Operating Systems

Disk & File System Implementation

Hongliang Liang, BUPT

Sep. 2023

Objective: Virtualizing Persistent Storage

The **file system** provides the mechanism for on-line storage and access to file contents, and resides permanently on **nonvolatile secondary storage**.

Hard Disk Drives

Organized as an array of **sectors**, each 512 bytes

- Address space: the sector number, from 0 to n - 1
- Writing a single sector is guaranteed to be **atomic**

File systems usually combine multiple sectors into a single **block** — say, 4KB in size

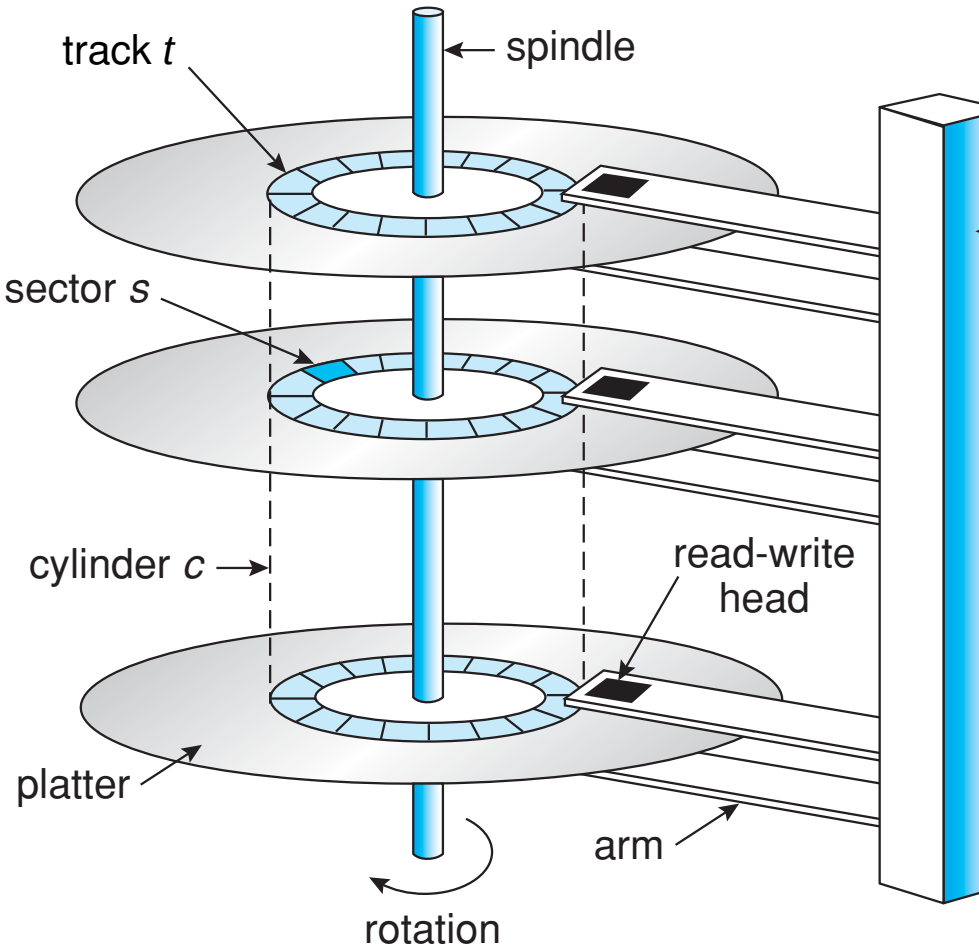
Hard Disk Drives

Disks have multiple platters

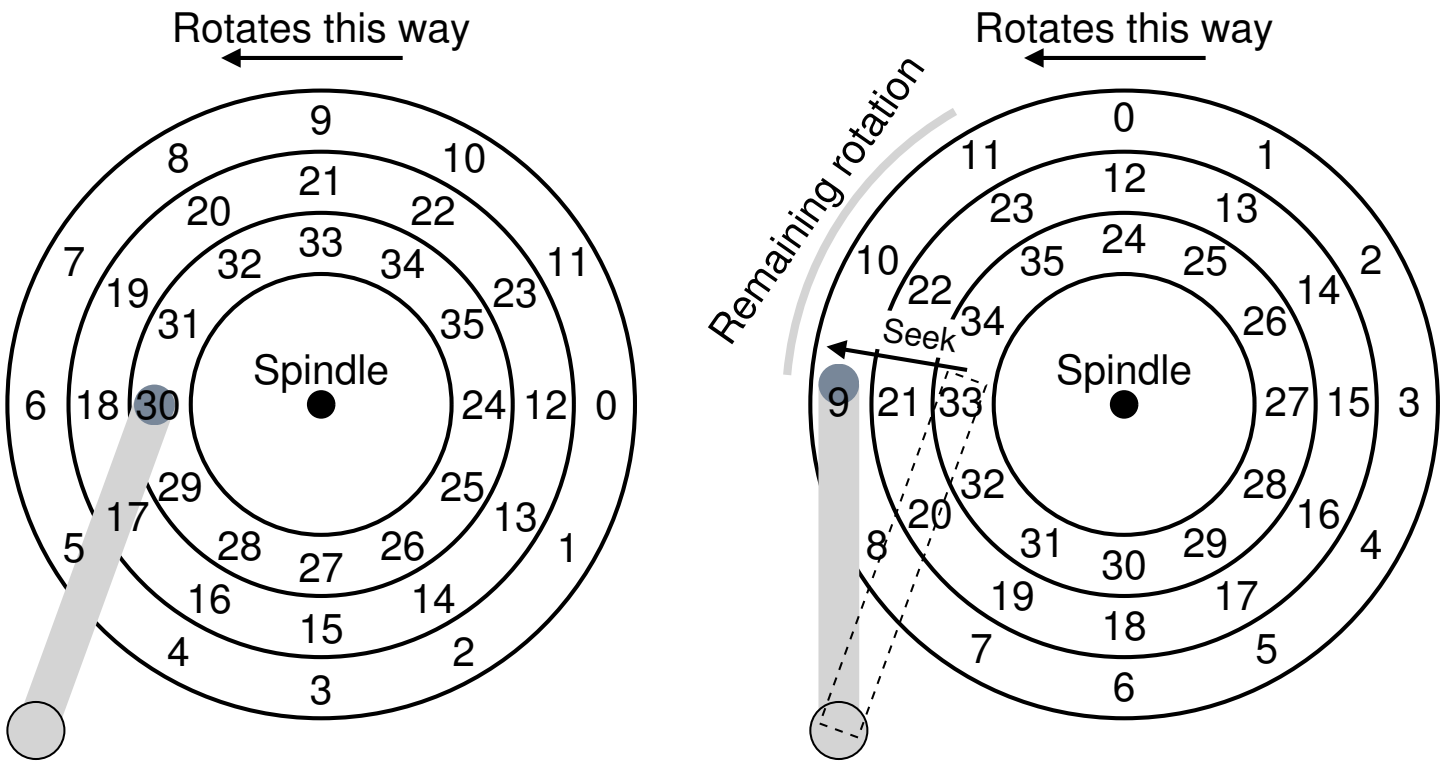
- Each platter has an arm and a head
- Different heads can access data in parallel

Each platter has multiple concentric tracks

- Same set of tracks across all platters is a cylinder
- Each track has multiple sectors



Rotational delay and seek time



a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer.

Disk Scheduling

OS is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth

Minimize seek time
Seek time $\approx$ seek distance

Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

```
* #tran_bytes / (completion_time - request_time)
```

Disk Scheduling (cont.)

Whenever a process needs I/O to or from the disk, it issues a system call to OS.

This request includes input or output mode, disk address, memory address, number of sectors to transfer

OS maintains queue of requests, per disk or device

Idle disk can immediately work on I/O request, busy disk means requests must queue

Optimization algorithms only make sense when a queue exists

Disk Scheduling (cont.)

Note that drive controllers have small buffers and can manage a queue of I/O requests

Several algorithms exist to schedule the servicing of disk I/O requests

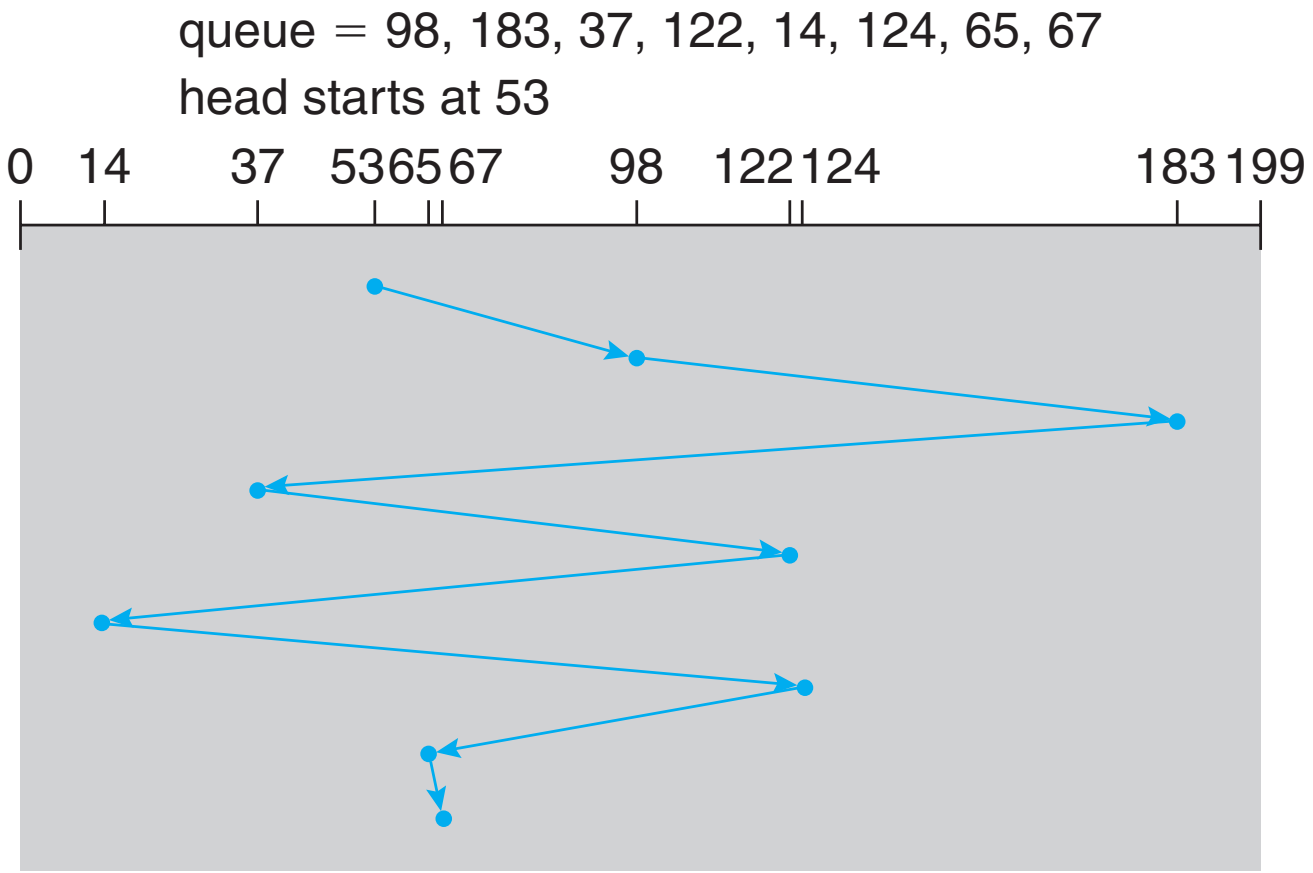
The analysis is true for one or many platters

We illustrate scheduling algorithms with a request queue (0-199)

`98, 183, 37, 122, 14, 124, 65, 67`

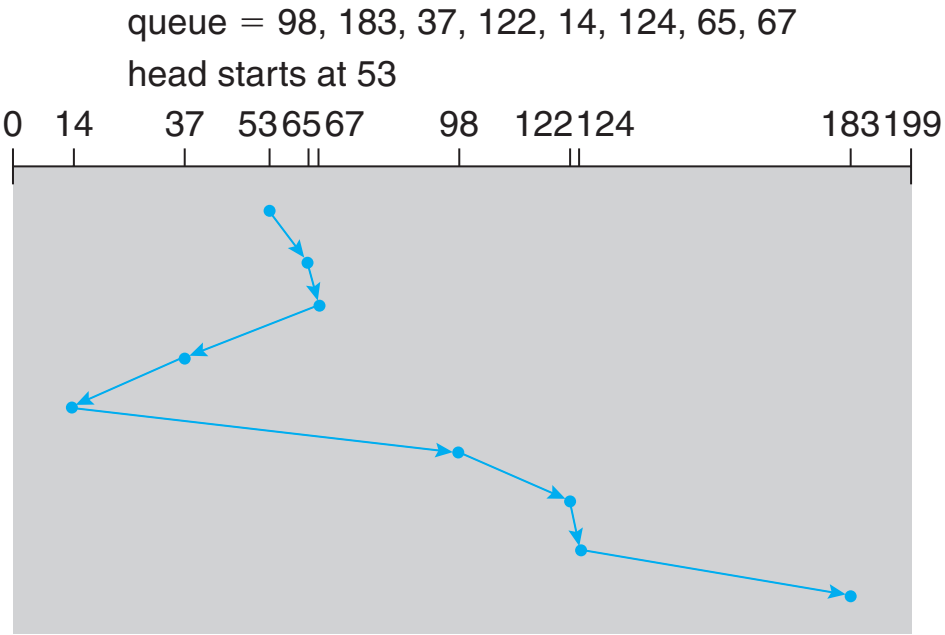
The disk head is initially at cylinder 53

First Come First Service (FCFS)



The figure shows total head movement of **640** cylinders

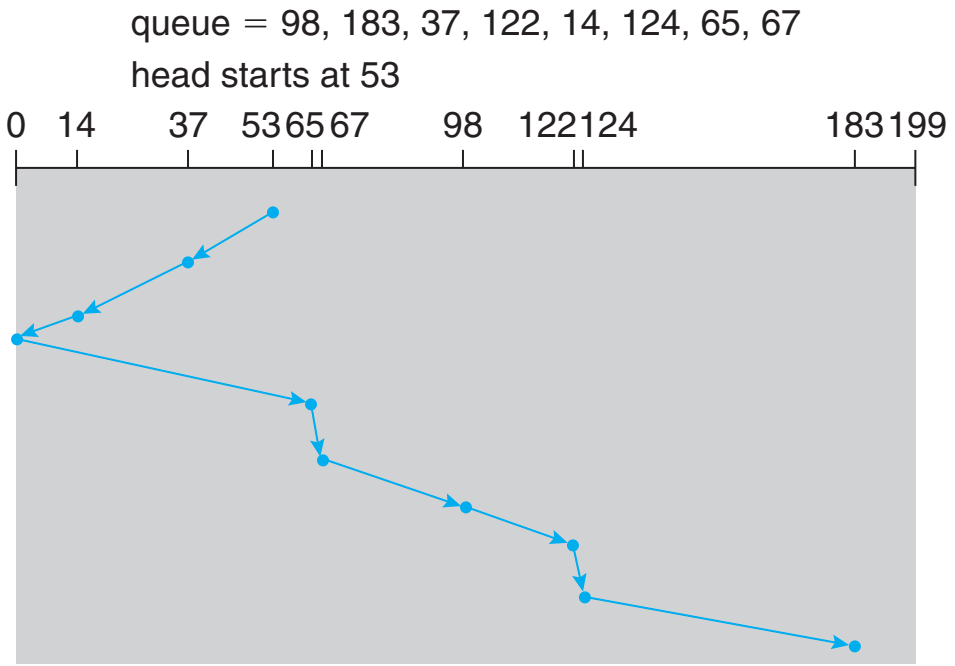
Shortest Seek Time First (SSTF)



- SSTF selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of **236** cylinders
- Not optimal. e.g., `53->37->14->...`

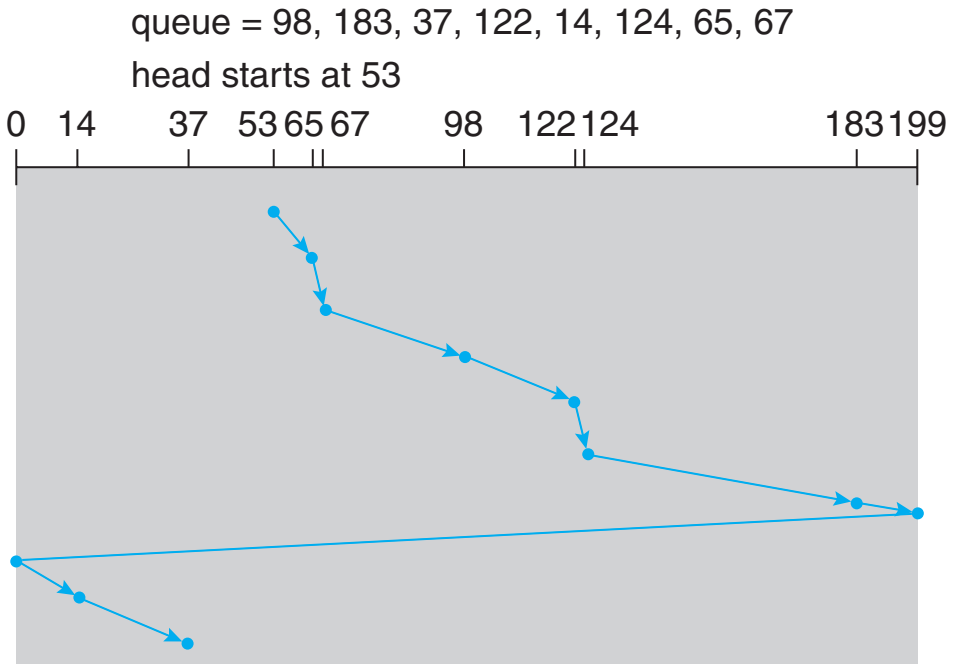
SCAN

- The disk arm starts at one end, and moves toward the other end, servicing requests until it gets to the other end, where the head movement is **reversed** and servicing continues.
- SCAN algorithm sometimes called the **elevator algorithm**
- But note that if requests are uniformly dense, the largest density of requests is at other end of disk and those wait the longest. so C-SCAN comes.



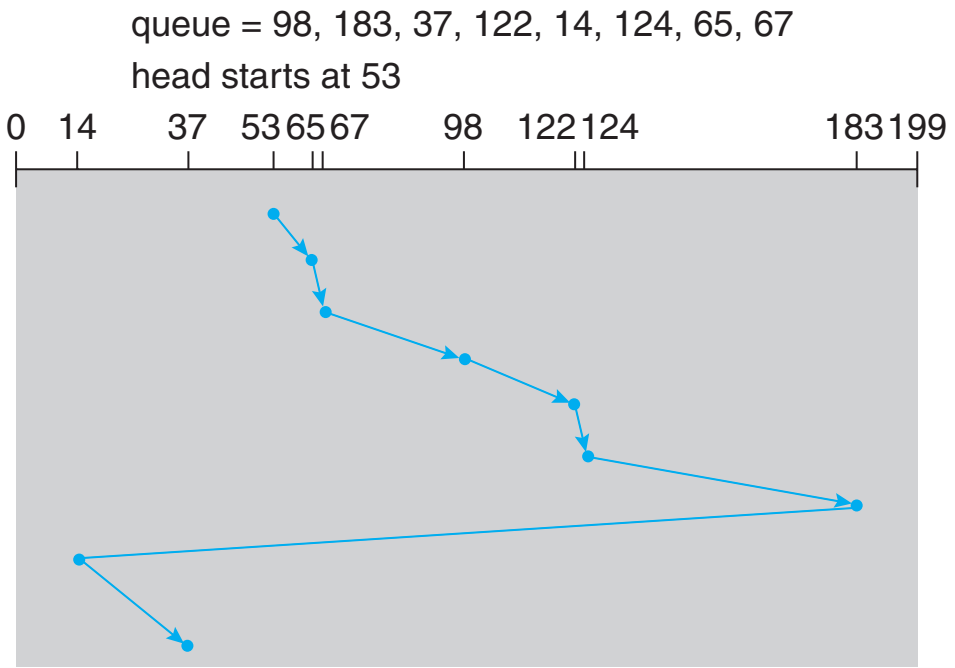
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end to the other, servicing requests as it goes
- When it reaches the other end, however, it immediately returns to the **beginning** of the disk, without servicing any requests on the **return** trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



C-LOOK

- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes **as far as the last request** in each direction, then reverses direction immediately, without first going all the way to the **end** of the disk



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a **heavy load** on the disk. Less starvation
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method and metadata layout
- The disk-scheduling algorithm should be written as a **separate module** of the operating system, allowing it to be replaced with a different algorithm if necessary
- Either **SSTF or LOOK** is a reasonable choice for the default algorithm

File Storage on Disk

Sector 0: "Master Boot Record" (MBR): contains the partition map

Rest of disk divided into "**partitions**"

- Partition: sequence of consecutive sectors

Each partition can be "**raw**" (e.g., swap partition), or "**cooked**," containing its own **file system**

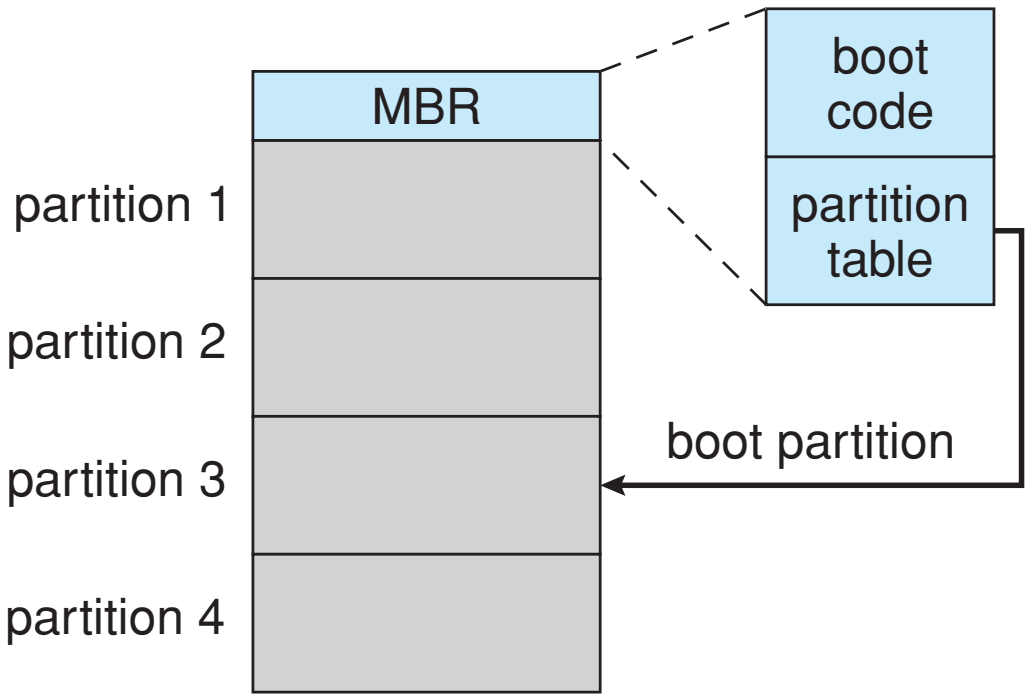
A bootable partition starts with a "boot block"

- Contains a small program, **bootloader**, which reads in an OS from the file system in that partition

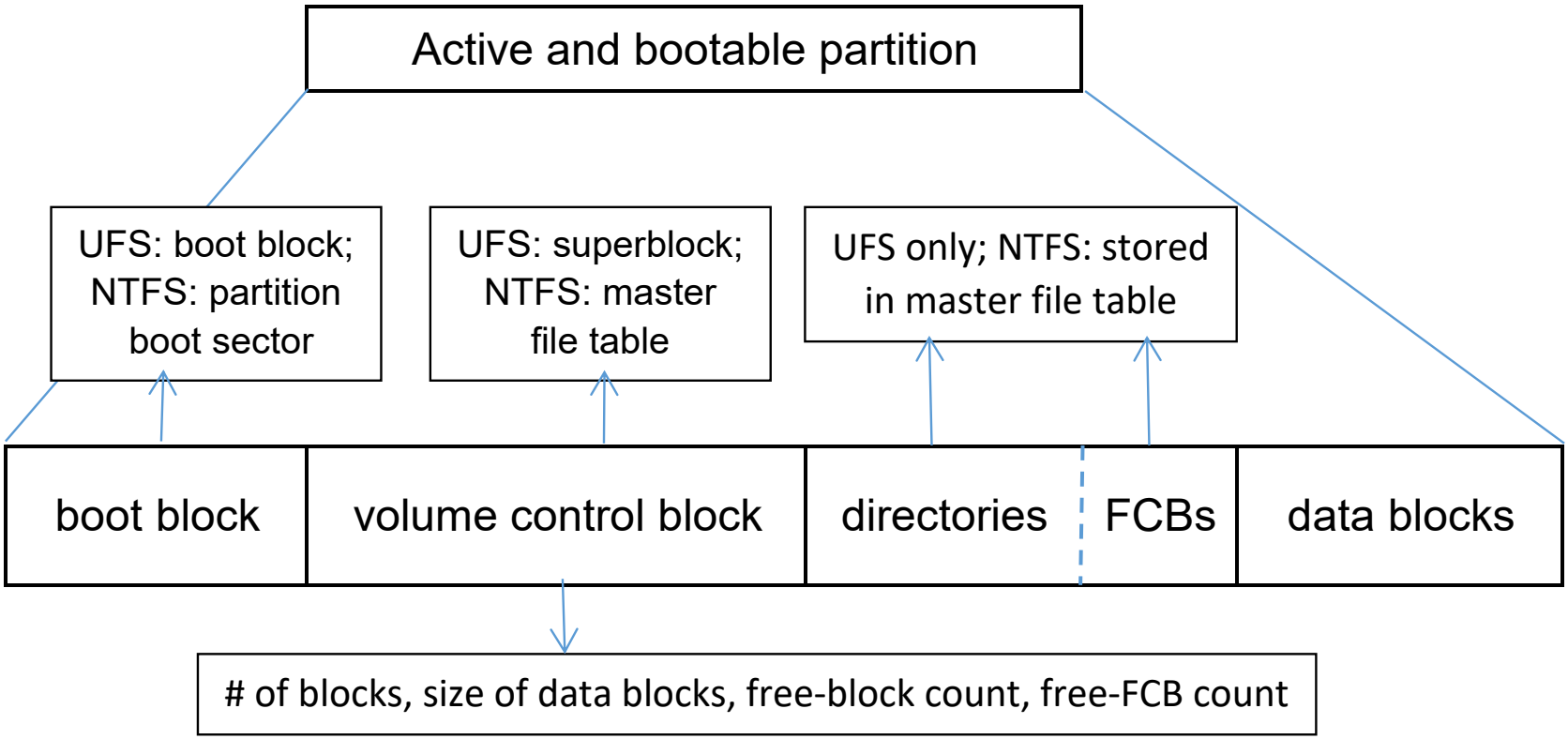
Booting the system (Optional)

- OS Boot — the legacy way
  - BIOS (Basic Input Output System) reads MBR, then reads & execs a boot block in the bootable partition
- OS Boot — the modern way
  - UEFI (Unified Extensible Firmware Interface) instead of BIOS
  - Active partition is no longer needed
  - Uses the GPT (GUID Partition Table) partitioning scheme, rather than MBR, which only works with drives up to 2TB and is no longer needed
  - Can boot from drives 2.2TB or larger
  - Developed in C, rather than assembly
  - 64-bit support and faster boot times
  - Supports secure boot

An Example Disk



A “cooked” partition with a file system (Optional)



Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system, implements vnodes which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines
- System call logic (open, seek, . . .) maps to VFS operations
- When implementing a new FS, implement the VFS API
- System calls are now independent of FS implementation

Virtual File System Implementation in Linux

- Linux VFS has four object types:
  - **inode for a file, file for an opened file, superblock for a whole filesystem, dentry for a directory**.
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table, which has addresses of routines to implement that function on that object. For example:
  - `int open(. . .)` — Open a file
  - `int close(. . .)` — Close an already-open file
  - `ssize_t read(. . .)` — Read from a file
  - `ssize_t write(. . .)` — Write to a file
  - `int mmap(. . .)` — Memory-map a file

Challenge: renaming files

- How would we implement `rename`?

Challenge: renaming files

- How would we implement `rename`?
- Renaming only changes the name of the file
- Directory contains the name of the file
- No data needs to be moved, inode remains unchanged

Challenge: renaming files

- How would we implement `rename`?
- Renaming only changes the name of the file
- Directory contains the name of the file
- No data needs to be moved, inode remains unchanged
- Note, we may need to move the data if it is on another disk/partition!

Filesystem implementation

- A filesystem is an exercise in data management
- Given: a large set (N) of blocks
- Need: data structures to encode (i) file hierarchy and (ii) per file metadata
  - Overhead (metadata size versus file data) should be low
  - Internal fragmentation should be low
  - File contents must be accessed efficiently (external fragmentation, number of metadata accesses)
  - Define operations for file API

“Files” — bytes vs. disk sectors

- Files are sequences of bytes
  - Granularity of file I/O is bytes
- Disks are arrays of sectors (512 bytes)
  - Granularity of disk I/O is sectors
  - File data must be stored in sectors
- A file system defines a **block** size
  - block size = 2^n * sector size
  - Contiguous sectors are allocated to a block

File systems' view of the disk partition

- File systems view the disk partition as an array of blocks
  - It needs to allocate blocks to file
  - It also needs to manage free space on disk
- But how?
- Objective:
  - Disk space utilized efficiently
  - Files can be accessed quickly

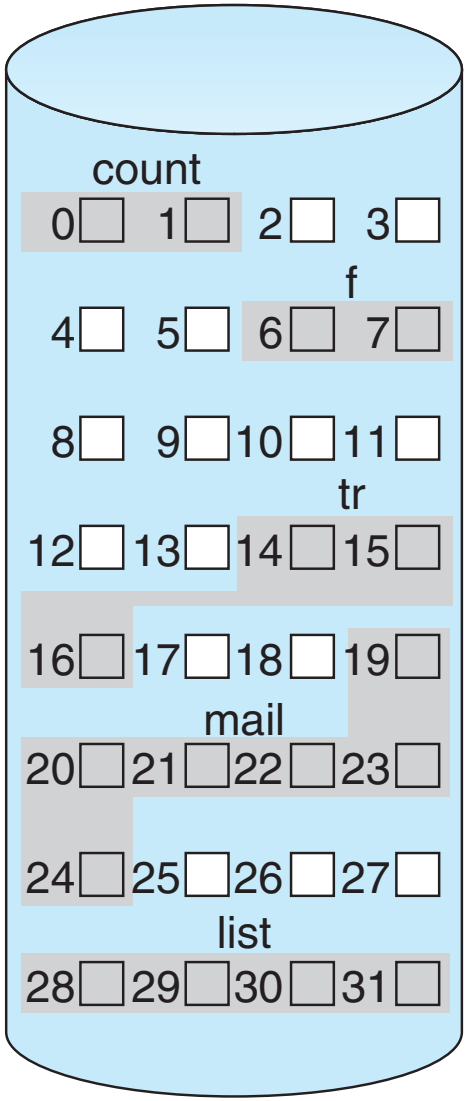
Allocating file data

- Contiguous
- Linked blocks (blocks end with a next pointer)
- File-allocation tables (table that contains block references)
- Indexed (inode contains data pointers)
- Multi-level indexed (tree of pointers)

For each approach, think about fragmentation, ability to grow/shrink files, sequential access performance, random access performance, overhead of meta data.

Try 1: Contiguous Allocation

Idea: All blocks in a file are contiguous on the disk



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

- Advantages —
  - Simple to implement (only needs starting block & length of file)
  - Good performance (for sequential reading)
- Disadvantages —
  - After deletions, disk becomes fragmented — **external fragmentation**
  - Will need periodic compaction — time-consuming
  - If new file is placed at end of disk, no problem
  - If new file is placed into a "hole", must know a file's maximum possible size, **at the time it is created!**

Contiguous Allocation

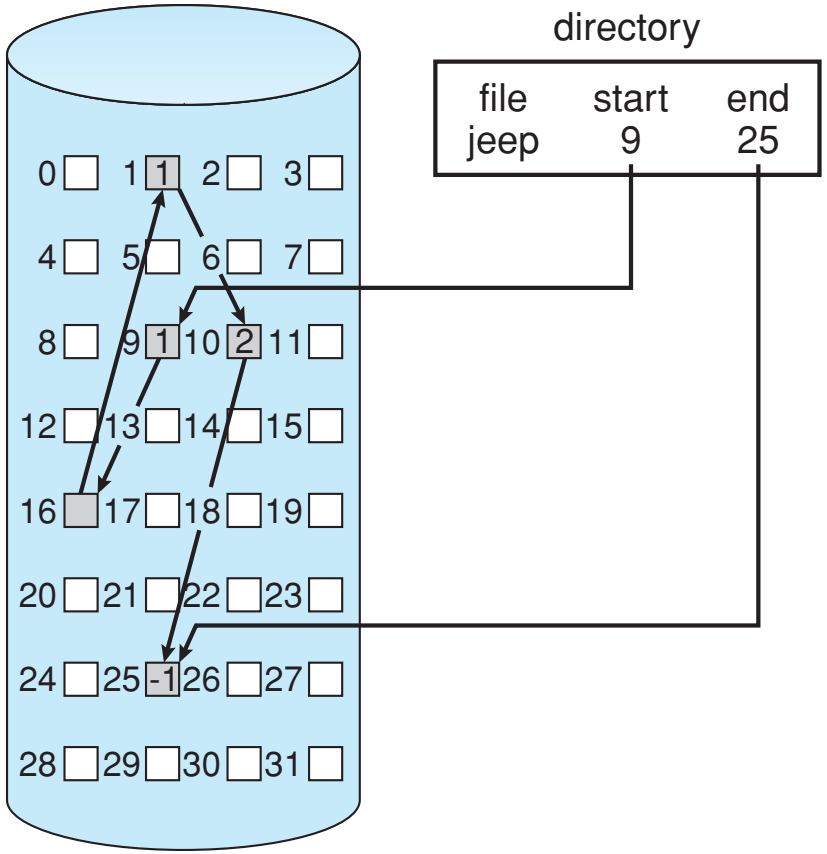
- What is it good for, then?
- Good for CD-ROMs and DVDs
  - All file sizes are known in advance
  - Files are never deleted
- UDF (Universal Disk Format)
  - Uses 30 bits to represent the length of a file
  - Accommodates up to 1GB
  - For DVD movies, 4 1-GB files may be necessary
- a modified contiguous-allocation scheme
  - if not enough, another chunk of contiguous space is added, called **extents**
  - A good idea to use extents for file systems? — Veritas FS

Try 2: Linked Allocation

Each file is a sequence of blocks

First word in each block contains a pointer to the next block

Random access into the file is slow!



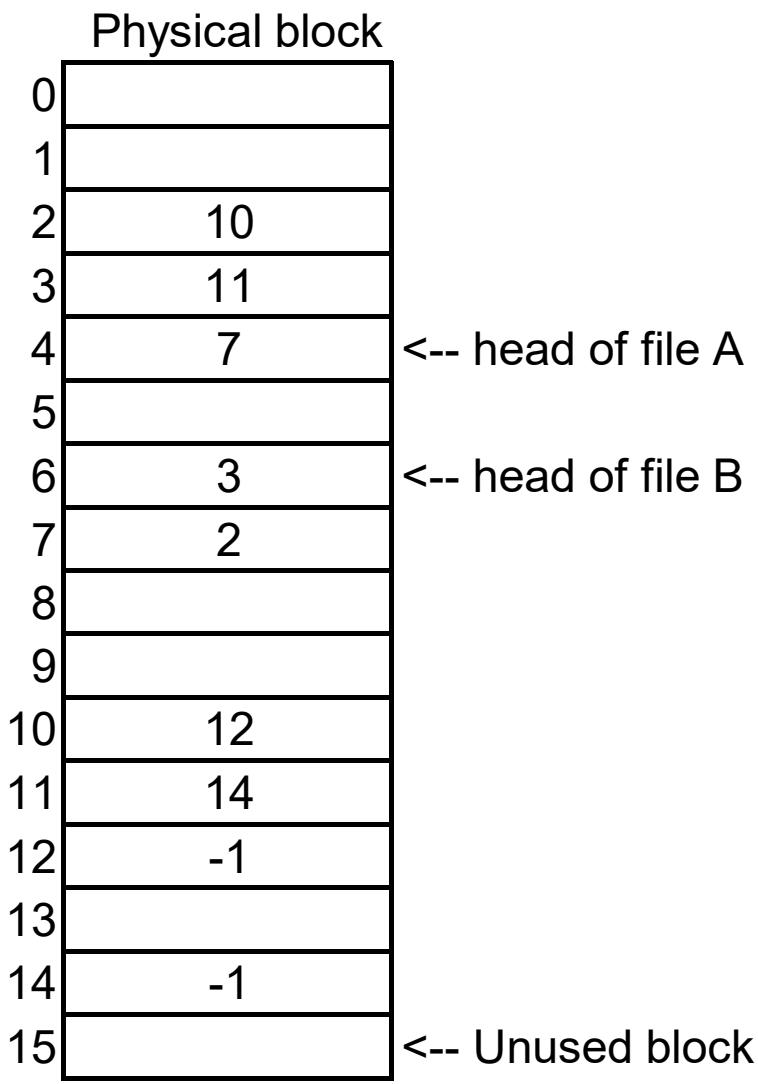
Linked Allocation

- Advantages —
  - No external fragmentation
  - The size of the file need not be declared when the file is created
- Disadvantages —
  - Can only be used for sequential access: random access is slow
  - Space required by pointers: overhead
    - mitigated by using **clusters** of blocks as unit
    - but the use of clusters increases **internal fragmentation**
  - Reliability: a damaged block leads to a bad pointer

Variation: File Allocation Table (FAT)

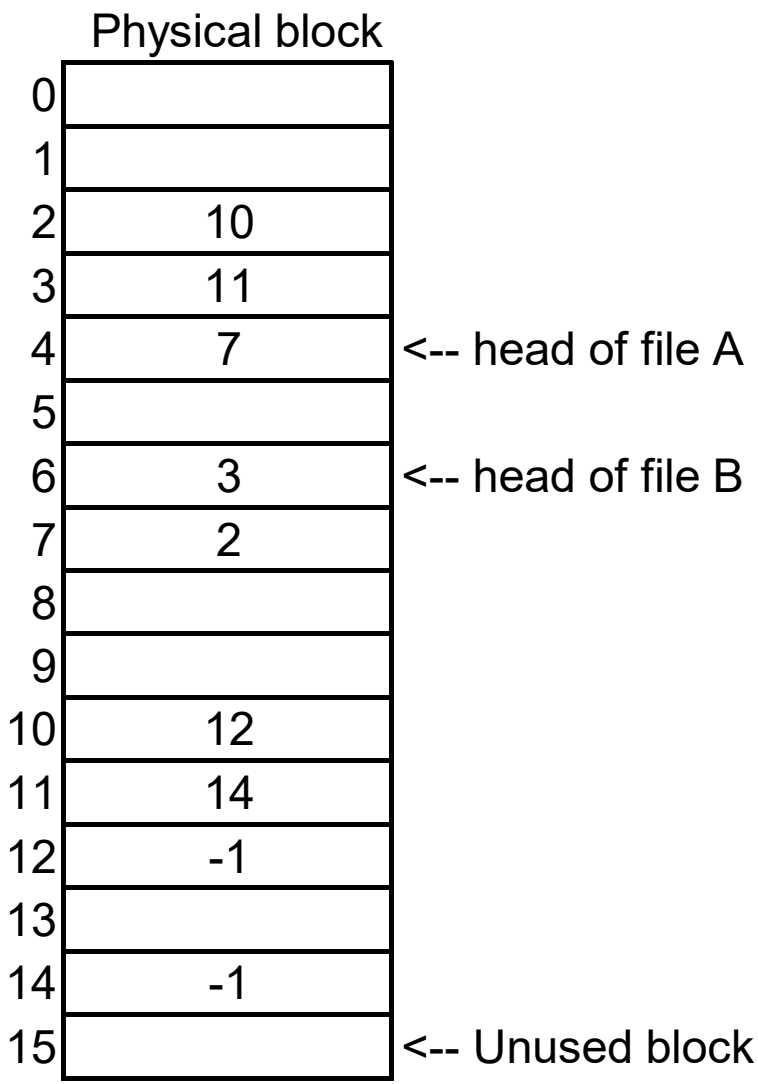
- Keep a table at the beginning of disk volume and in memory
- One entry per block on the disk
- Each entry contains the address of the "next" block
  - "End of file" marker is -1
- A special value (0) indicates that the block is free
- Used in MS-DOS and IBM OS/2

File Allocation Table (FAT)



Which blocks does each file contain respectively?

File Allocation Table (FAT)



Which blocks does each file contain respectively?

```
File A: 4->7->2->10->12
File B: 6->3->11-14
```

File Allocation Table (FAT)

- Random access
  - Search the linked list (but all in memory)
- Directory entry needs only one number
  - The starting block number
- **Disadvantage —**
  - Entire table must be in memory all at once!
  - Example:
    - 20 GB = disk size
    - 1 KB = block size
    - 4 bytes = FAT entry size
    - 80 MB of memory used just to store the FAT!

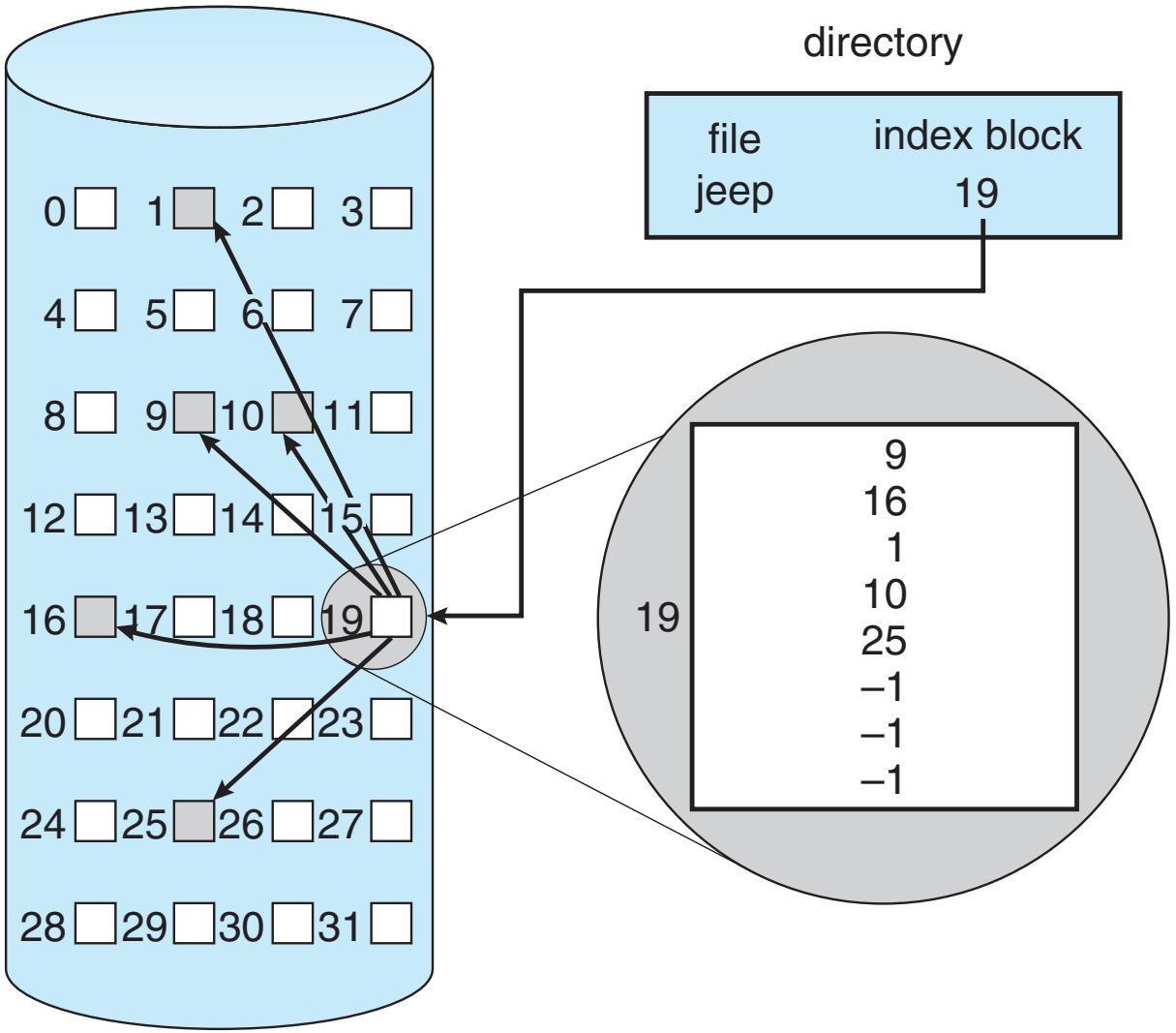
What should we do now?

- The file system designer's dilemma
  - If we don't cache the file allocation table, random access is slow
  - If we do cache it, we don't have enough memory!
- But why do we need to cache the entire file allocation table?
- Can we **cache only** parts of the file allocation table, corresponding to the files that are declared "**open**" by the user programs?
  - We need to add "open" and "close" to the system-call interface for the applications to "open" a file
  - Then we can work only with "open" files!

Try 3: Indexed Allocation

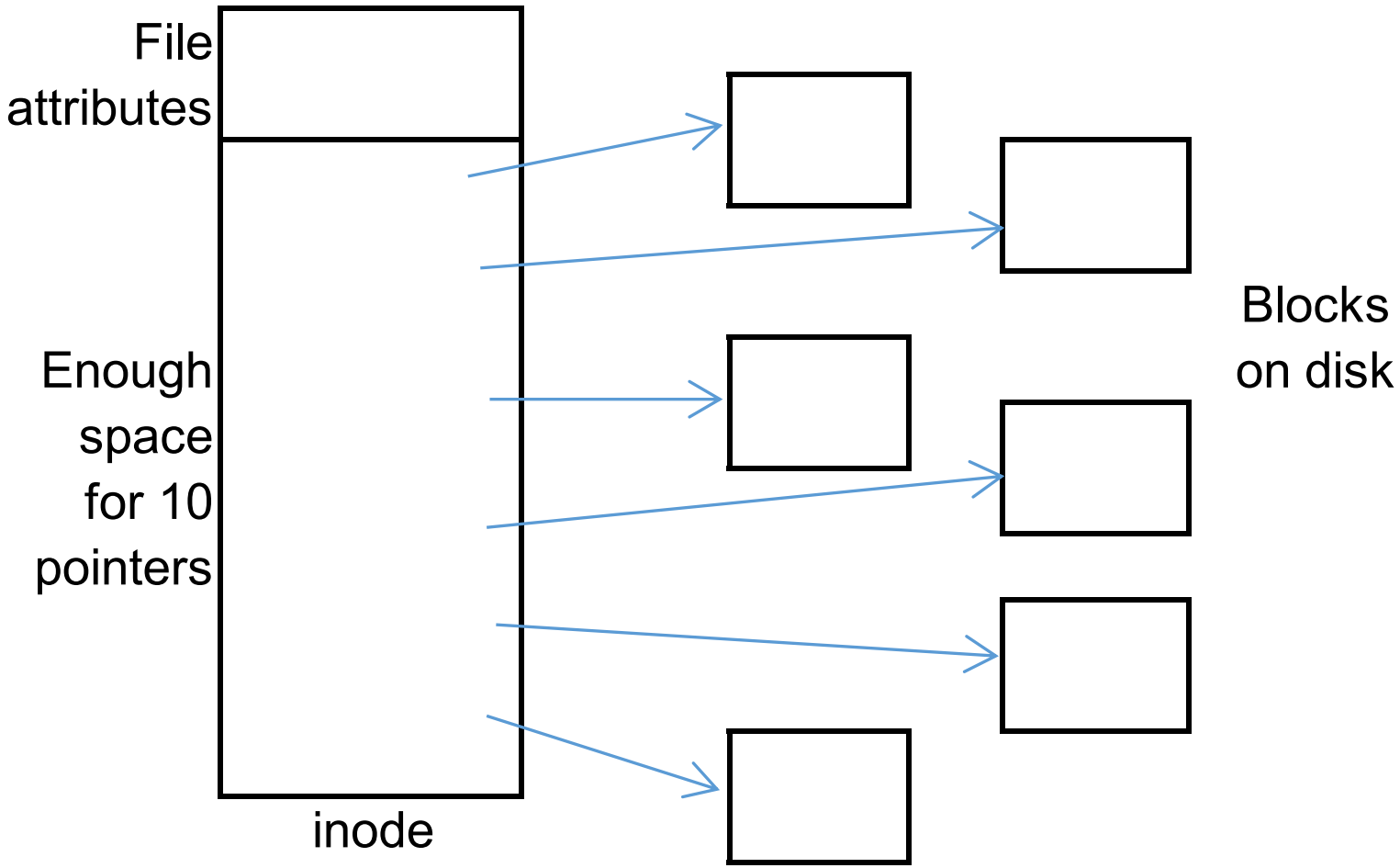
- Idea: Bring all the pointers together into one location – the **index block**
- Each file has its own index block: an array of disk-block addresses
  - The i^{th} entry points to the i^{th} block of the file
  - The directory contains the address of the index block of the file

Indexed allocation



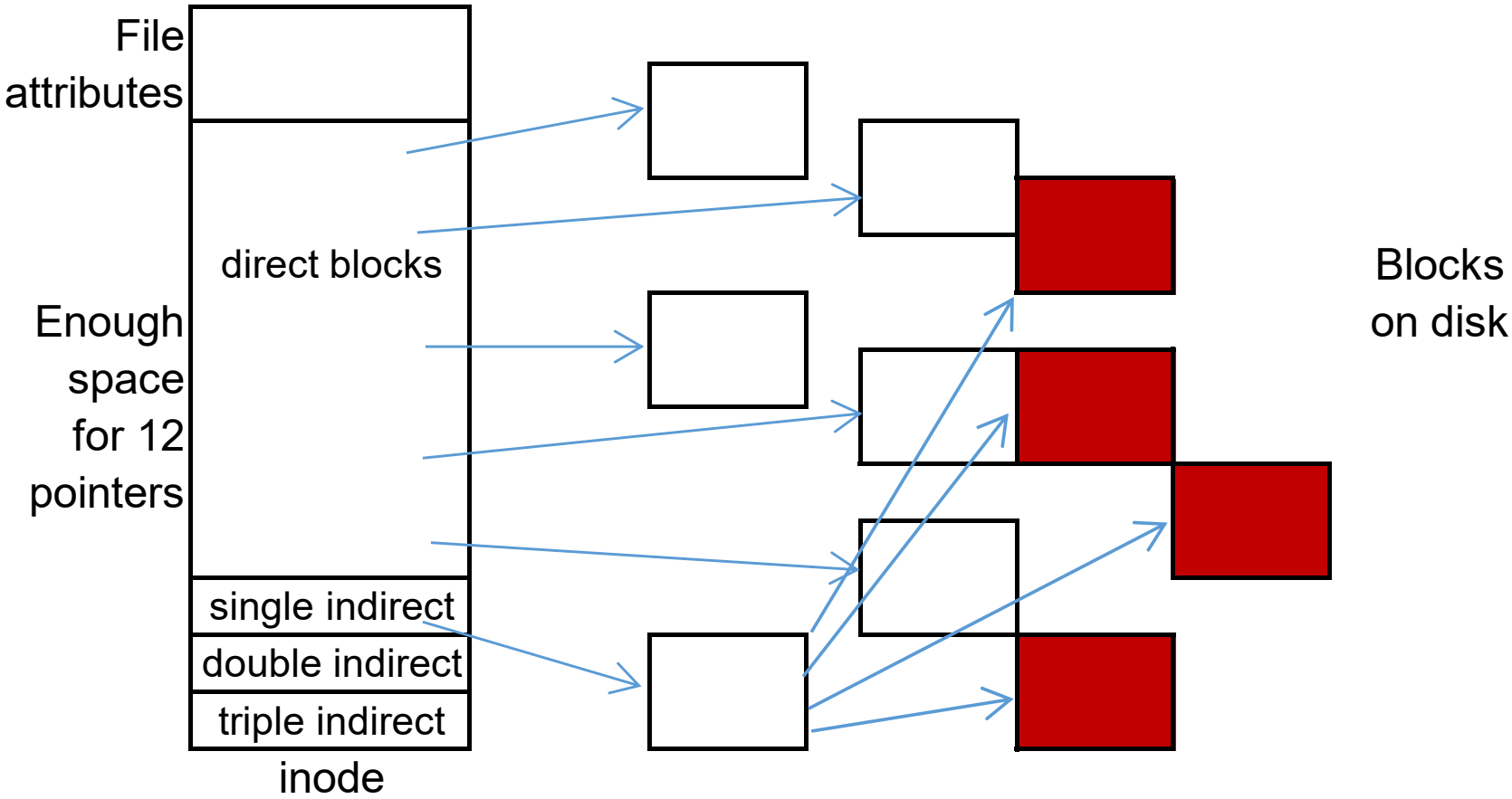
FCB example: inodes in UNIX

- Each inode ("index-node") contains
  - file attributes (permissions, timestamps, owner)
  - the index block



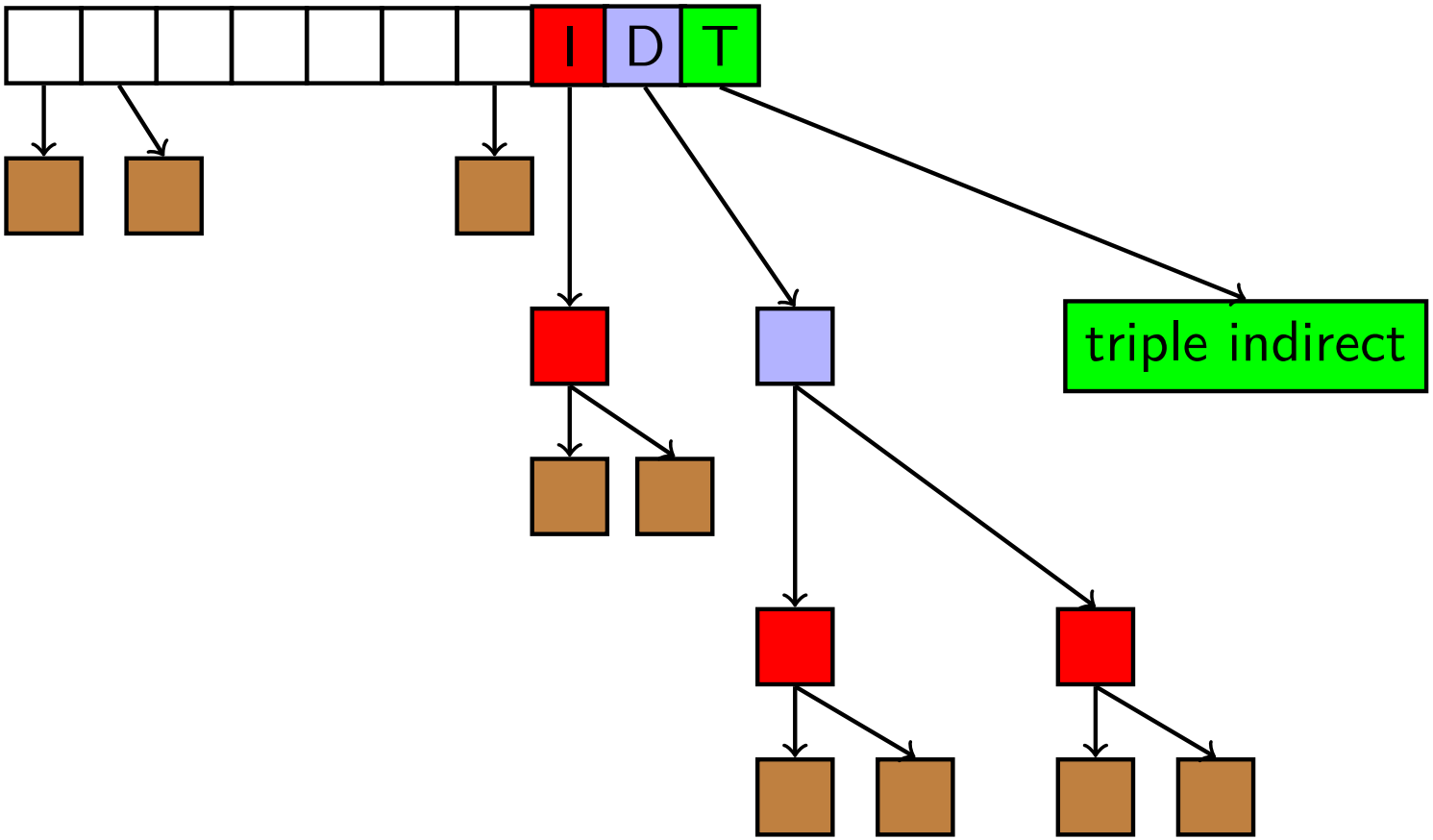
FCB example: inodes in UNIX

- But what if we have a large file?
  - If we increase the size of the index block, all files (including small files) will use the new size for theirs
  - Solution: **multi-level indexing**



File allocation: multi-level indexing (1/2)

Idea: have a mix of direct, indirect, double indirect, and triple indirect pointers



File allocation: multi-level indexing (2/2)

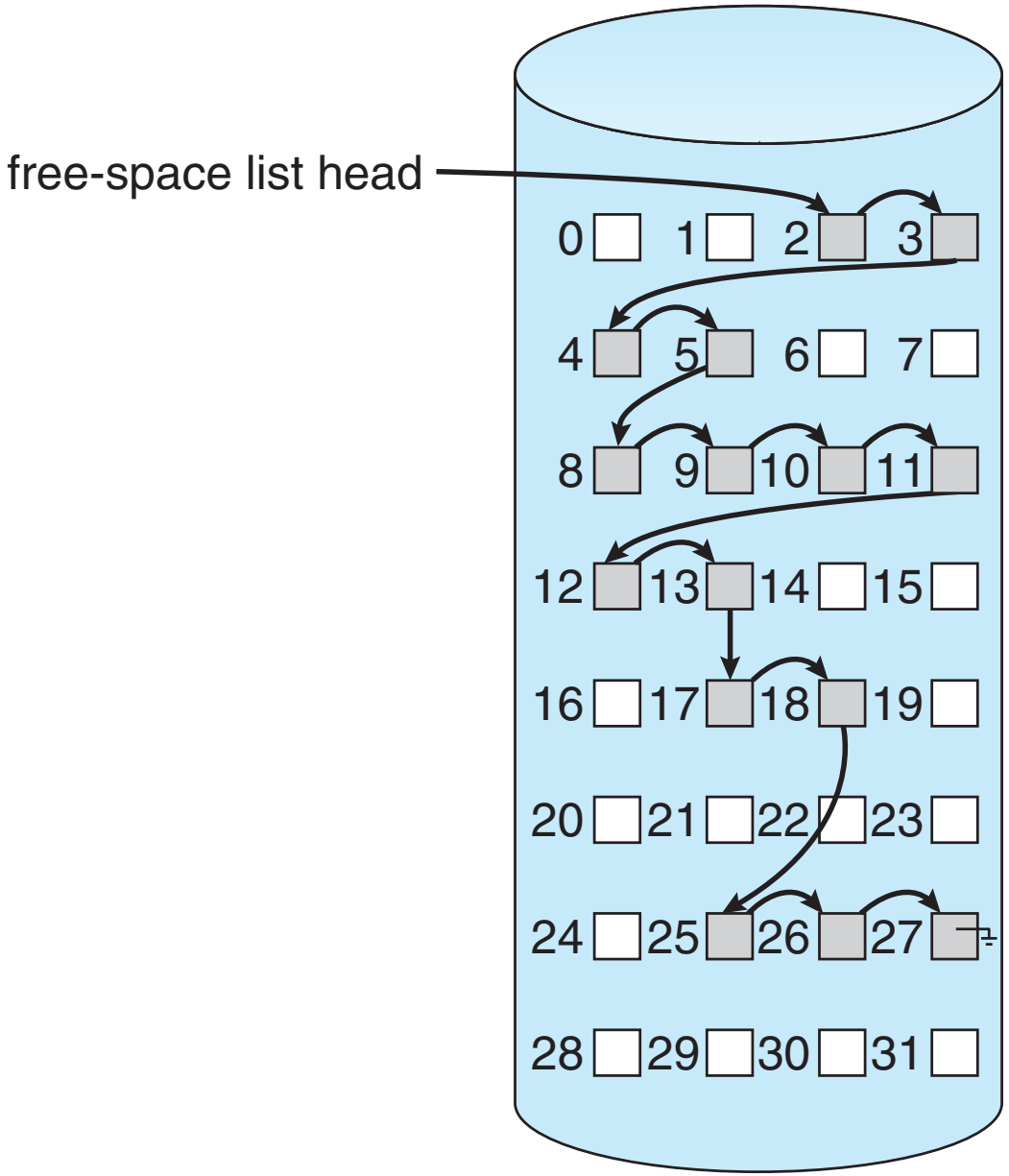
Idea: have a mix of direct, indirect, double indirect, and triple indirect pointers

- No external fragmentation
- Files can grow easily up to maximum size
- Reasonable read and low seek cost
- Low metadata overhead but needs extra reads for indirect/double indirect access

Managing free blocks

- A bitmap: 1 = used, 0 = free
  - 1.3 GB disk partition, 512-byte blocks: over 332 KB to contain the bitmap — performance is only satisfactory if it is cached
  - 1 TB disk with 4-KB blocks: 32 MB required!
- A linked list of disk block numbers
  - Use a disk block to contain all the free block numbers
  - Use one block number in the disk block to point to the next disk block
  - 1 KB block size, 32 bit block numbers: 255 free blocks
  - 500 GB disk partition: 488 million blocks in total, 1.9 million blocks required to contain free block numbers!

Keeping Track of Free Blocks



Counting

Usually, several contiguous blocks may be allocated or freed simultaneously

Idea: rather than keeping a list of `n` free disk addresses, we can keep the address of the first free block, and the number (`n`) of free contiguous blocks that follow the first block

Each entry in the free-space list consists of a **disk address** and a **count** - Used in Sun's ZFS

Creating and Deleting Files

- Only needs to maintain one block of bitmaps in the main memory (if bitmap is used)
  - A natural advantage is to be able to allocate free blocks contiguously when a file is created
- Only needs to maintain one block of free disk blocks in the main memory (if linked list is used)
  - When a file is created, needed blocks are taken from the in-memory block of free disk blocks
  - When it runs out, a new block of pointers is read in from the disk
  - When a file is deleted, its blocks are added to the block, and later written to disk when it is full

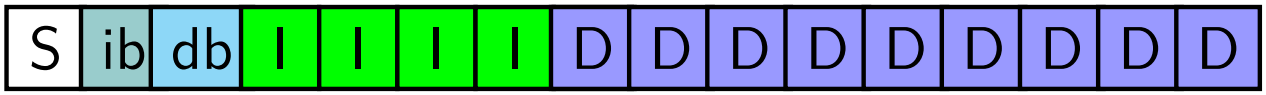
Improving File System Performance

- Problem: Disk operations are slow!
- **Solution: The buffer cache**
  - Upon a read() system call, first check if the needed block is in the cache
  - If not, it is first read into the cache, and then copied to wherever it is needed
  - Subsequent requests to the same block can be satisfied without disk access
- Needs a cache replacement algorithm when the buffer cache is full: **LRU** is a good choice

Synchronous vs. asynchronous write

- Write back (asynchronous write)
  - data are stored in the buffer cache and written back to the disk at a later time asynchronously
  - Unix: **update** daemon uses the **sync system call** forces all modified blocks to the disk every 30 seconds
- Write through (synchronous write)
  - writes are not buffered (only reads are buffered)

Simple FS



- Superblock (S): file system metadata
- Bitmaps (ib, db): indicates free blocks
- Inodes (I): hold file/directory metadata, reference data blocks
- Data blocks (D): file contents, referenced by an inode

The inode size may be different (smaller) from the data block size.

Simple FS: superblock

- The superblock stores the characteristics of the filesystem
- What do you store in the superblock?

Simple FS: superblock

- The superblock stores the characteristics of the filesystem
- What do you store in the superblock?

- Magic number and revision level
- Mount count and maximum mount count
- Block size of the filesystem (1, 2, 4, 8, 16, 32, 64K for ext4)
- Name of the filesystem
- Number of inodes/data blocks
- Number of free inodes/data blocks
- Number of "first" inode (i.e., root directory)

Simple FS: inode

- The inode stores all file metadata
- What would you store in an inode?

Simple FS: inode

- The inode stores all file metadata
- What would you store in an inode?

- File type
- File uid, gid
- File permissions (for user, group, others)
- Size
- Access time
- Create time
- Number of links

Maximum file size

- Maximum file size is related to
  - Block size
  - Number of direct inodes
  - Number of indirect inodes
  - Number of double indirect inodes
  - Number of triple indirect inodes
- `blocksize * (direct + inodeblock + inodeblock^2 + inodeblock^3)`

Directories

- Directories are special files (`inode->type`)
- Store a set of file name to inode mappings
- Special entries `.` for current directory and `..` for parent directory

File operation: create /foo/bar

- Read root inode (locate directory data)
- Read root data (read directory)
- Read foo inode (locate directory data)
- Read foo data (read directory)
- Read/write inode bitmap (allocate inode)
- Write foo data (add file name)
- Read/write bar inode (create file)
- Write foo inode (update date, maybe allocate data block)

File operation: open /foo/bar

- Read root inode (locate directory data)
- Read root data (read directory)
- Read foo inode (locate directory data)
- Read foo data (read directory)
- Read bar inode (read file metadata)

File operation: write to /foo/bar

- First: `open("/foo/bar")`
- Read bar inode (read file metadata)
- Read/write data bitmap (allocate data blocks)
- Write bar data (write data)
- Write bar inode (update inode)

File operation: read from /foo/bar

- First: `open("/foo/bar")`
- Read bar inode (read file metadata)
- Read bar data (read data)
- Write bar inode (update time)

File operation: close /foo/bar

- No disk I/O

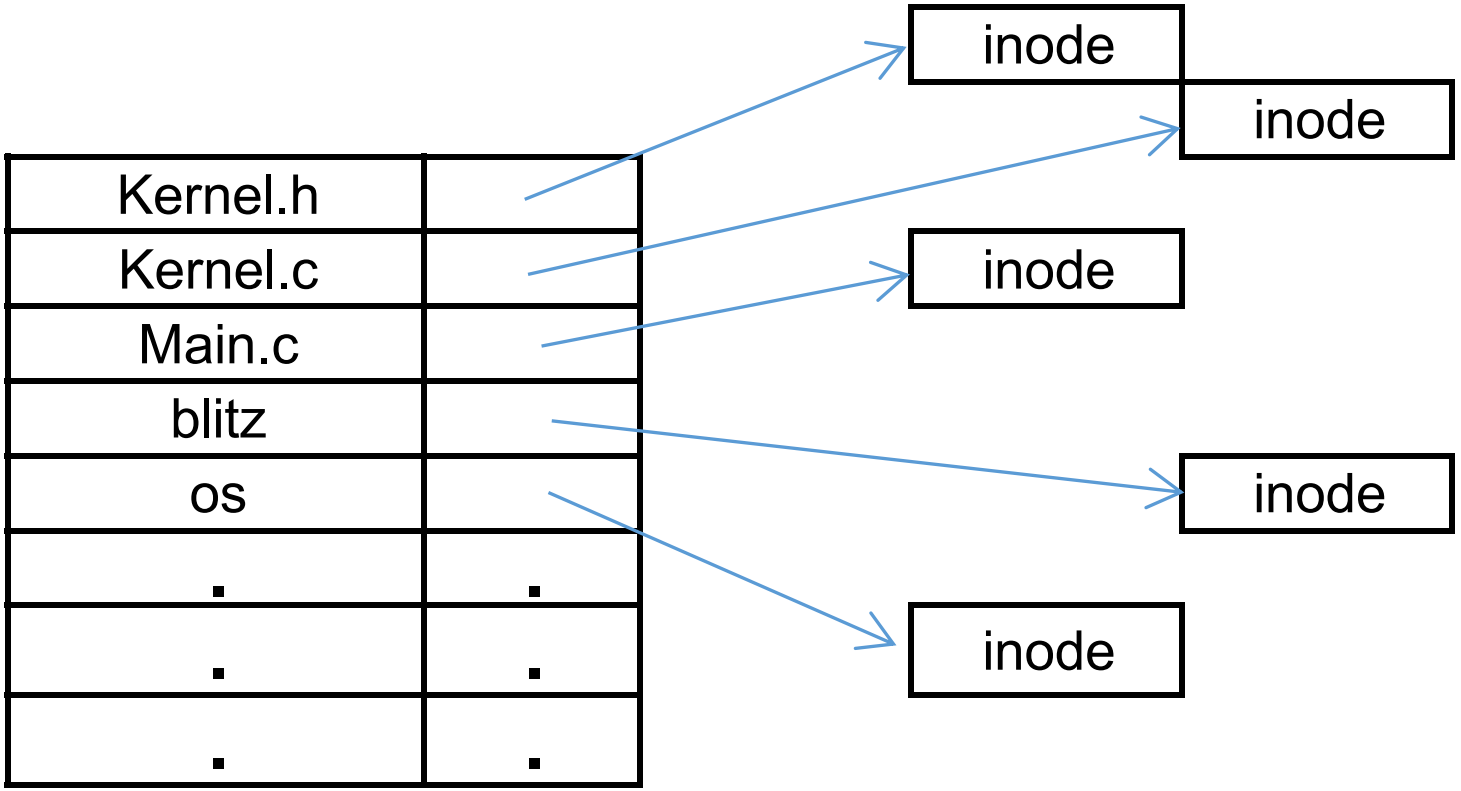
File operation: observations

- Path traversal and translation is costly
  - Reduce number of lookups (file descriptors!)
  - Introduce caching (dcache)
- Lookup aside, operations are cheap and local

Example inode in the Linux ext2 file system

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file acl	a new permissions model beyond mode bits
4	dir acl	called access control lists

Implementing Directories with Linear List



But finding a file requires a linear search — **expensive!**

Searching for a File with a Path

- We wish to search for a file `/usr/bin/blitz`
  - Locates the root directory (inode 2)
  - Looks up the string "usr" in the root directory for the inode number of the /usr directory
  - The inode of the /usr directory is fetched, string "bin" searched
  - The inode of the /usr/bin directory is used to look up "blitz" for its inode number
- How do we improve its performance?
  - Caching all results of previous searches
  - Try to find a match for subsequent searches

Sharing Files

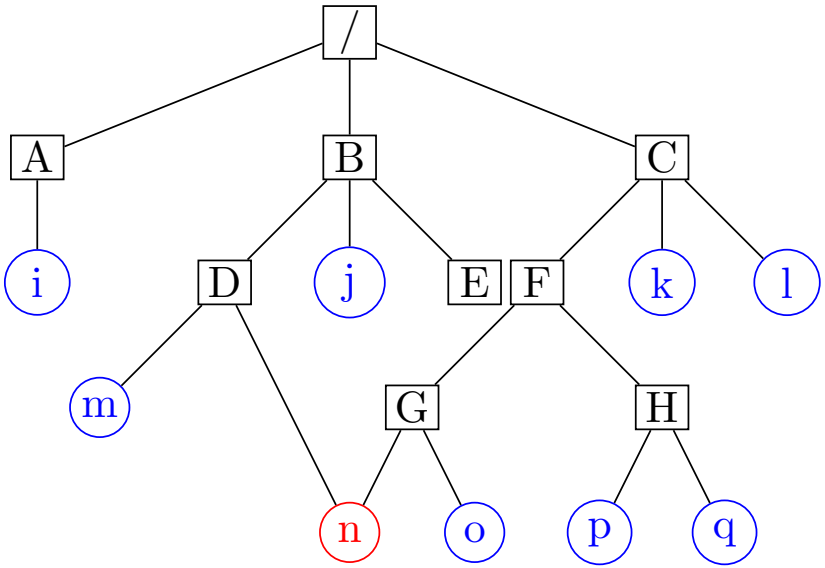
One file appears in several directories

Tree $\rightarrow$ DAG (Directed Acyclic Graph)

What if the file changes?

New disk blocks are used.

Better not store this info in the directories!



Hard Links and Symbolic Links In Unix

- Hard links
  - Both directory entries point to the same inode
- Symbolic links
  - One directory entry points to the file's inode
  - Other directory entries contains the "path"

Hard Links

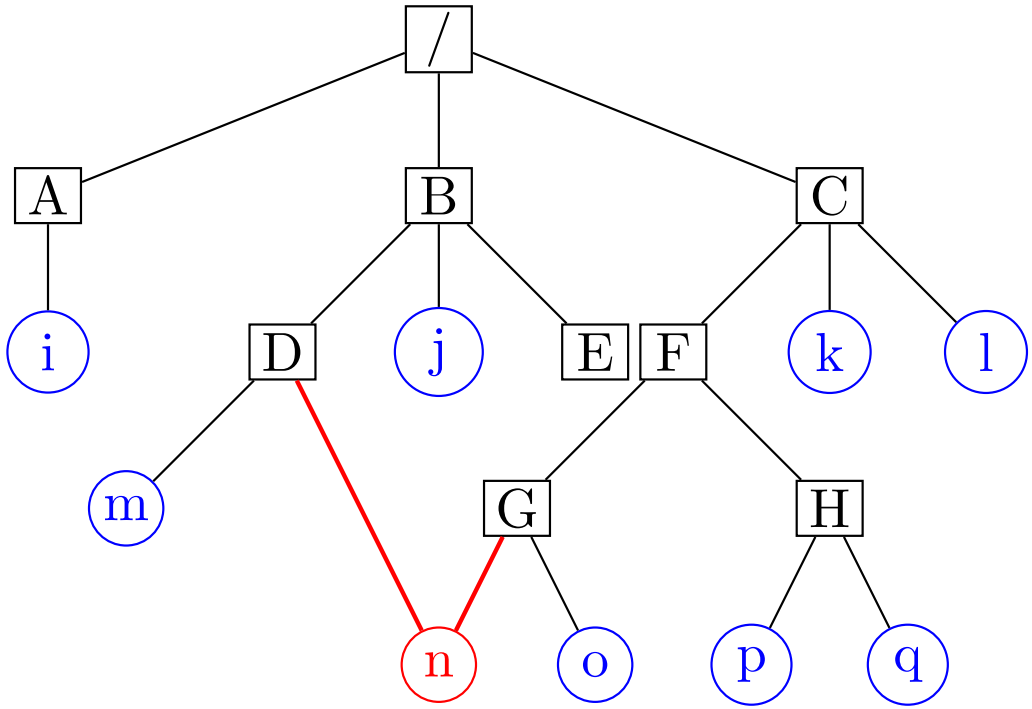
Assume inode number of “n” is 45

Directory “D”

“m”	123
“n”	45
...	...

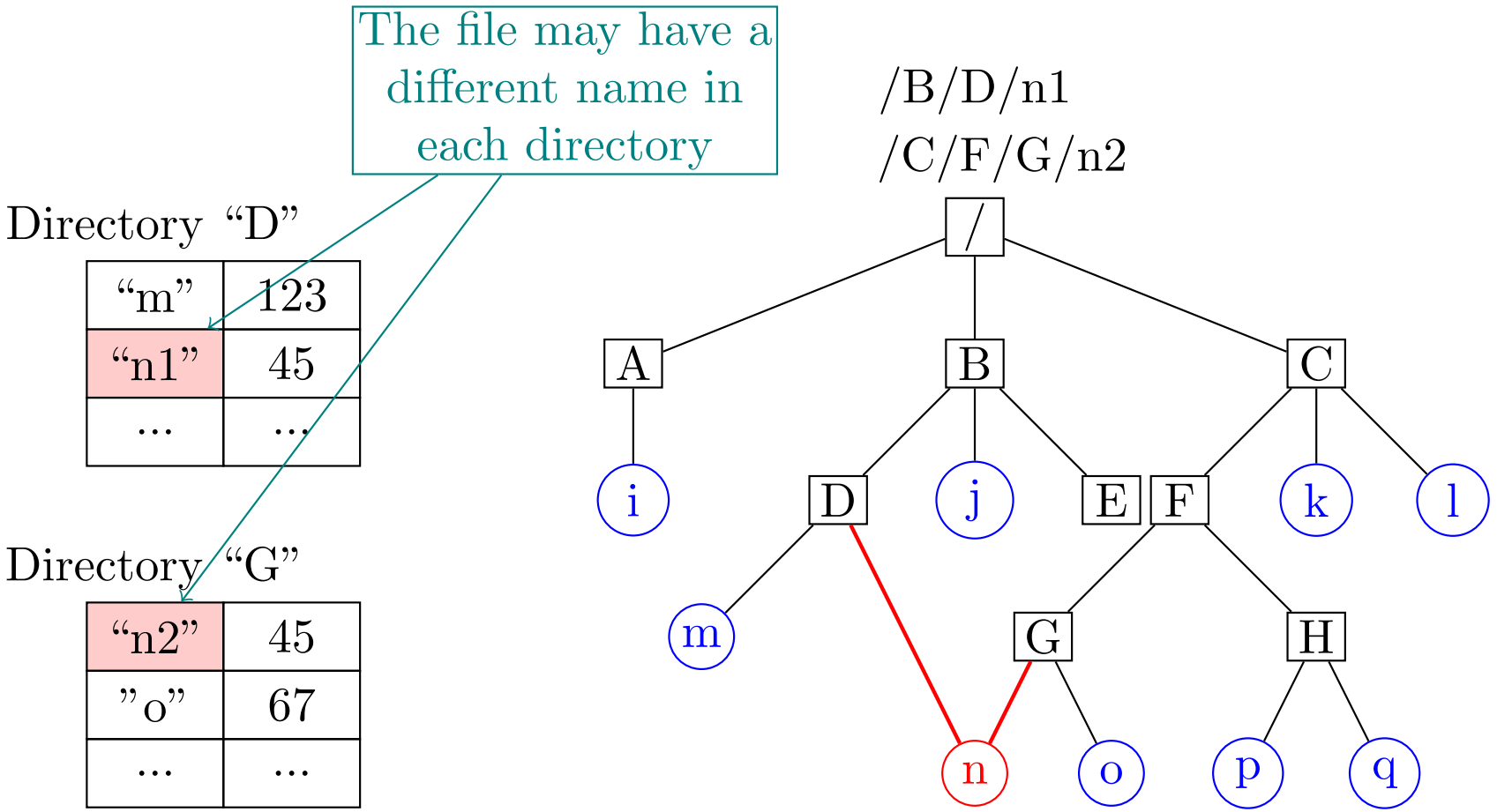
Directory “G”

“n”	45
”o”	67
...	...



Hard Links

Assume inode number of “n” is 45



Symbolic Links

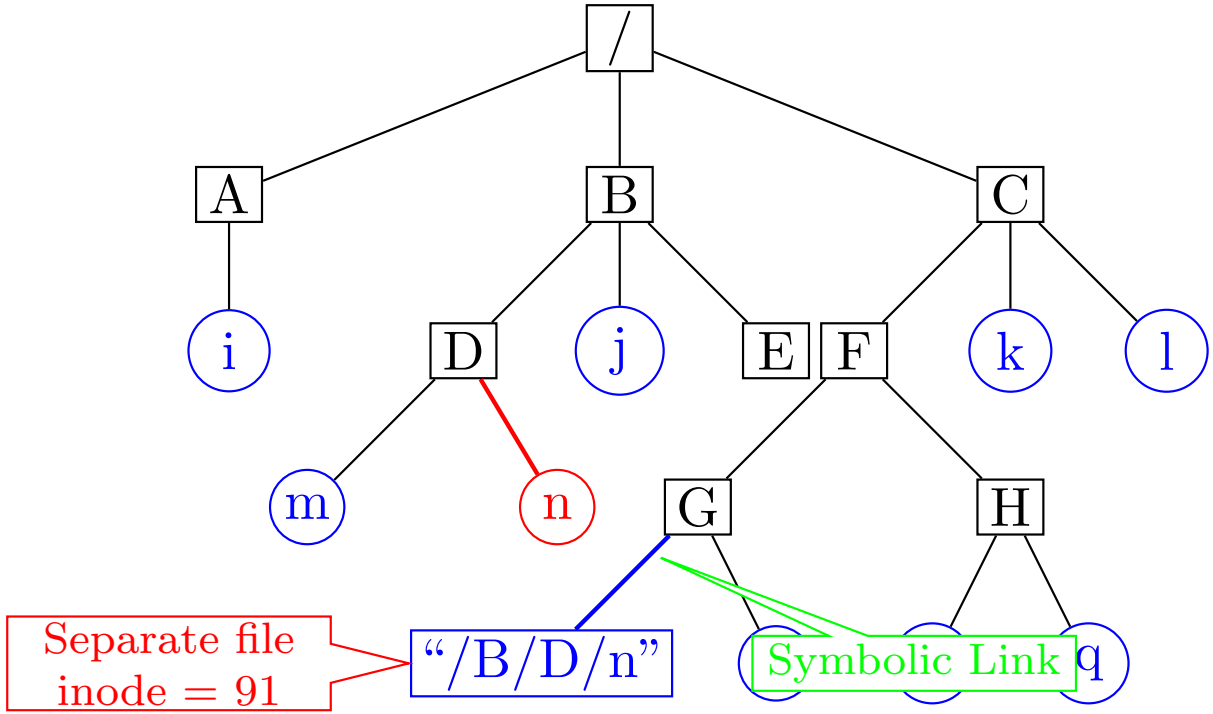
Assume inode number of “n” is 45

Directory “D”

“m”	123
“n”	45
...	...

Directory “G”

“n”	89
”o”	67
...	...



Deleting a File

- Directory entry is removed from directory
- All blocks in file are returned to the free list
- What about sharing?
  - Multiple links to one file (in Unix)
- Hard Links
  - Put a "**reference count**" field in each inode
  - Counts the number of directories that point to the file
  - When removing file from directory, decrement the count
  - When count goes to zero, reclaim all blocks in the file
- Symbolic Link
  - Remove the actual file (normal file deletion)
  - Symbolic link becomes "broken"

Opening a file using the open() system call

- The open() call passes a file name to the file system
- It first searches the **system-wide open-file table** to see if the file is already in use by another process
- If it is, a **per-process open-file table** entry is created, pointing to the existing system-wide open-file table
- If not, the directory structure is searched for the given file name
  - Parts of the directory structure are cached in memory to improve performance
- Once found, the **File Control Block (inode in UNIX)** is copied into a system-wide open-file table in memory
  - This table not only stores the FCB, but also tracks the number of processes that have the file open

Opening a file (continued)

- An entry is then made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table, and some other fields
  - A pointer to the current location in the file — for next **read() or write()**
  - the access mode in which the file is open (read-only or writable)

The File Descriptor

- The open() call returns an **index** to the corresponding entry in the per-process open-file table
  - called a **file descriptor** in UNIX and BLITZ
  - called a **file handle** in Windows
- This index will be used for subsequent read(), write(), seek(), and close() system calls
- User-level processes must not be allowed to use pointers into kernel memory and cannot be allowed to touch kernel data structures

The File Manager in BLITZ

```
class FileManager
  superclass Object
  fields
   fileManagerLock: Mutex
   fcbTable: array [MAX_NUMBER_OF_FILE_CONTROL_BLOCKS]
             of FileControlBlock //
   anFCBBecameFree: Condition
   fcbFreeList: List [FileControlBlock]
   openFileTable: array [MAX_NUMBER_OF_OPEN_FILES]
                  of OpenFile //
   anOpenFileBecameFree: Condition
   openFileFreeList: List [OpenFile]
   ...
```

The File Control Block in BLITZ

```
class FileControlBlock superclass Listable
 fields
  fcbID: int
  numberOfUsers: int // count of OpenFiles pointing
  startingSectorOfFile: int
  sizeOfFileInBytes: int
  bufferPtr: int
  relativeSectorInBuffer: int
  bufferIsDirty: bool
```

The OpenFile structure in BLITZ: Current position

```
class OpenFile superclass Listable
 fields
  kind: int
  currentPos: int
  fcb: ptr to FileControlBlock
  numberOfUsers: int // count of Processes pointing

class ProcessControlBlock superclass Listable
 fields
    ...
    fileDescriptor: array [MAX_FILES_PER_PROCESS] of
                    ptr to OpenFile //
```

Why do we need to allow multiple PCBs to point to the same OpenFile?

Parent and child processes share an OpenFile

- When a process is cloned with the `fork()` system call, all open files in the parent process must be shared with the child process, with **the current position also shared**
- We now needs reference counting in OpenFile, similar to FCB
  - When the reference count goes to zero, return the OpenFile to the free pool, and decrement the reference count in the corresponding FCB