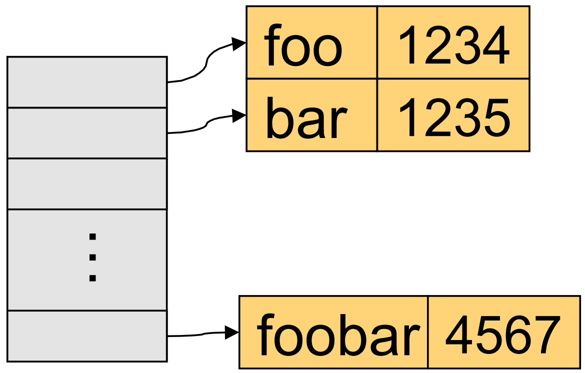
Tree Data Structure

- 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)



Disk I/Os for Read/Write A File

- Disk 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"
- Disk I/Os to write a file
  - Read the i-node of the directory and the directory file.
  - 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***

Links

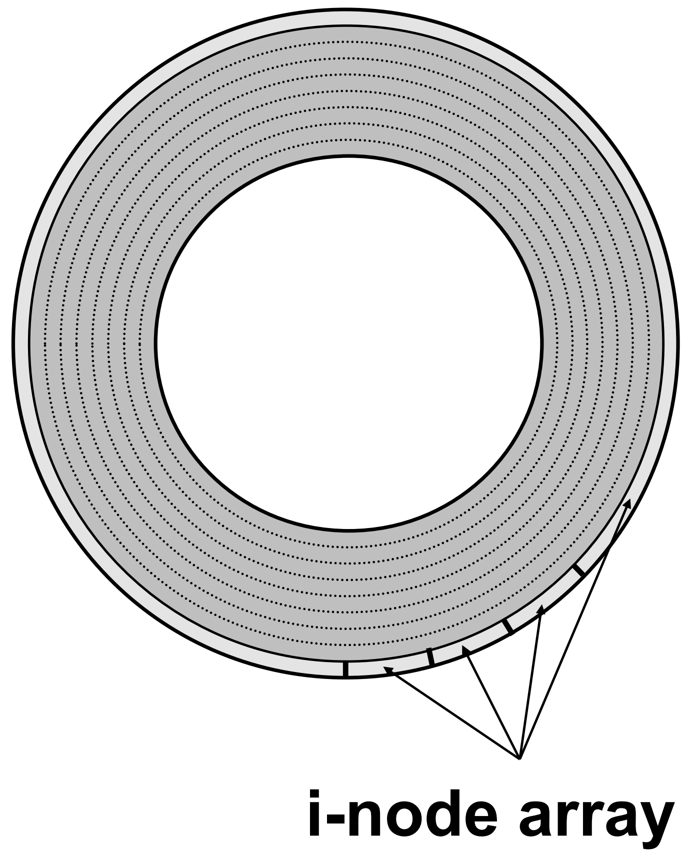
- Symbolic (soft) links
  - A symbolic link is a pointer to a file
  - Use a new i-node for the link

    ```
    ln -s source target
    ```
- Hard links
  - 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)
- Why symbolic or hard links?
- How would you implement them?

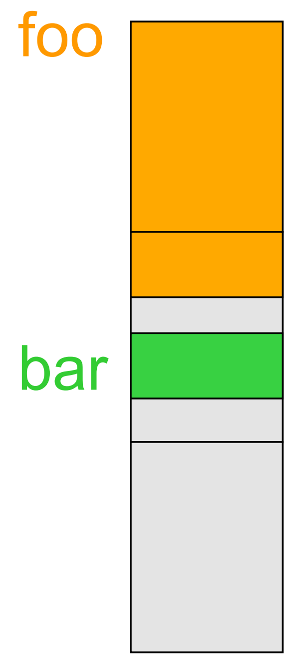
Original Unix File System

- 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!)



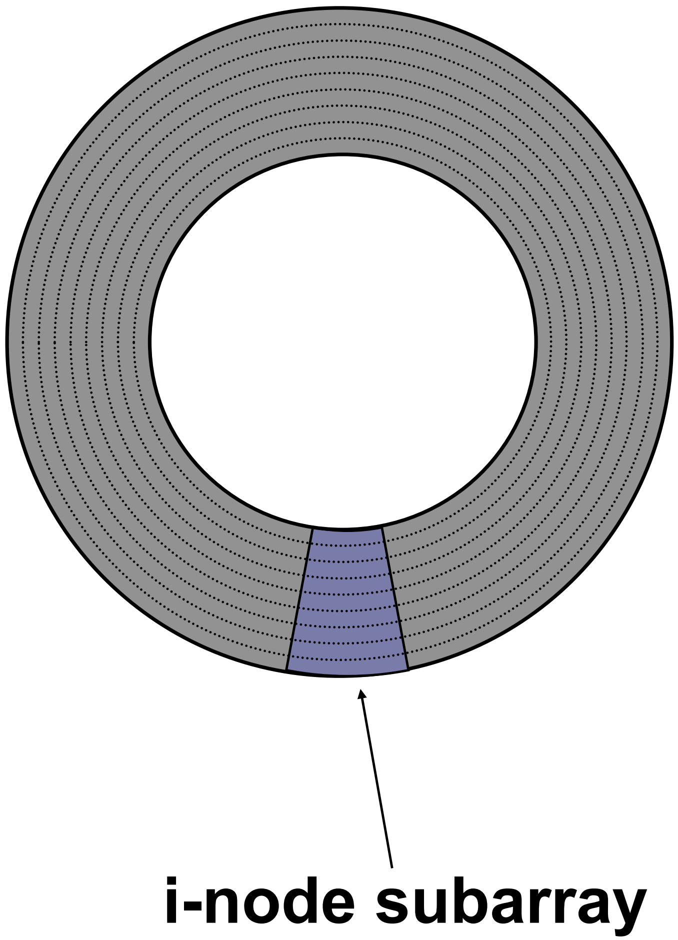
BSD FFS (Fast File System)

- 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
  - 10% reserved disk space



FFS Disk Layout

- i-nodes are grouped together
  - A portion of the i-node array on each cylinder
- 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 still do a lot 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