File System Implementation

Operating Systems

Baochun Li
University of Toronto

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

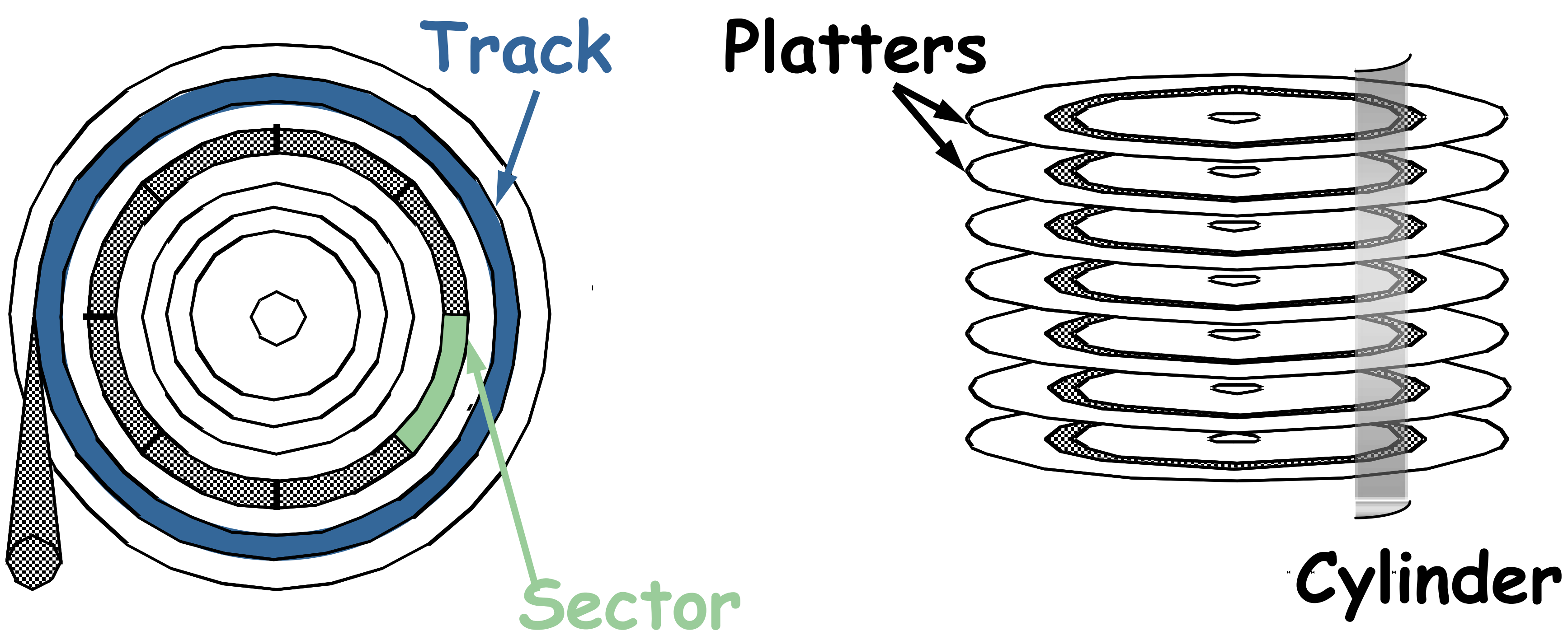
Each platter has an arm and a head

Different heads can access data in parallel

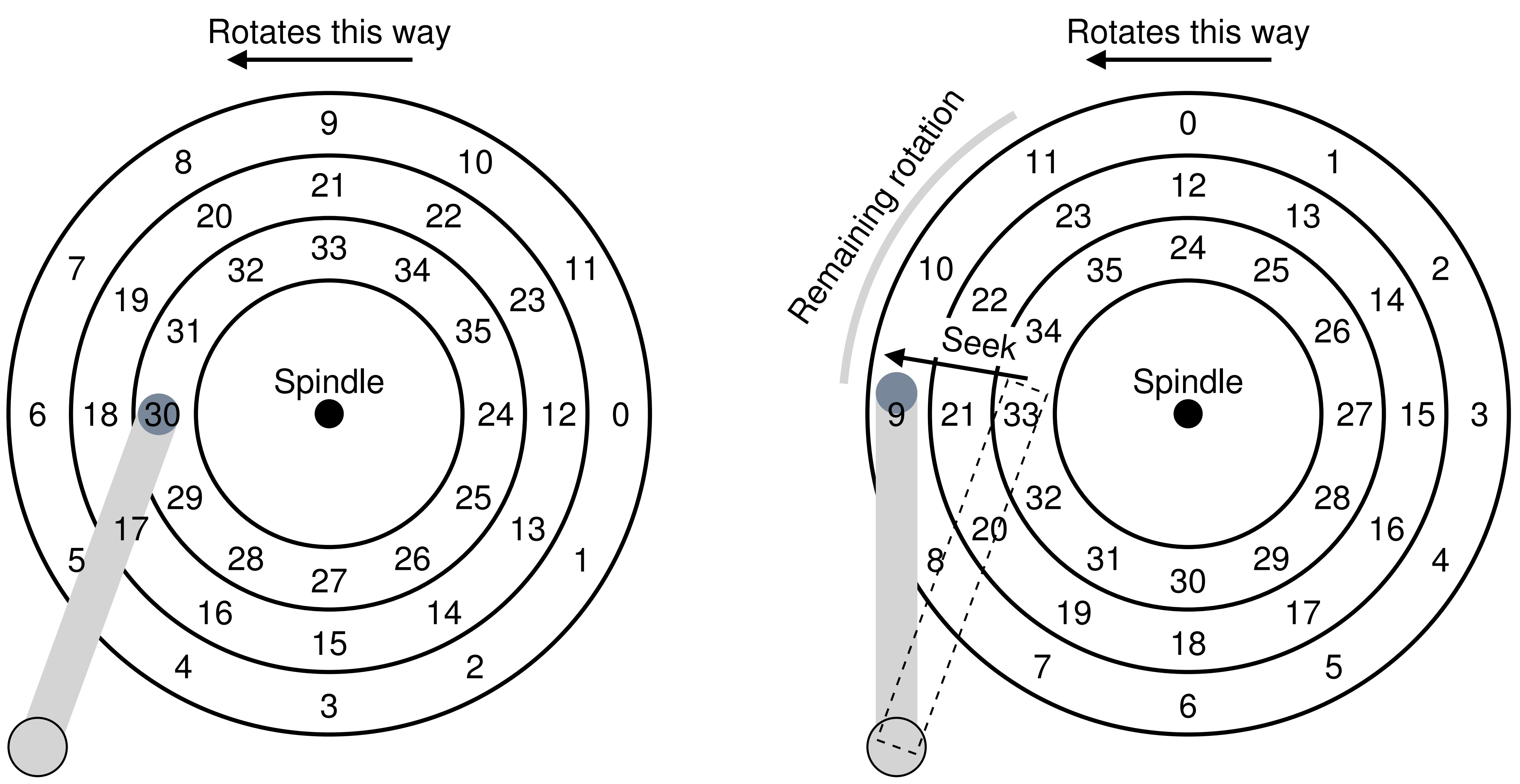
Each platter has multiple concentric tracks

Same set of tracks across all platters is a cylinder

Each track has multiple sectors



Rotational delay and seek time



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
- Called a **bootloader**, this "boot program" reads in an OS from the file system in that partition

Booting the system

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

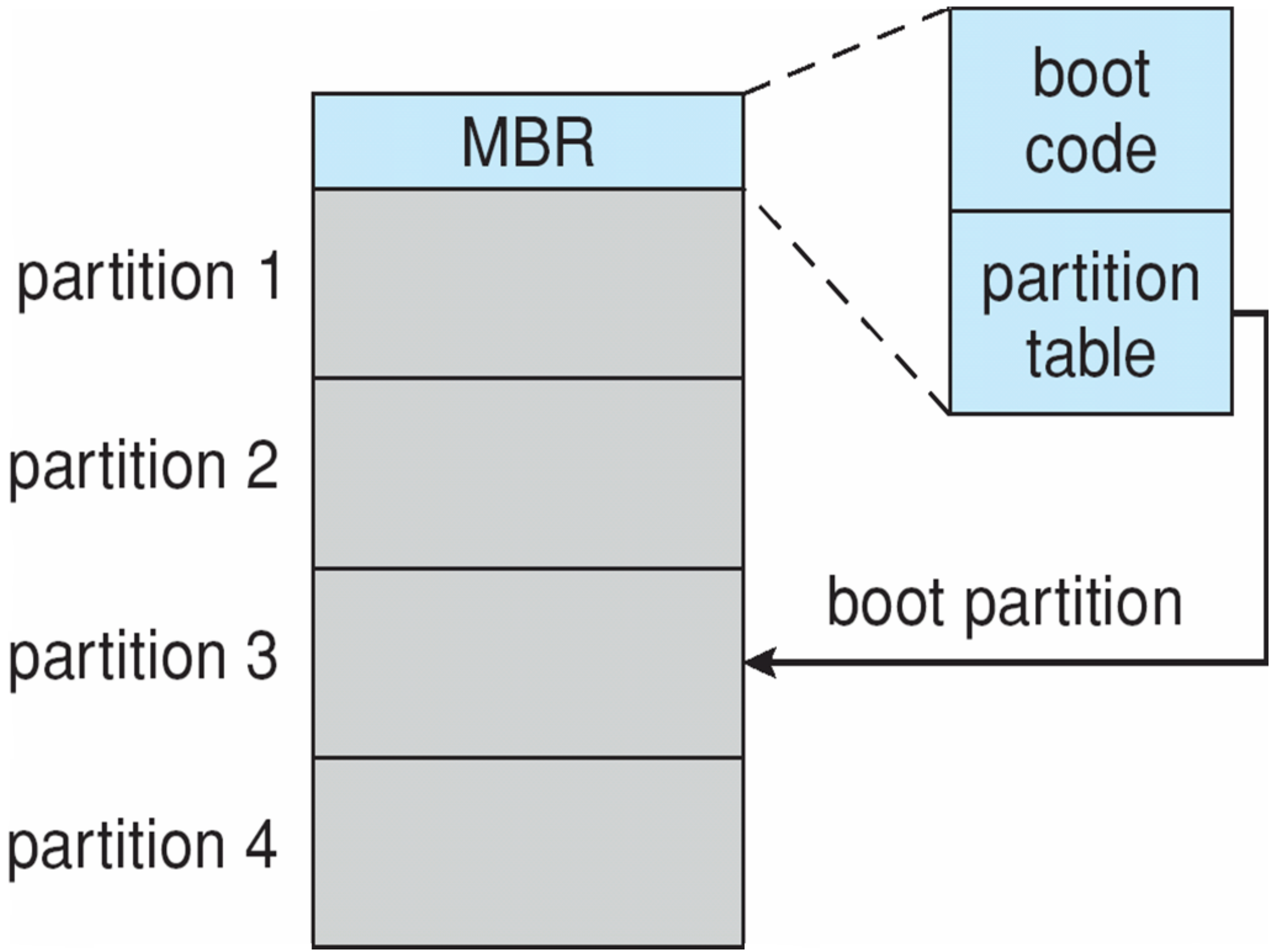
Can boot from drives 2.2TB or larger

Developed in C, rather than assembly

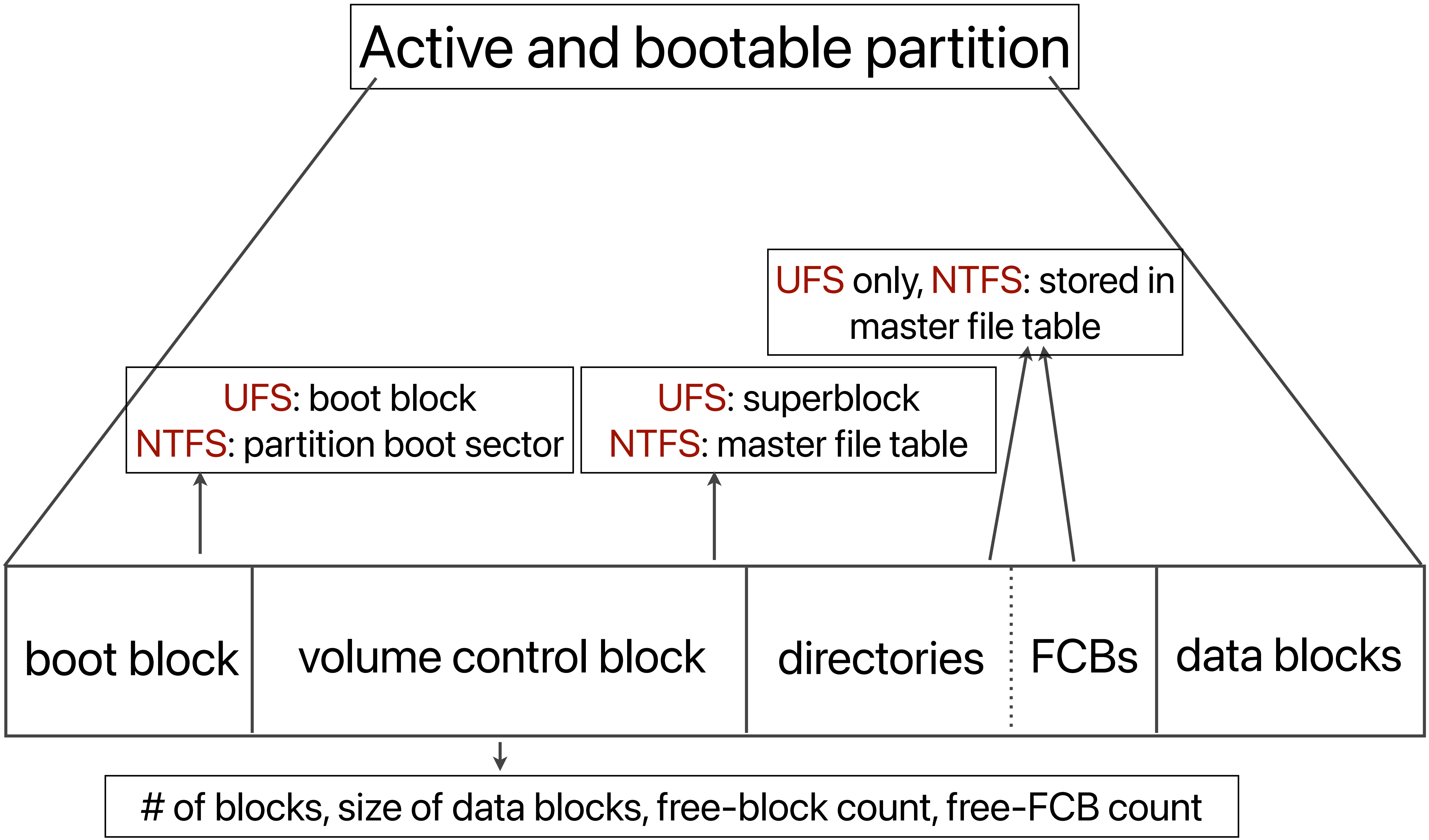
64-bit support and faster boot times

Supports secure boot

An Example Disk



A "cooked" partition with a file system



"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?

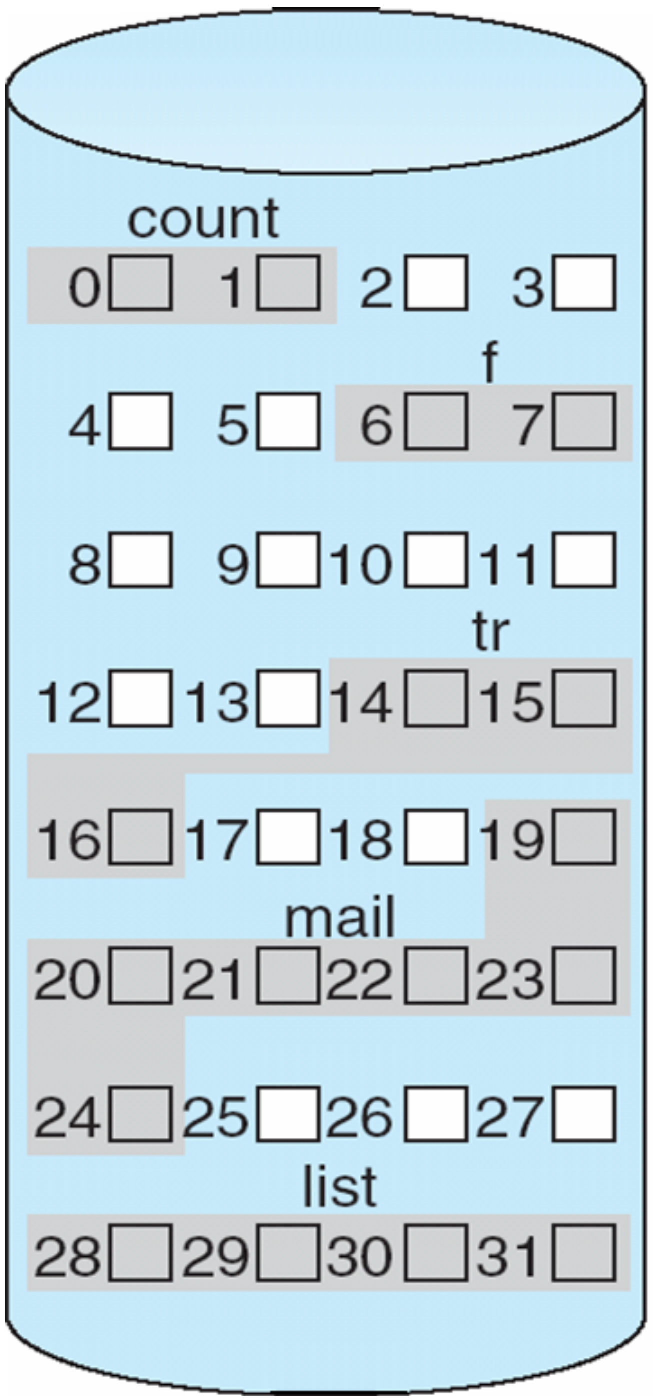
Objective:

- Disk space utilized efficiently
- Files can be accessed quickly

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

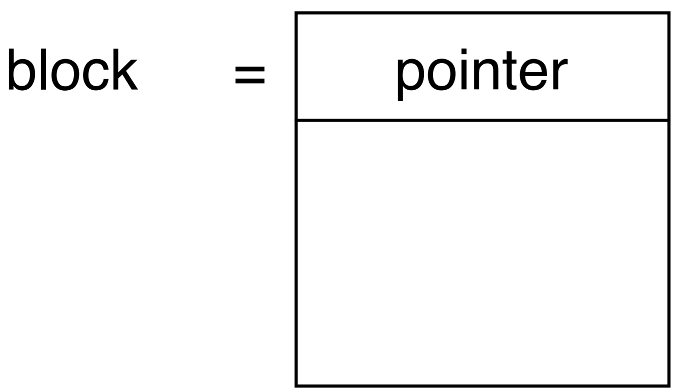
UDF (Universal Disk Format)

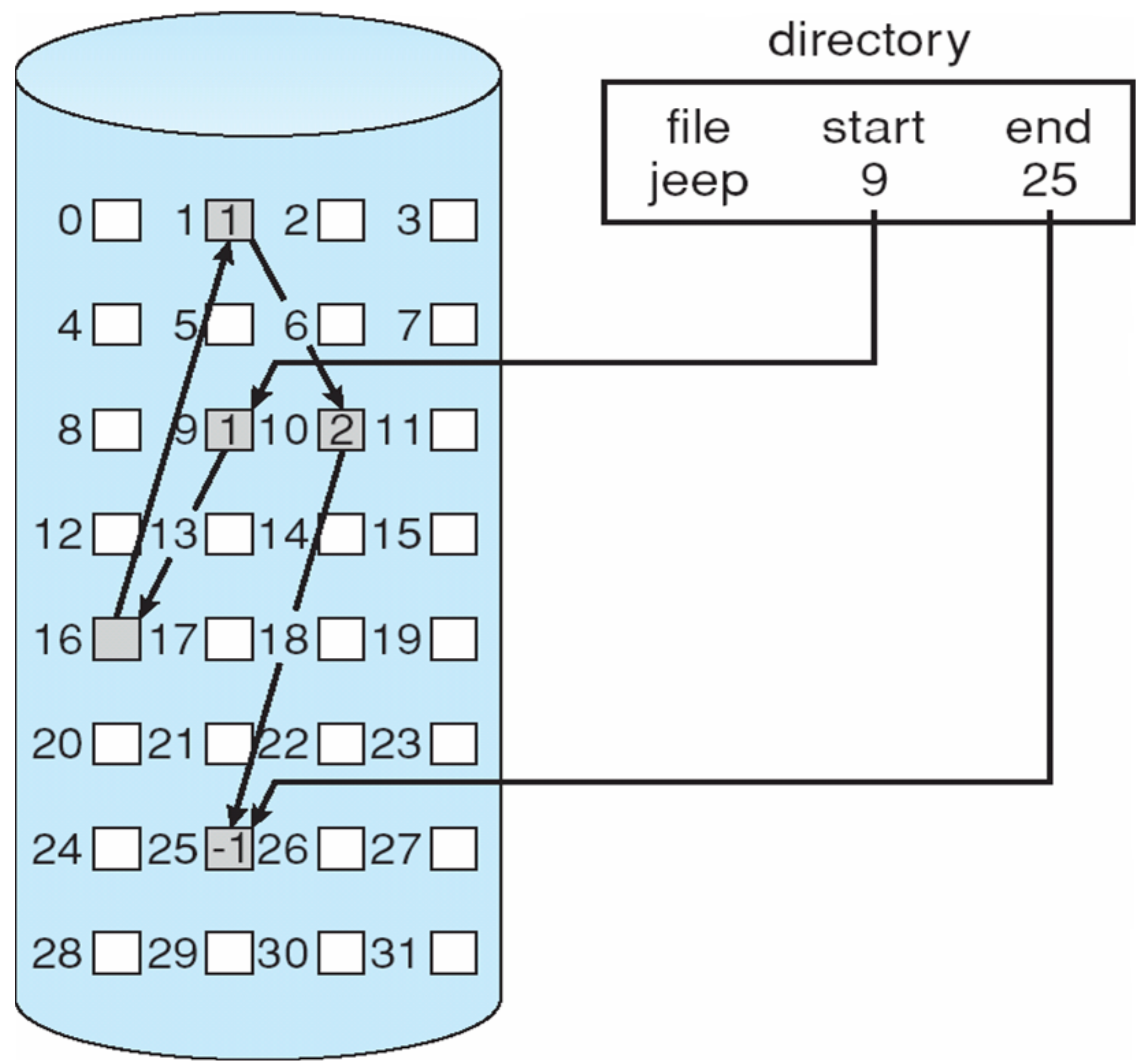
- Uses 30 bits to represent the length of a file
- Accommodates up to 1GB
- For DVD movies, 4 1-GB files may be necessary
- Called extents
- A good idea to use extents for file systems? — Veritas FS

Try 2: Linked List 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 List 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 and creates inconvenience of peculiar sizes per block
    - 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

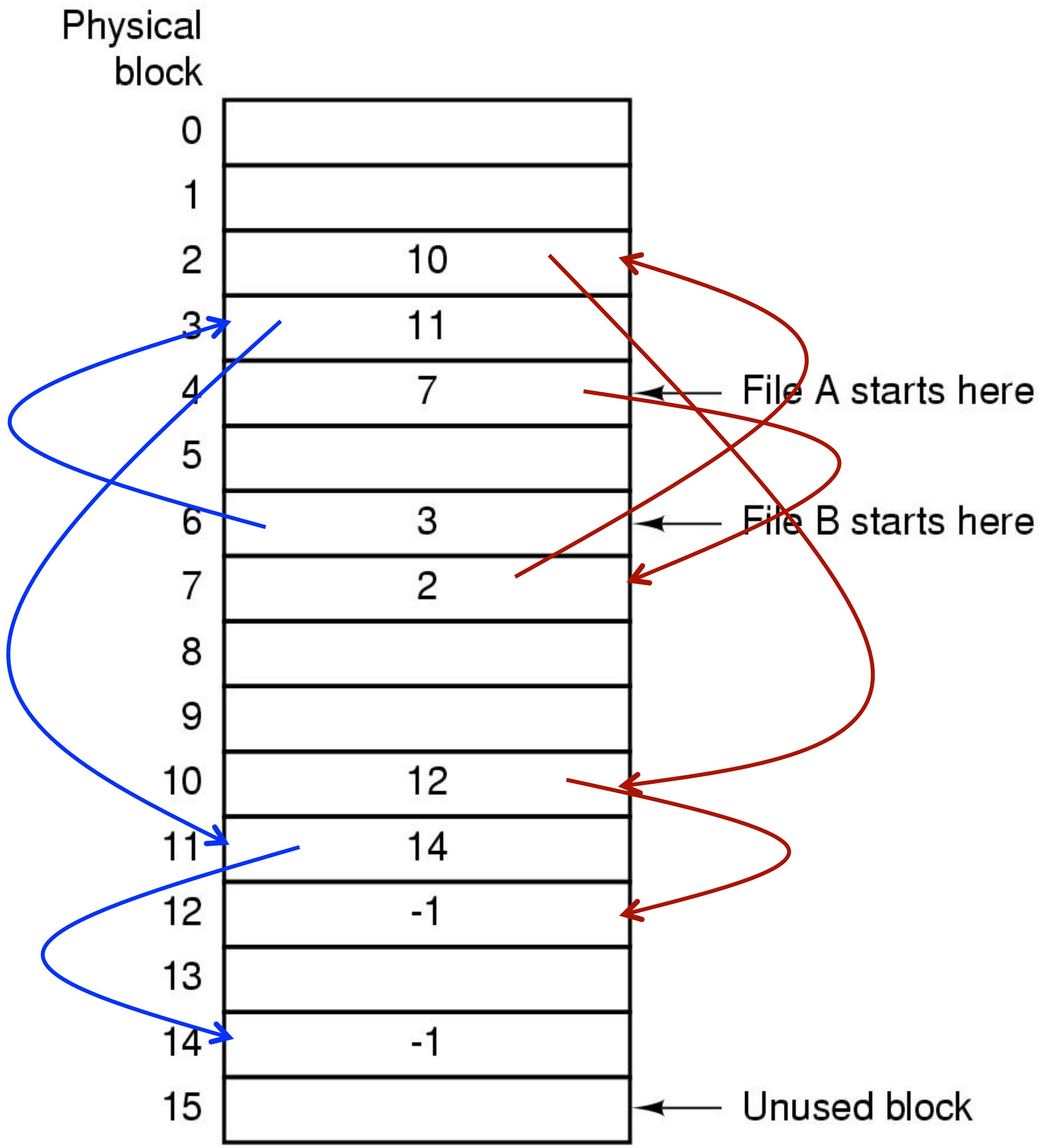
Each entry contains the address of the "next" block

- "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)



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!

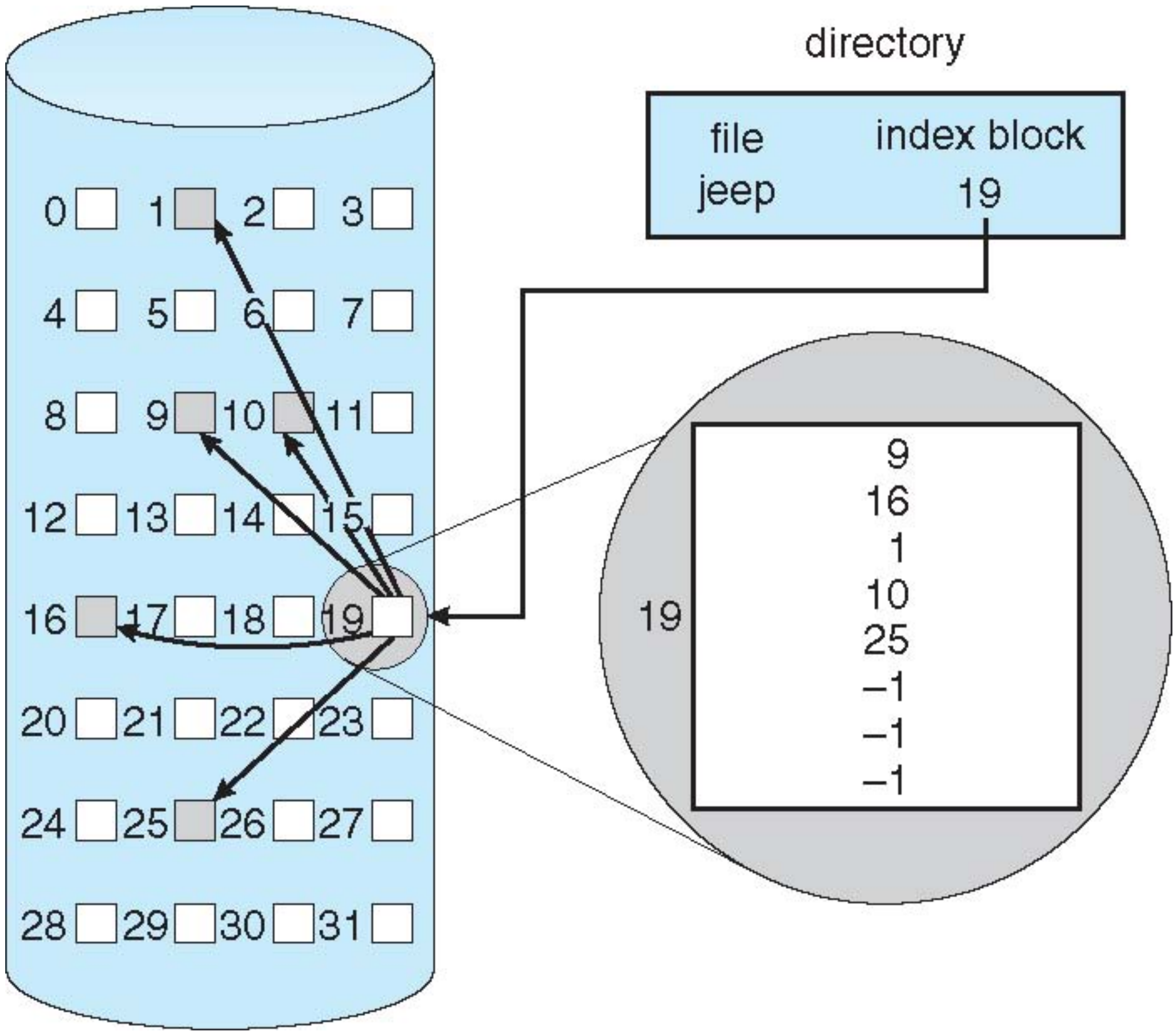
Try 3: Indexed Allocation

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

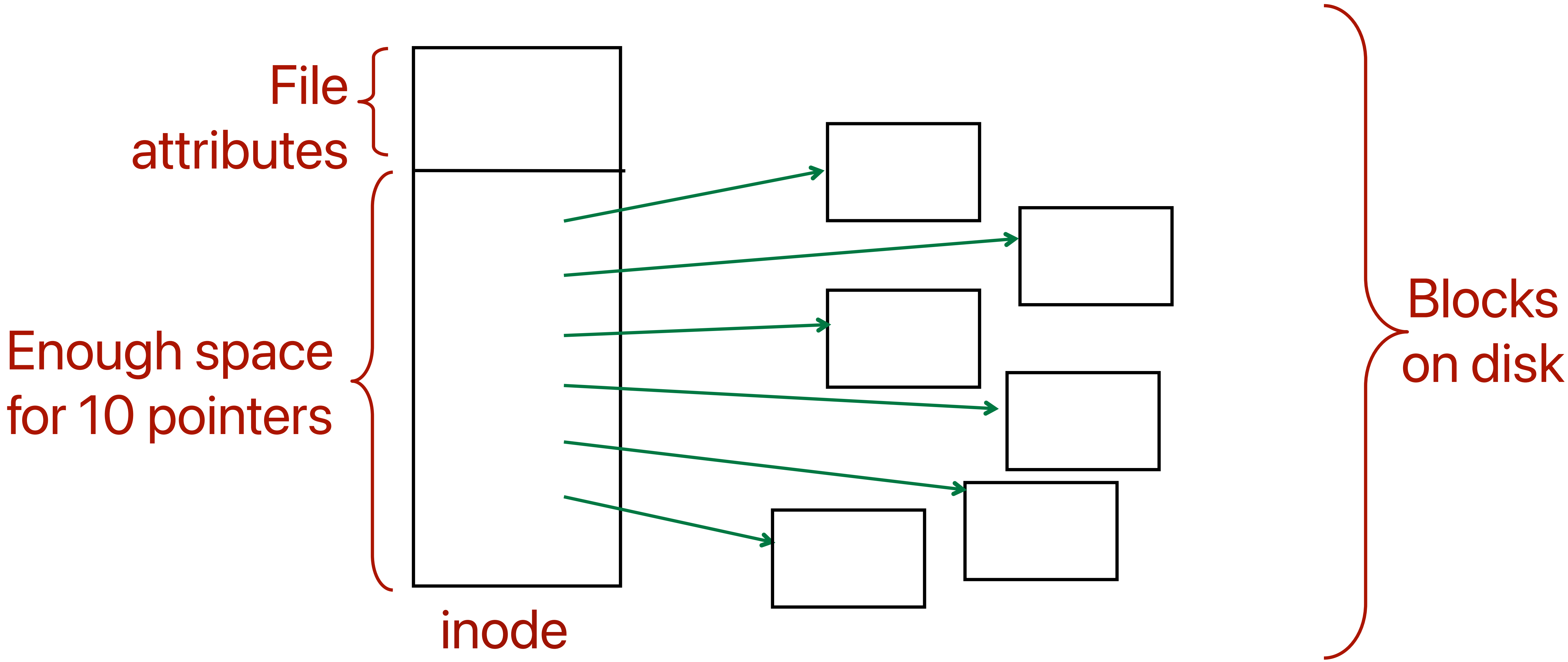


inodes in UNIX is the index block

Each inode ("index-node") contains

file attributes (permissions, timestamps, owner)

the index block

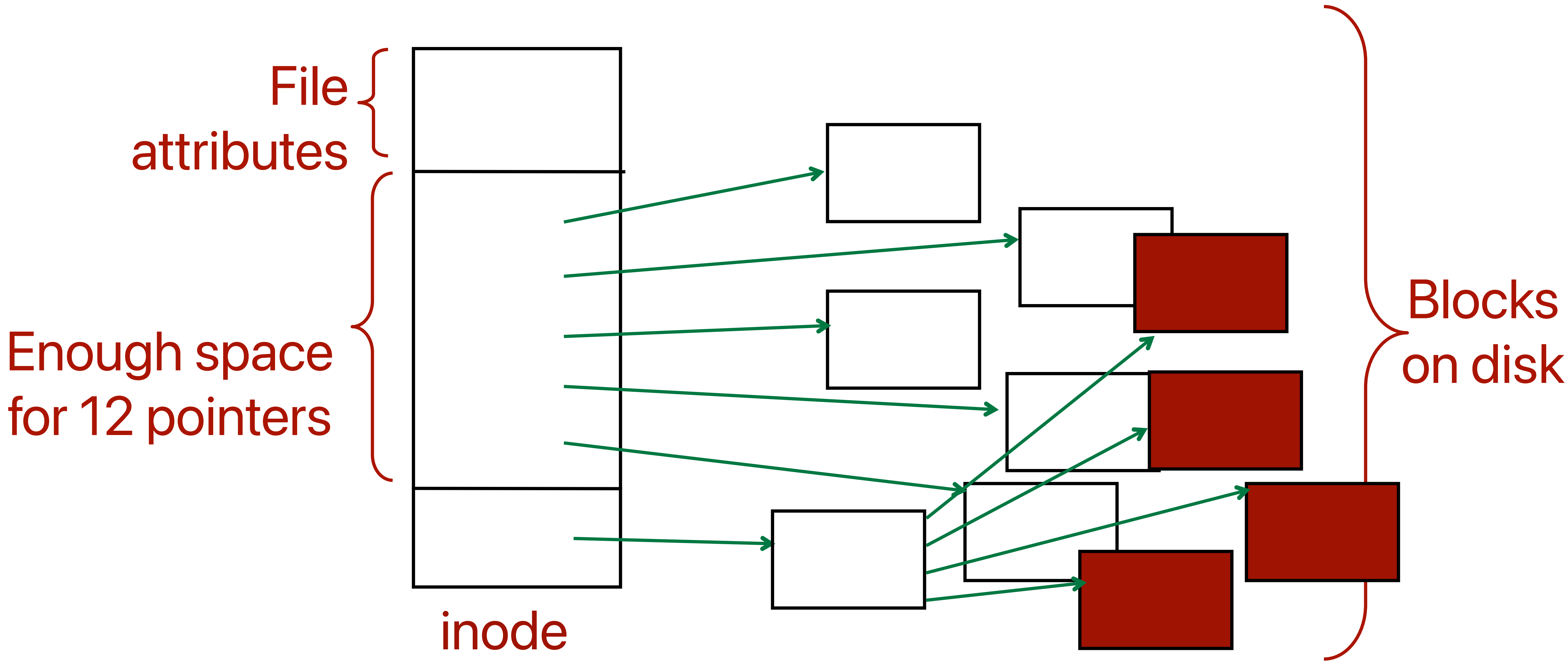


inodes in UNIX is the index block

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

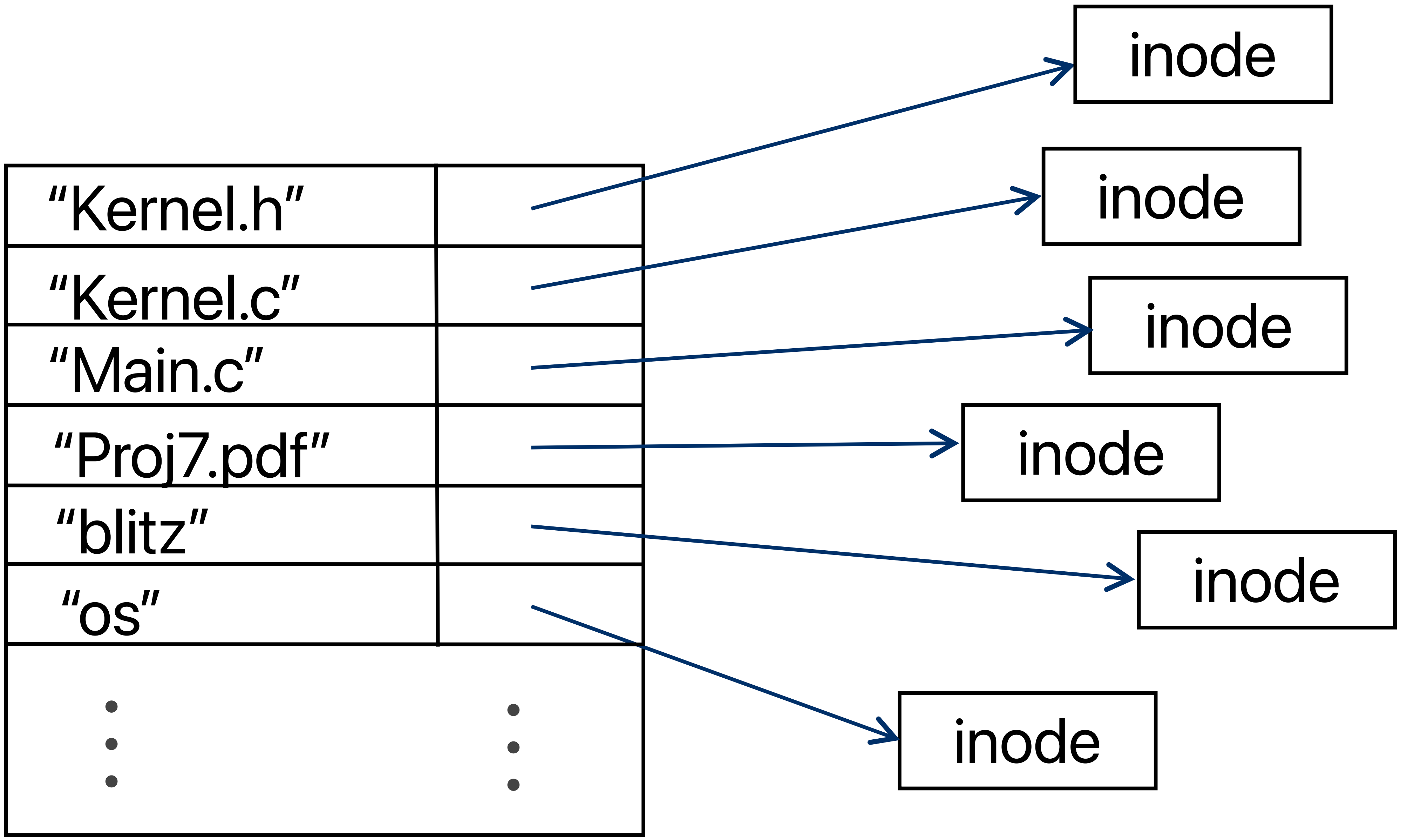
Solution: **multi-level indexing**



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

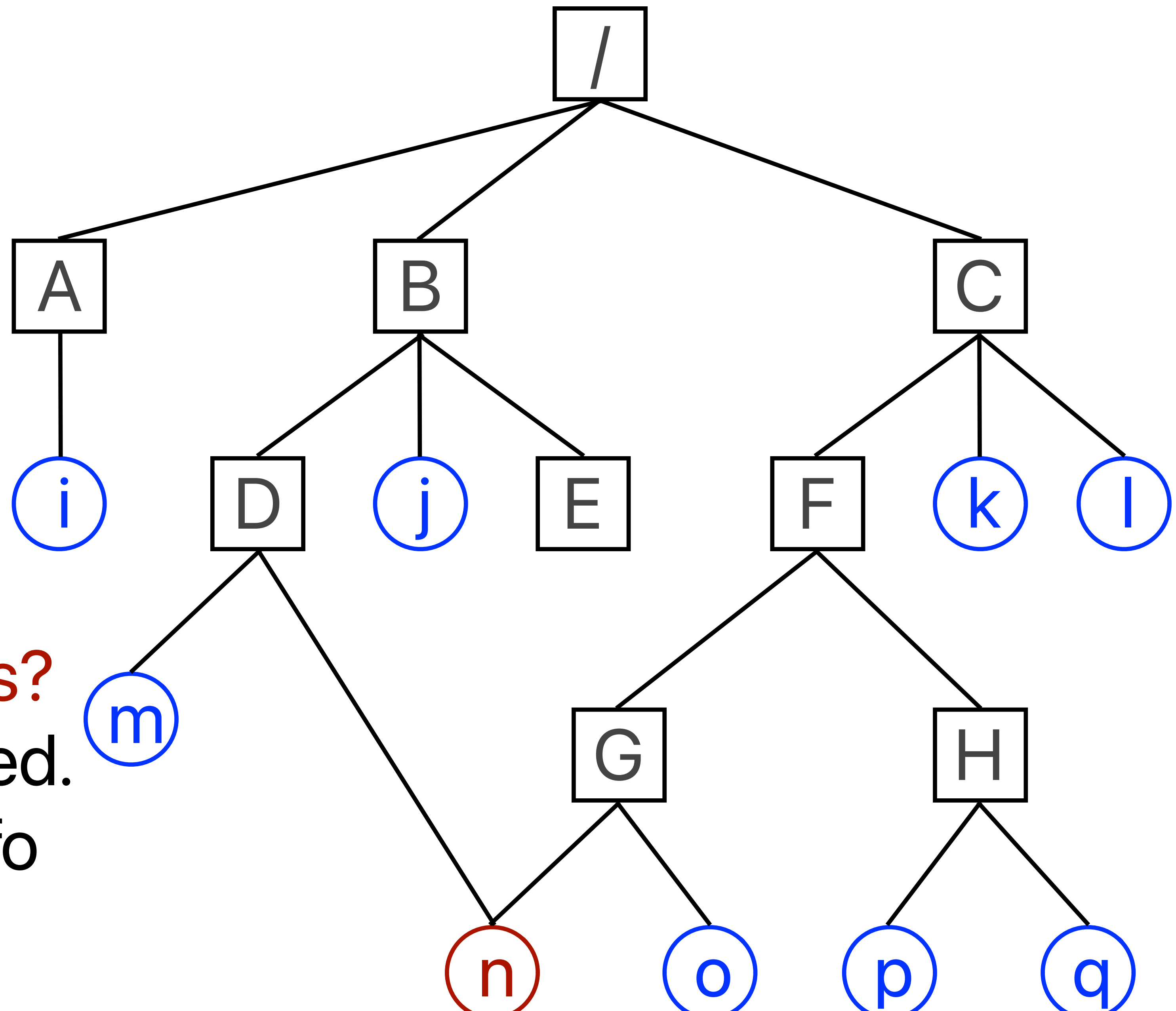
How do we improve its performance?

- Caching all results of previous searches
- Try to find a match for subsequent searches

Sharing Files

One file appears in several directories

Tree → 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

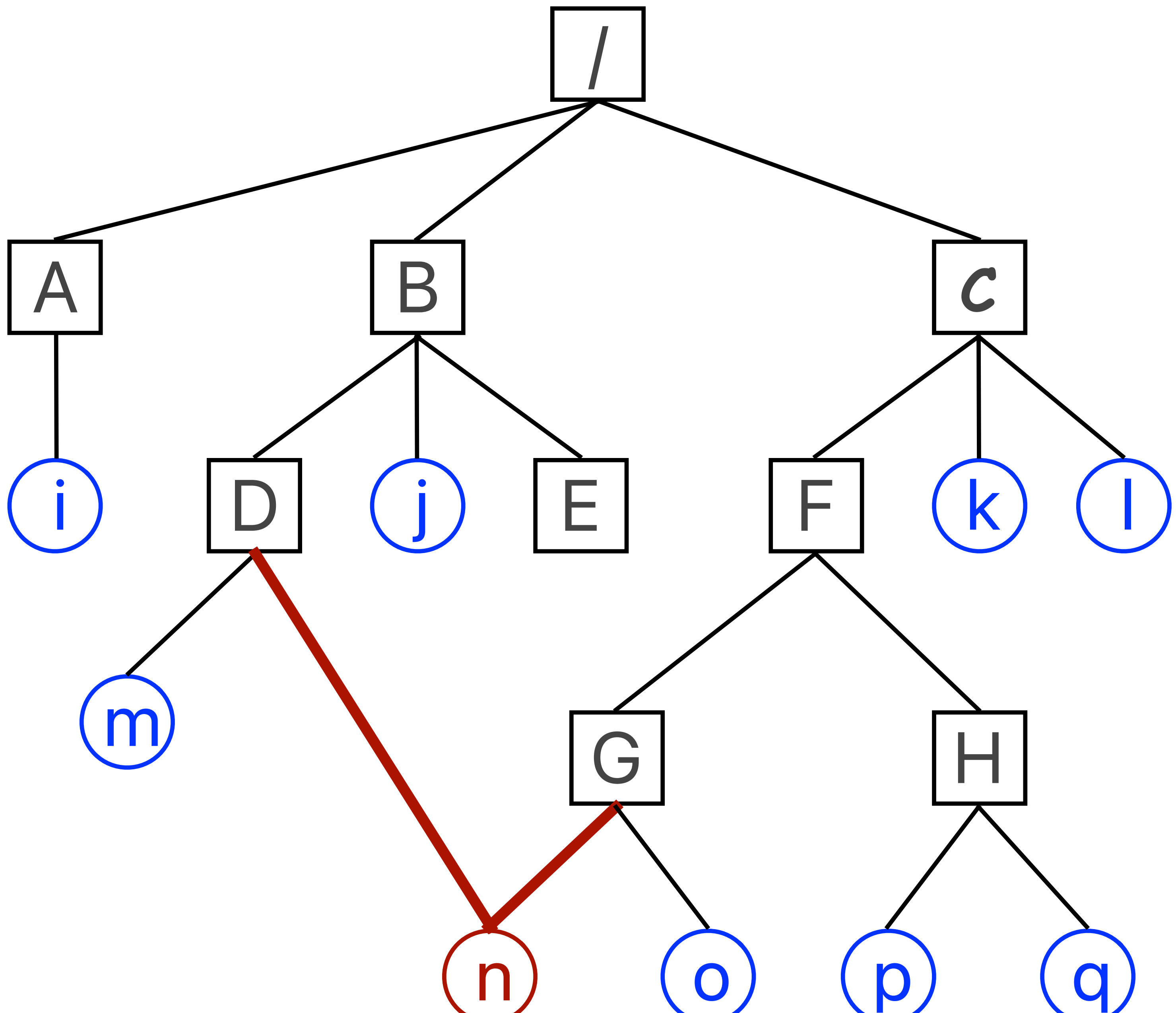
Assume inode number of “n” is 45

Directory “D”

“m”	123
“n”	45
⋮	⋮

Directory “G”

“n”	45
“o”	87
⋮	⋮



Hard Links

Assume inode number of “n” is 45

The file may have a different name in each directory

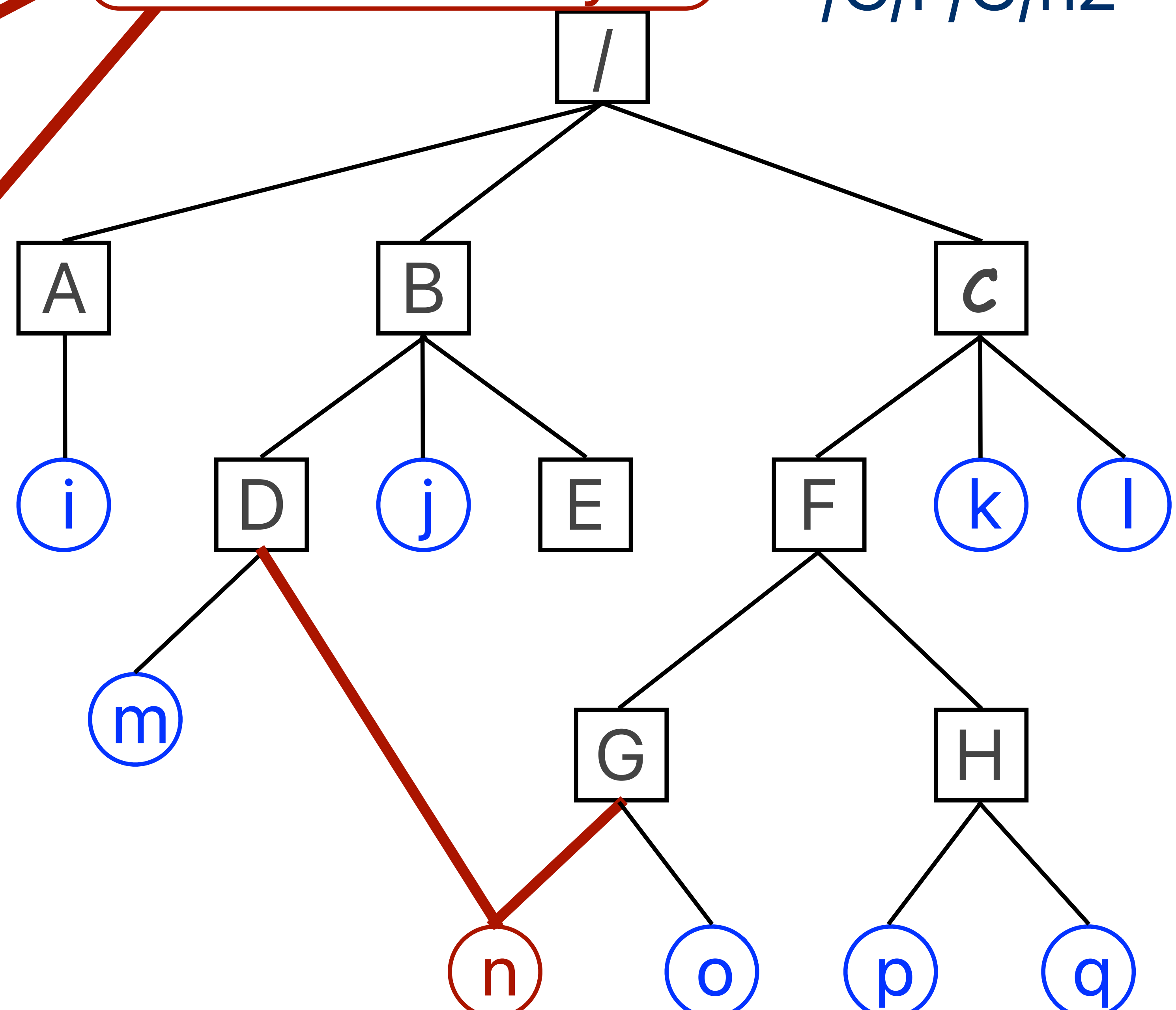
/B/D/n1
/C/F/G/n2

Directory “D”

“m”	123
“n1”	45
⋮	⋮

Directory “G”

“n2”	45
“o”	87
⋮	⋮



Symbolic Links

Assume inode number of "n" is 45

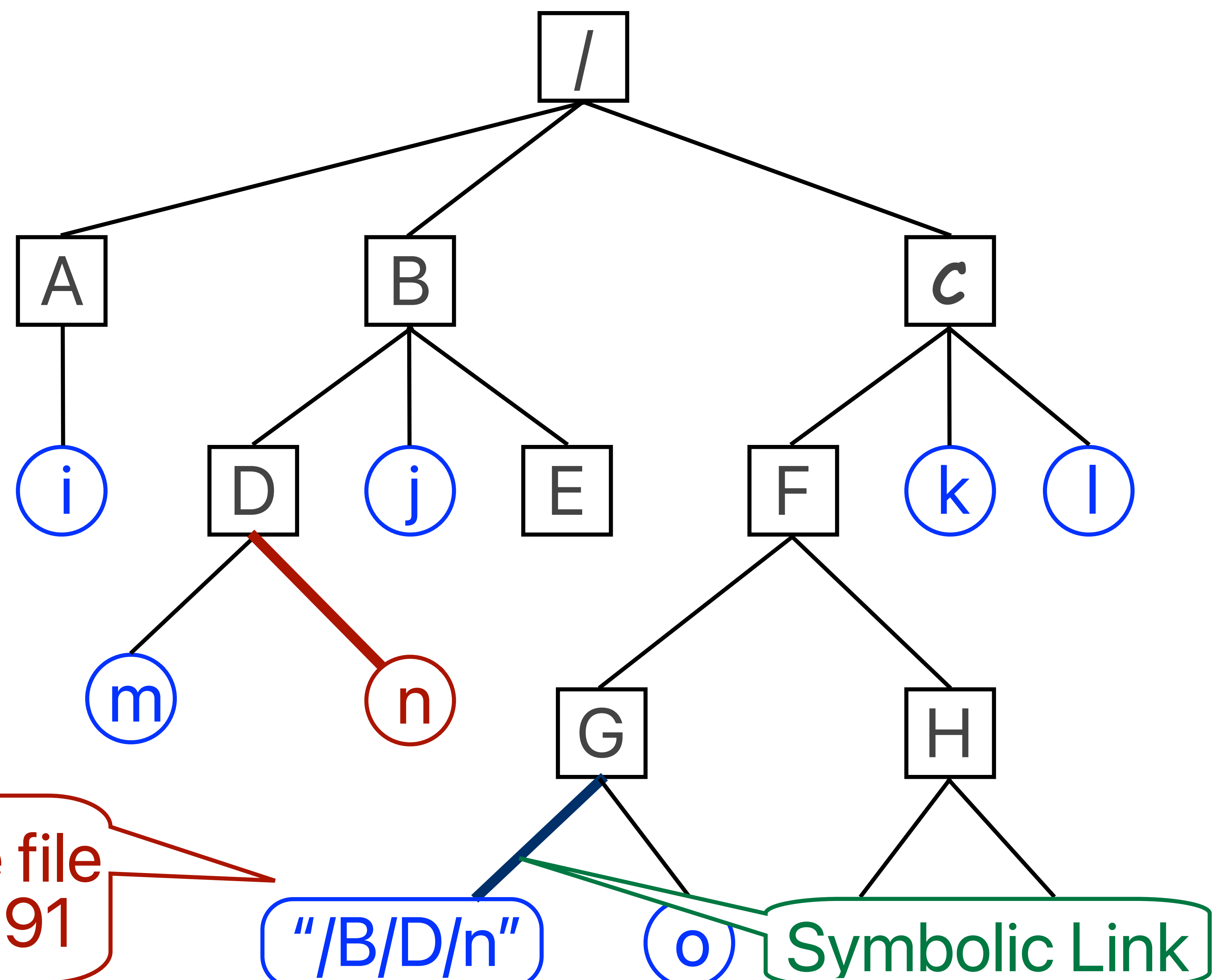
Directory "D"

"m"	123
"n"	45
⋮	⋮

Directory "G"

"n"	91
"o"	87
⋮	

Separate file inode = 91



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

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

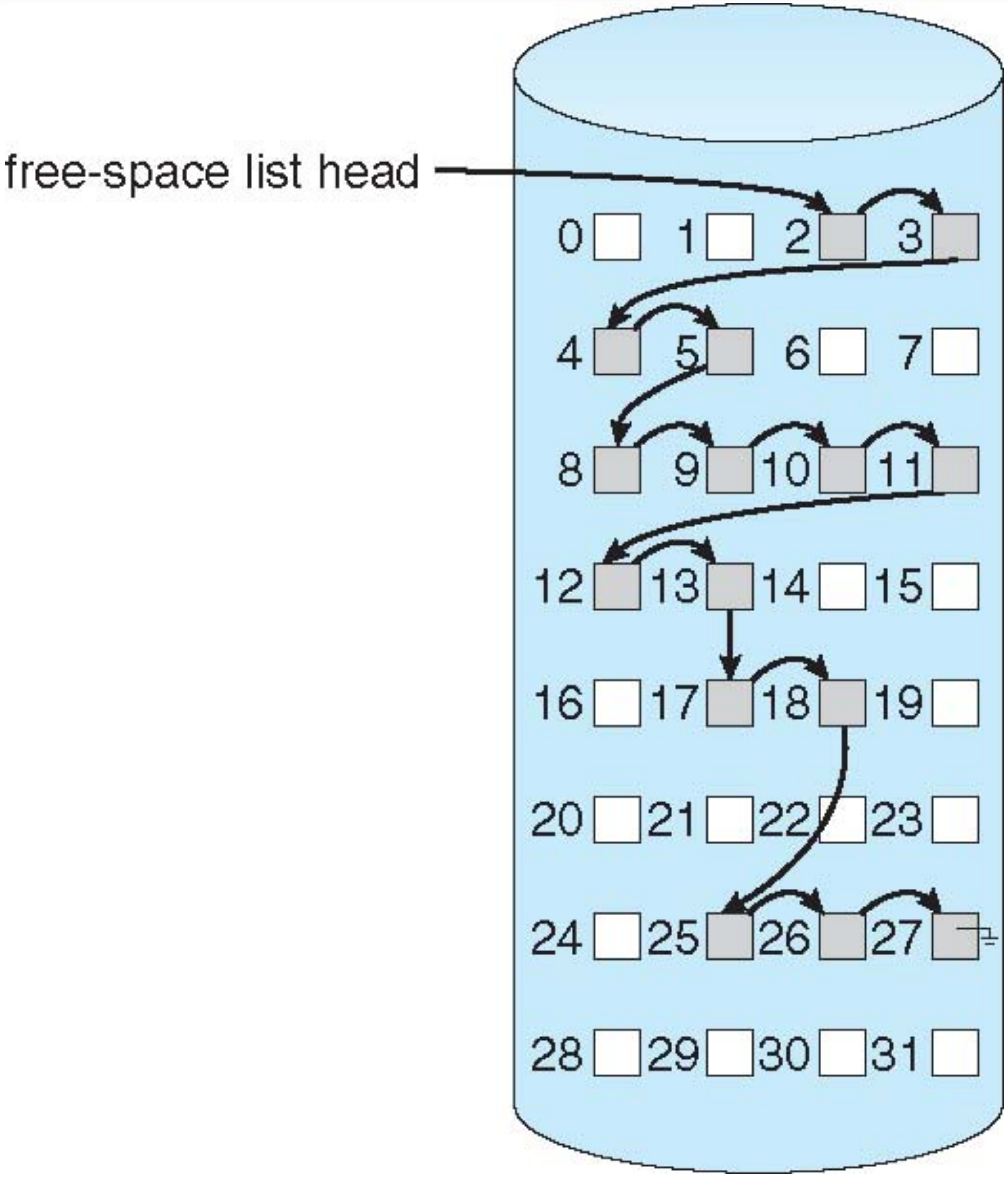
Use a disk block to contain all the free block numbers

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)

What we've covered so far

Three Easy Pieces: Chapter 37.1-37.3 (Hard Disk Drives), Chapter 39 (Files and Directories), 40 (File System Implementation)