Chapter 11: File System Implementation

Prof. Li-Pin Chang

National Chiao Tung University

Chapter 11: File System Implementation

- File-System Structure
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems

Objectives

- To describe the details of implementing local file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers

Layered File System

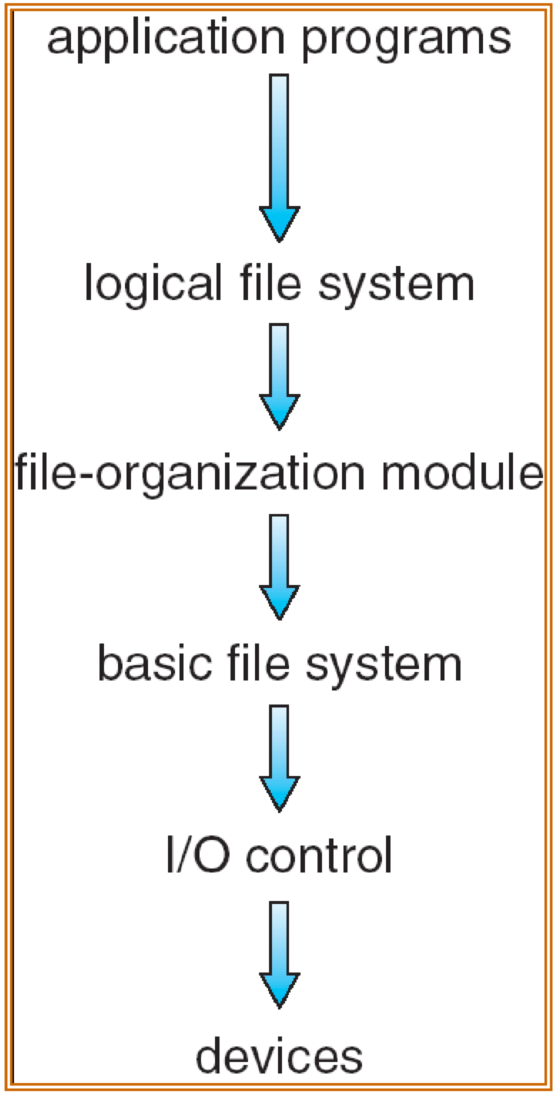
fread() / fwrite()

fs->read, fs->write

Read write to
Buffer/cache

Disk read/disk write

Control signals



Applications

File-system interface

File-system
implementation

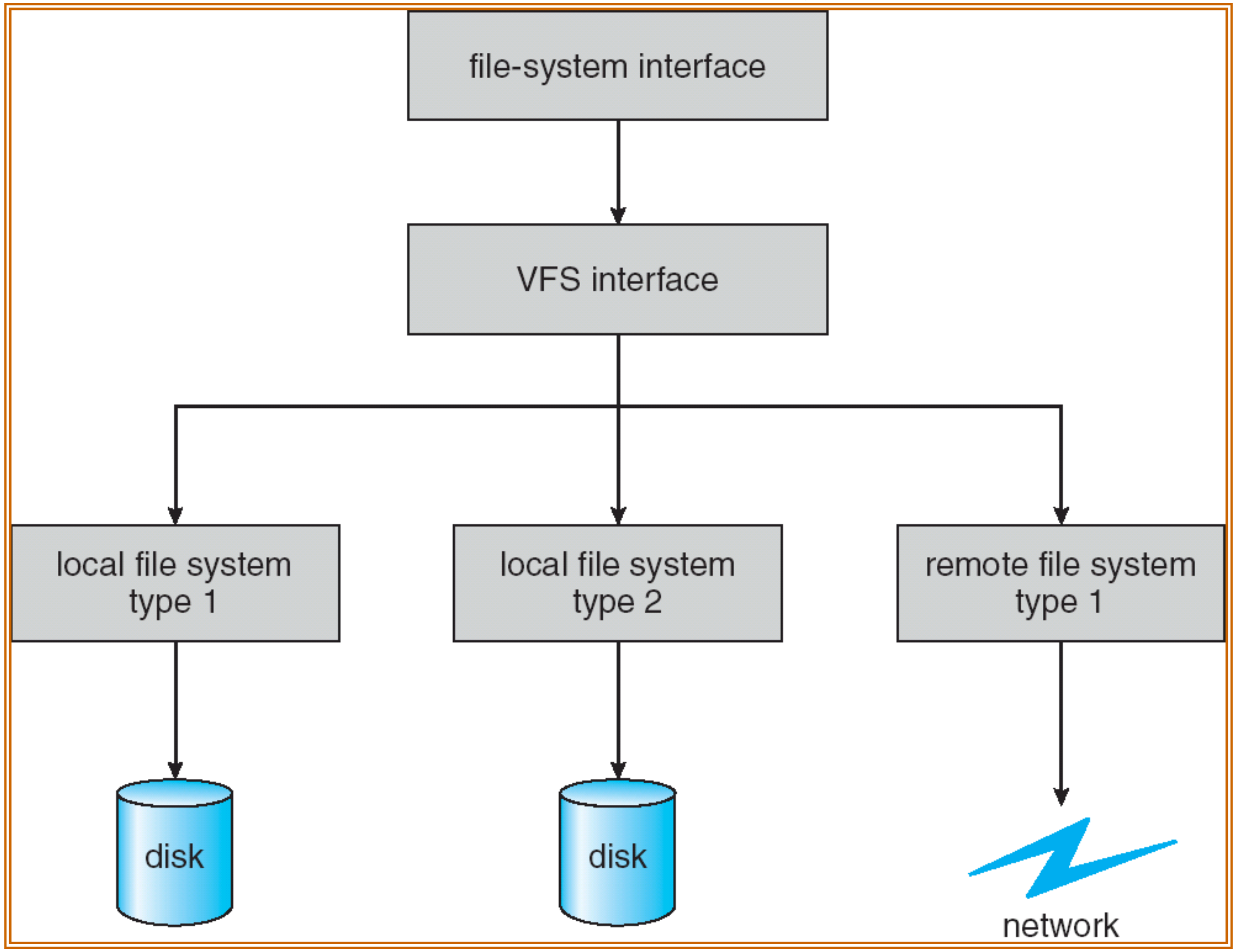
Disk cache

Disk scheduler

Disk driver

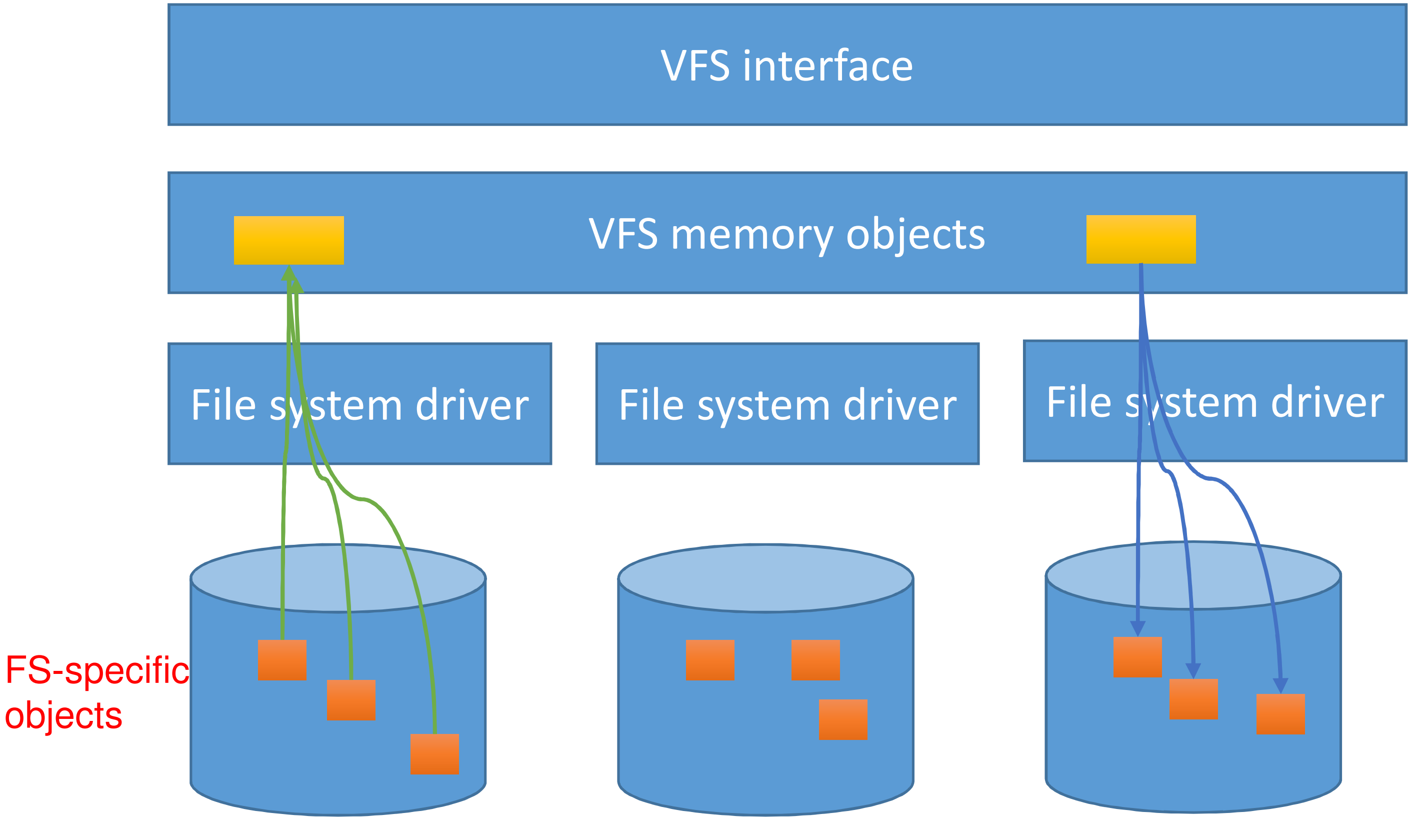
Disk

Schematic View of Virtual File System



Linux Virtual File System Architecture

- File system drivers fills VFS objects with information in their disk data structures

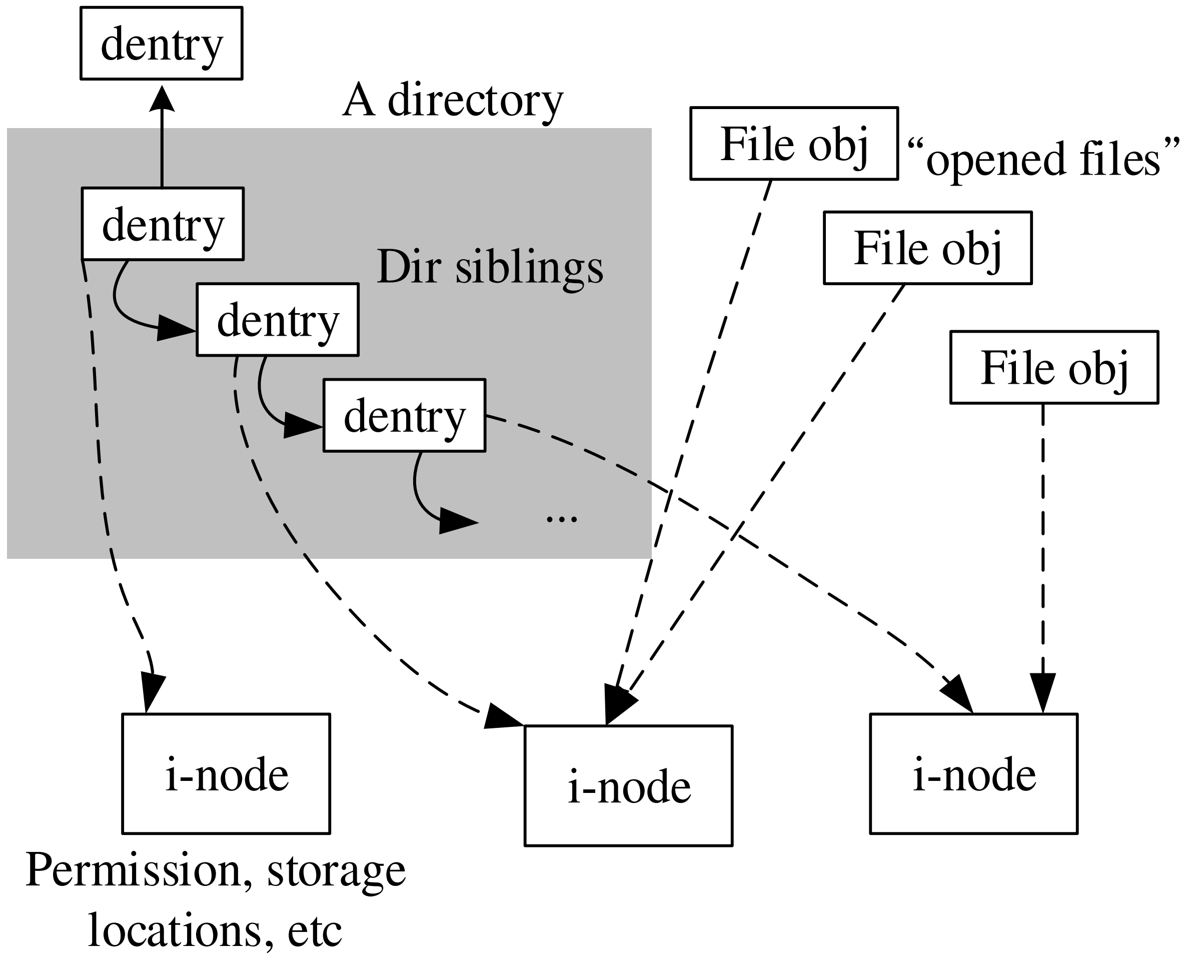


In-memory Objects in Linux VFS

- Inode
  - Uniquely represent an individual file
- File object
  - Represent an opened file, one for each fopen instance
- Superblock
  - Represent the entire filesystem
- Dentry object
  - Represent an individual directory entry

- A collection of operations are defined on each type of object

In-memory objects of Linux VFS

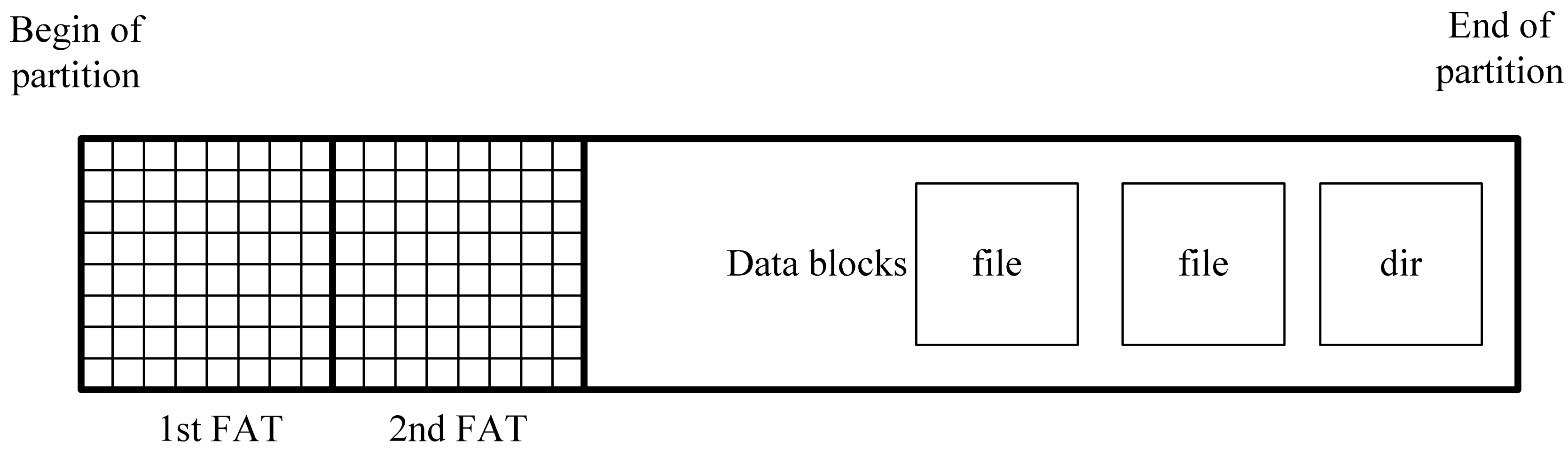


On-disk data structures

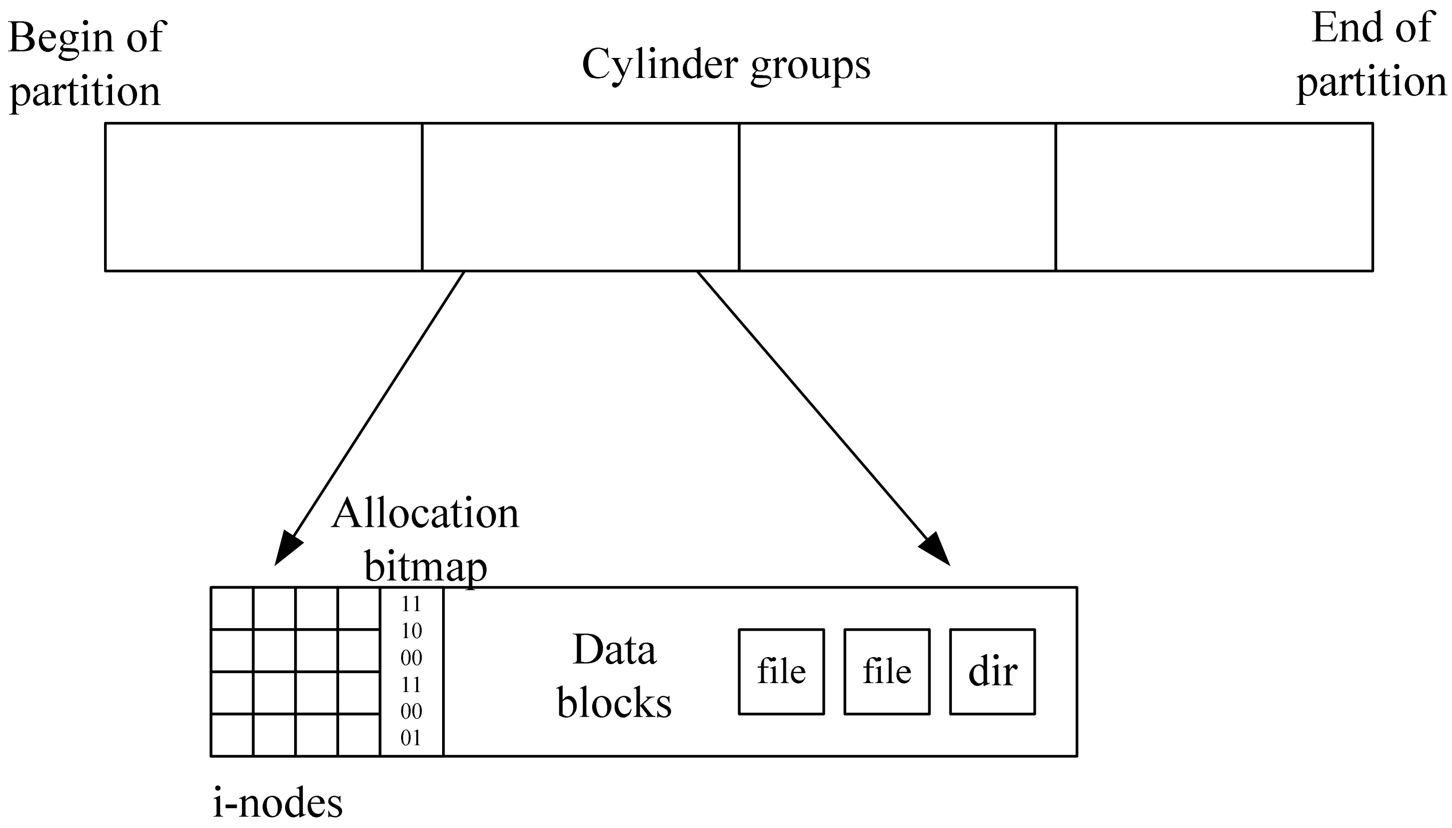
- File-system-specific
- Linux ext file system
  - Super block
  - Inode
  - Allocation bitmaps
- Microsoft FAT file system
  - File allocation tables
  - Directory

- File system driver must fill the in-memory objects with the information from the on-disk data structures
  - For example, FAT file system does not have on-disk i-nodes

Disk layout of FAT 12/16/32 file systems



Disk Layout of the Linux ext 2/3/4 file systems



- Directory implementation
- Allocation (index) methods
- Free-space management

Directory Implementation

- Linear list of file names with pointer to the data blocks.
  - simple design
  - time-consuming operations
  - FAT file system
- B-trees (or variants)
  - Efficient search, variable size
  - XFS, NTFS, ext4
  - For very large file systems

Example: Directory Dump in FAT

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000167936	41	6D	00	79	00	64	00	69	00	72	00	0F	00	E6	32	00	Am.y.d.i.r....2.
000167952	00	00	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	....
000167968	4D	59	44	49	52	32	20	20	20	20	20	10	00	00	90	B1	MYDIR2 .....
000167984	A6	42	A6	42	00	00	90	B1	A6	42	04	00	00	00	00	00	.B.B.....B......
000168000	41	6D	00	79	00	64	00	69	00	72	00	0F	00	DE	31	00	Am.y.d.i.r....1.
000168016	00	00	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	....
000168032	4D	59	44	49	52	31	20	20	20	20	20	10	00	64	6A	B1	MYDIR1 ..dj.
000168048	A6	42	A6	42	00	00	6A	B1	A6	42	03	00	00	00	00	00	.B.B..j..B......
000168064	41	6D	00	79	00	66	00	69	00	6C	00	0F	00	8B	65	00	Am.y.f.i.l....e.
000168080	31	00	2E	00	74	00	78	00	74	00	00	00	00	00	FF	FF	1...t.x.t.......
000168096	4D	59	46	49	4C	45	31	20	54	58	54	20	00	64	99	B1	MYFILE1 TXT .d..
000168112	A6	42	A6	42	00	00	99	B1	A6	42	05	00	0F	00	00	00	.B.B.....B......
000168128	E5	6D	00	79	00	66	00	69	00	6C	00	0F	00	5B	65	00	.m.y.f.i.l...[e.
000168144	32	00	2E	00	74	00	78	00	74	00	00	00	00	00	FF	FF	2...t.x.t.......
000168160	E5	59	46	49	4C	45	32	20	54	58	54	20	00	64	77	8B	.YFILE2 TXT .dw.
000168176	A7	42	A6	42	00	00	77	8B	A7	42	07	00	22	20	09	00	.B.B..w..B.." ..
000168192	41	6C	00	64	00	65	00	5F	00	32	00	0F	00	5D	36	00	Al.d.e._.2...]6.
000168208	31	00	2E	00	74	00	67	00	7A	00	00	00	00	00	FF	FF	1...t.g.z.......
000168224	4C	44	45	5F	32	36	31	20	54	47	5A	20	00	64	77	8B	LDE_261 TGZ .dw.
000168240	A7	42	A6	42	00	00	77	8B	A7	42	07	00	22	20	09	00	.B.B..w..B.." ..

Allocation/Index Methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Files cannot grow beyond the allocated space
- Efficient access
  - file offset can be directly translated into sector block # without extra disk access
  - Always sequential disk read/write
- Wasteful of space (dynamic storage-allocation problem)
  - File deletion leaves holes (external fragmentation) in the file system
  - Needs compaction, maybe done in background or downtime

Contiguous Allocation

- Mapping from logical to physical

Q

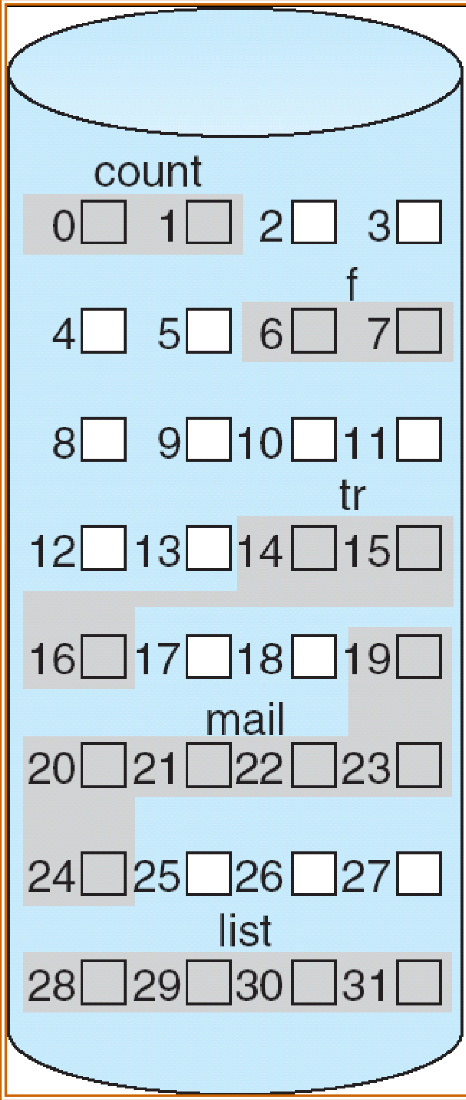
LA/512

R

Block to be accessed = Q + starting address (block)
Displacement into block = R

LA=byte address
1 block=512B

Contiguous Allocation of Disk Space



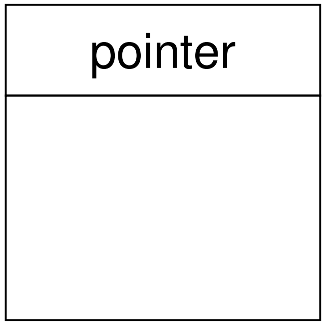
directory

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

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk

block =



Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system
  - no waste of space (no external fragmentation)
  - No random access (need to traverse the linked blocks)
- Mapping

Q

LA/511

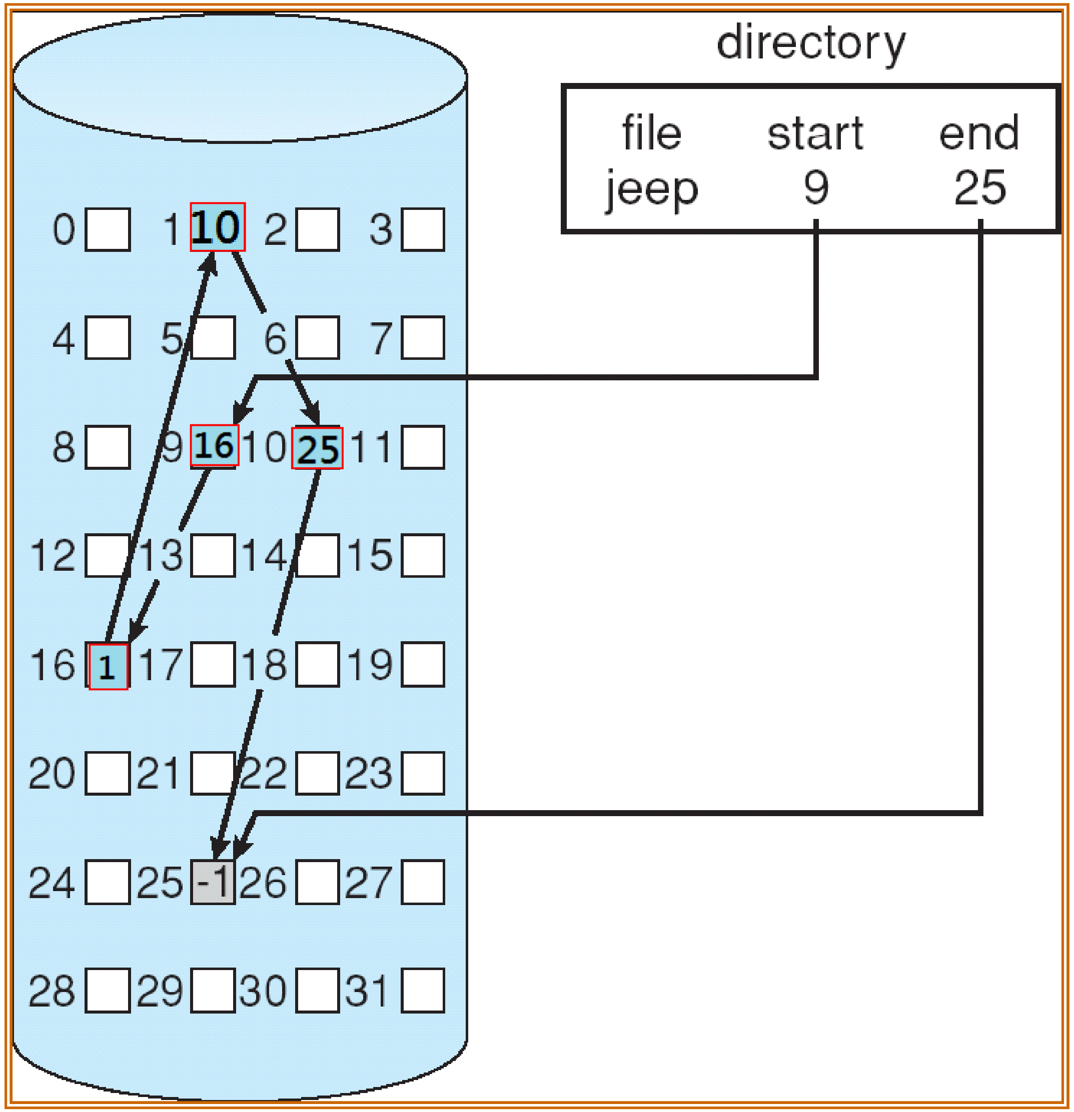
R

1 data block =512B, 1 for ptr,
So 511B for user data

Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1 (the 0^{th} byte is for pointer)

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.

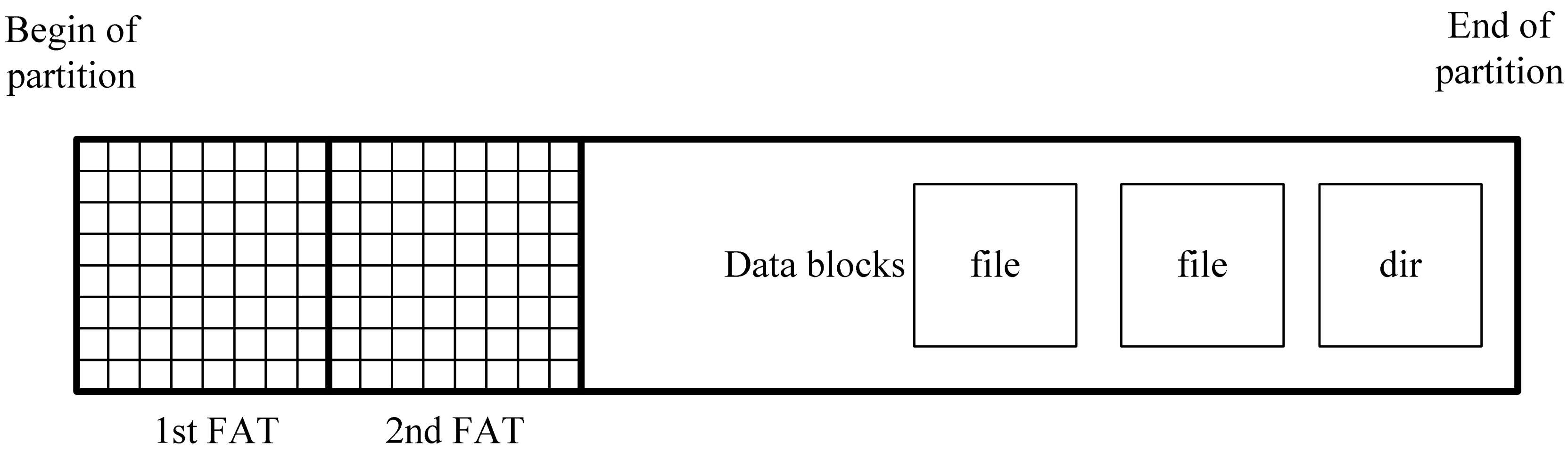
Linked Allocation



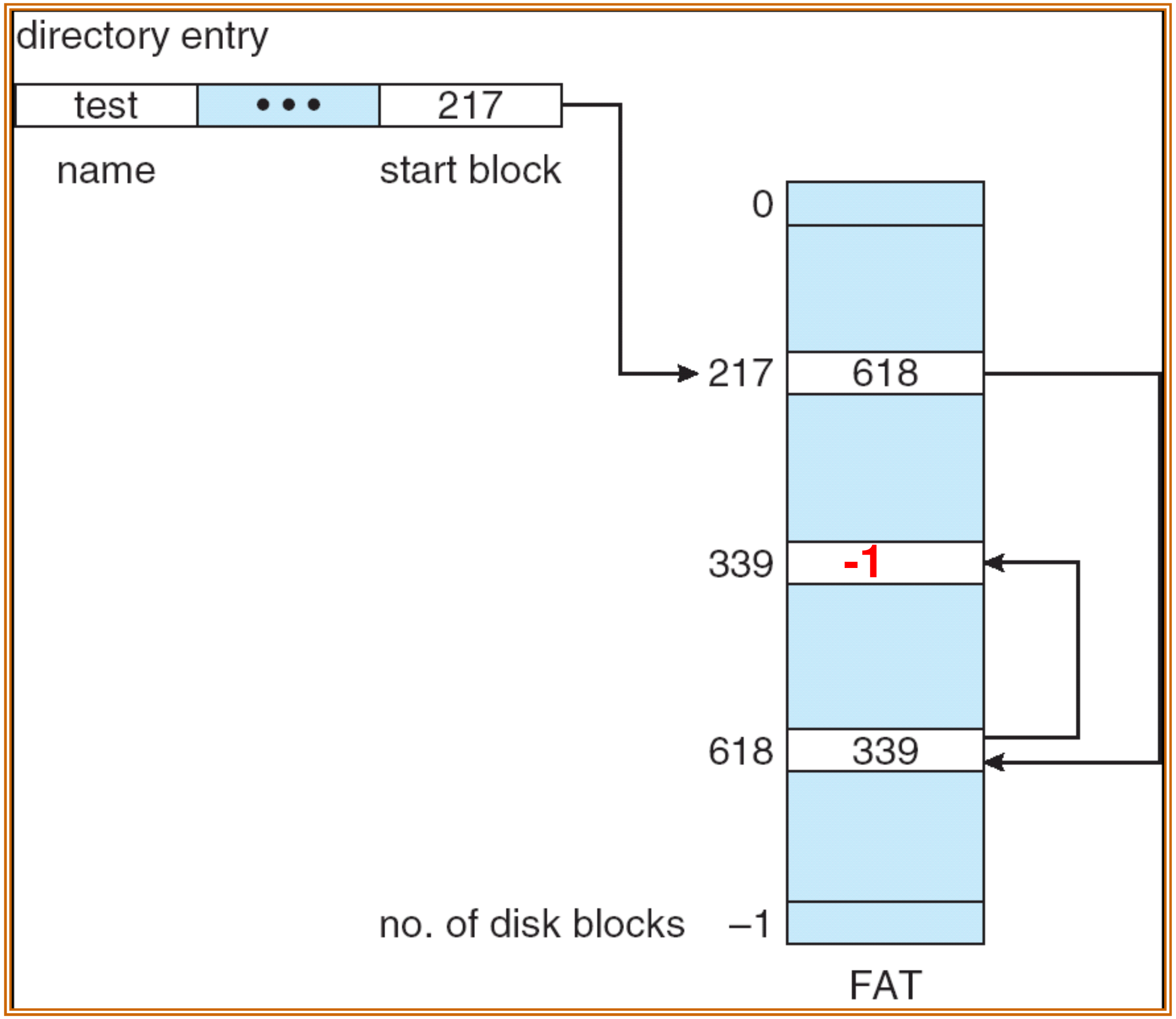
Linked Allocation

- Separating the pointers from data blocks
  - Make data size a power of 2
- Example: FAT file system

The layout of FAT 12/16/32 file system



File-Allocation Table



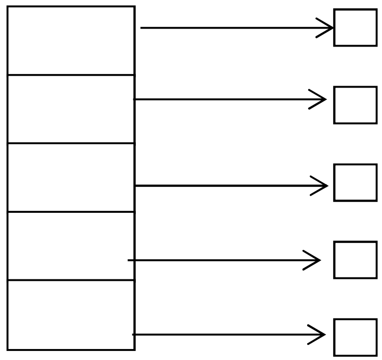
- A bad list maintains bad clusters
- Scans 0 for unallocated clusters

File-Allocation Table

- Space is allocated in terms of “cluster”
  - Given: volume size = 2^x bytes
  - Use FAT 16, total 2^16 clusters
  - 1 cluster = 2^(x-16) bytes
- If the FAT is not cached, accessing every next data block involves a lookup to the FAT
  - FAT will be a read/write bottleneck
- FAT table can also be a single –point-of-failure
  - Two identical copies to reduce risk

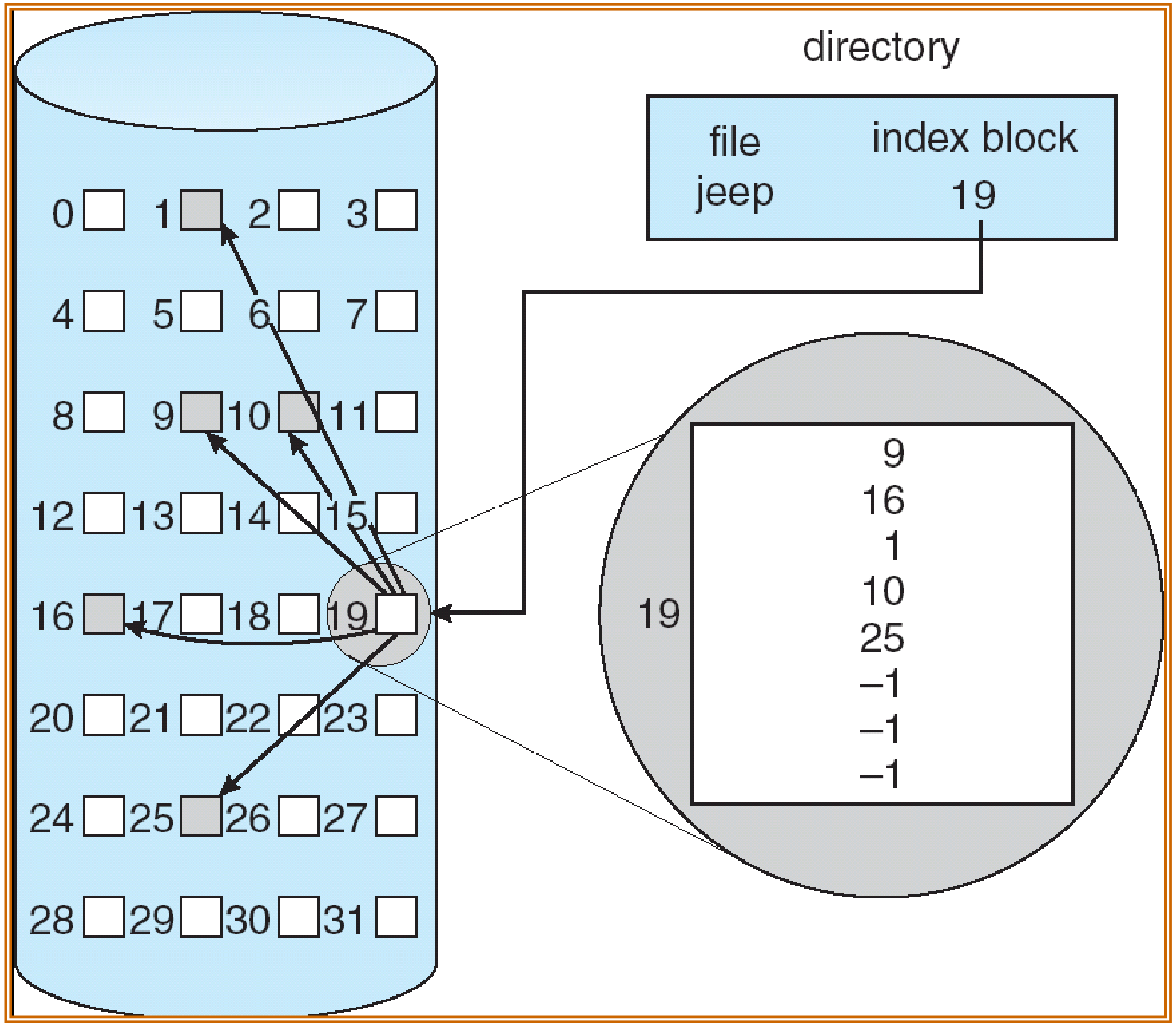
Indexed Allocation

- Brings all pointers together into the index block.
- Logical view.



index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

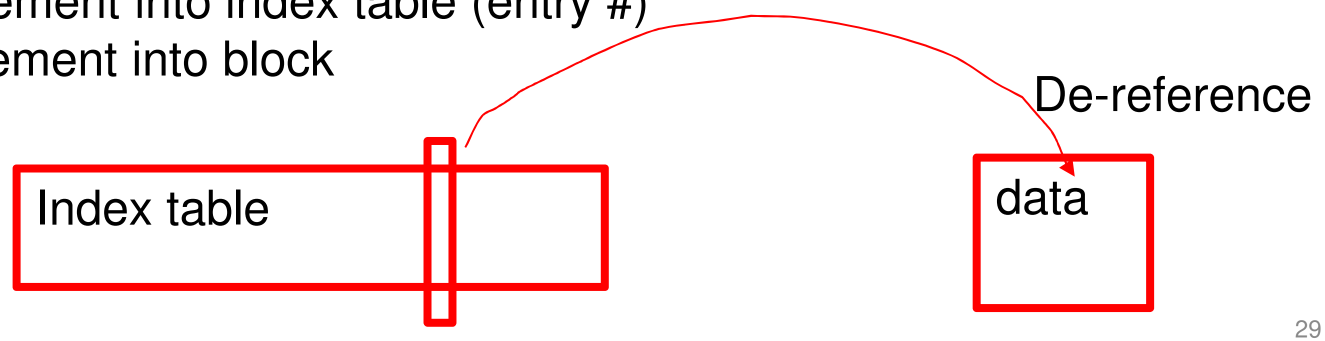
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.

LA/512 → Q, R

LA=byte address

A block is of 512B

Q = displacement into index table (entry #)
R = displacement into block

- 

Indexed Allocation – Mapping

- Two-level index (maximum file size is 512^3)

$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

Q_1 = displacement into outer-index
R_1 is used as follows:

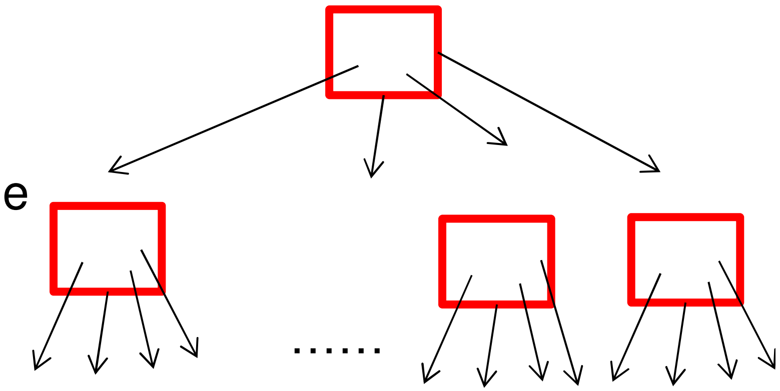
$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of index table
R_2 displacement into block of file:

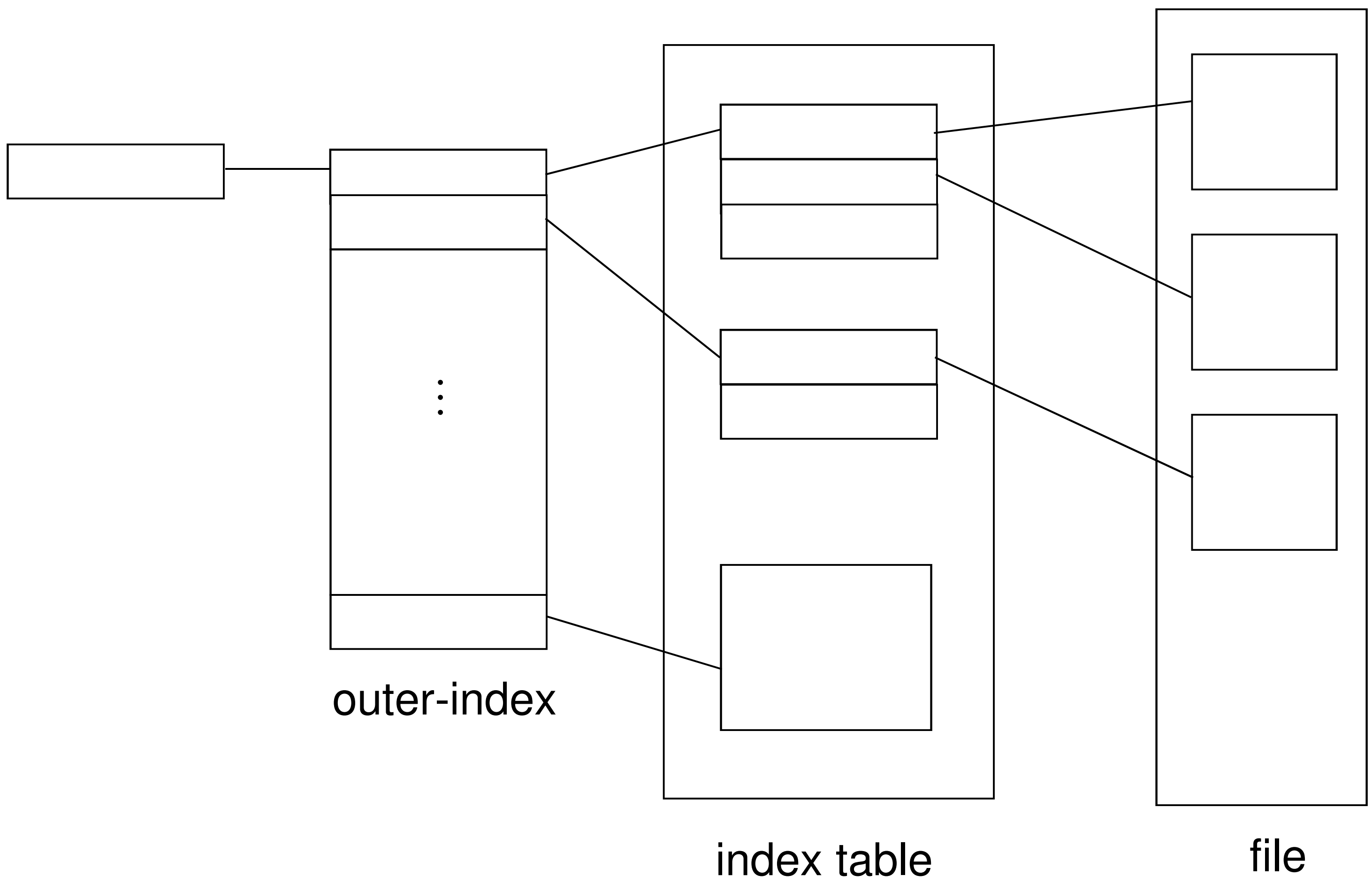
- 1^{st} level: pointers to index tables
- 2^{nd} level: pointers to data blocks

- 1block=512B, 1ptr=1B
- 1 index block has 512 ptr,

- Total addressable size=512*512*512 bytes

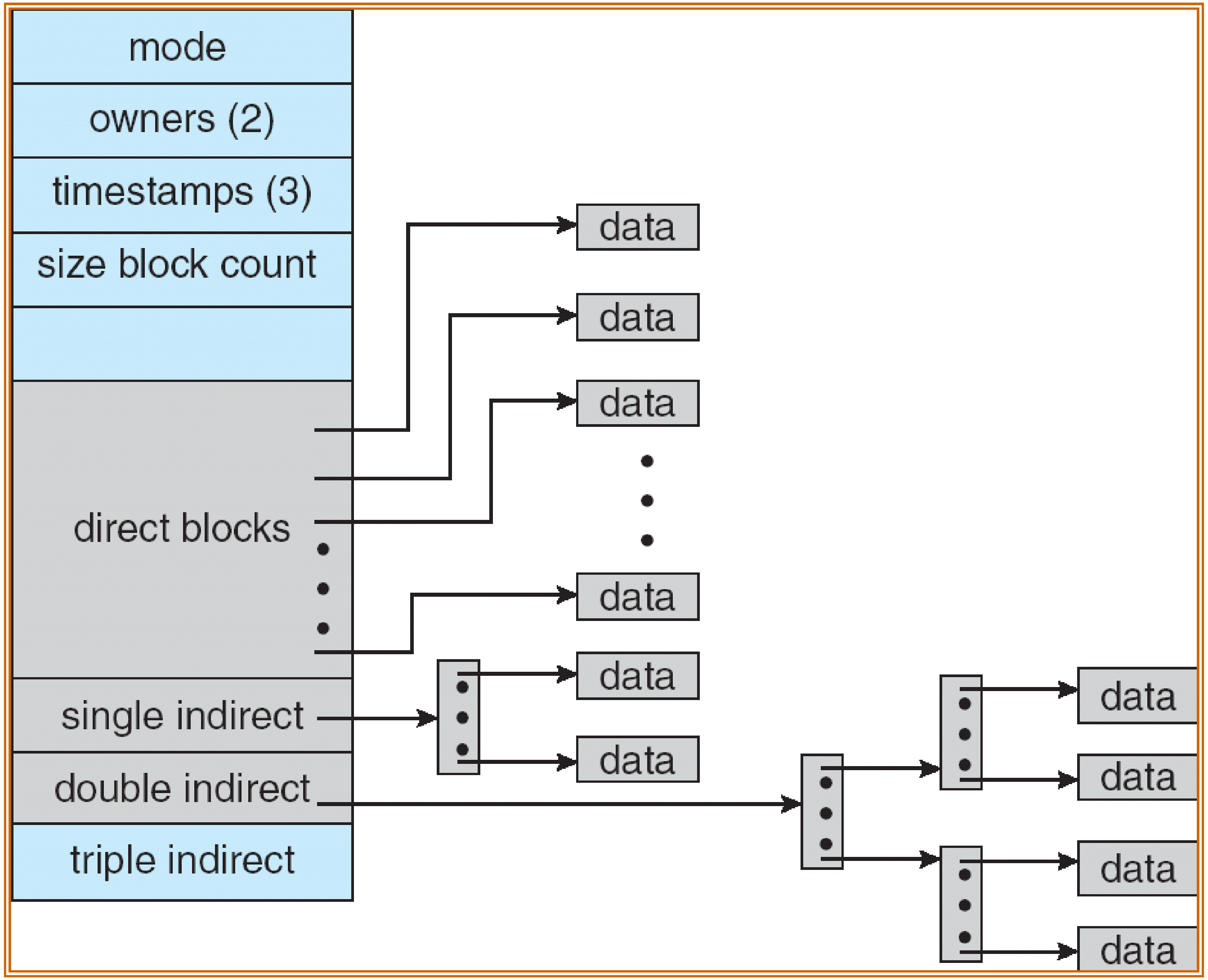


Indexed Allocation – Mapping (Cont.)



UNIX inode

An i-node. Small files use only direct blocks



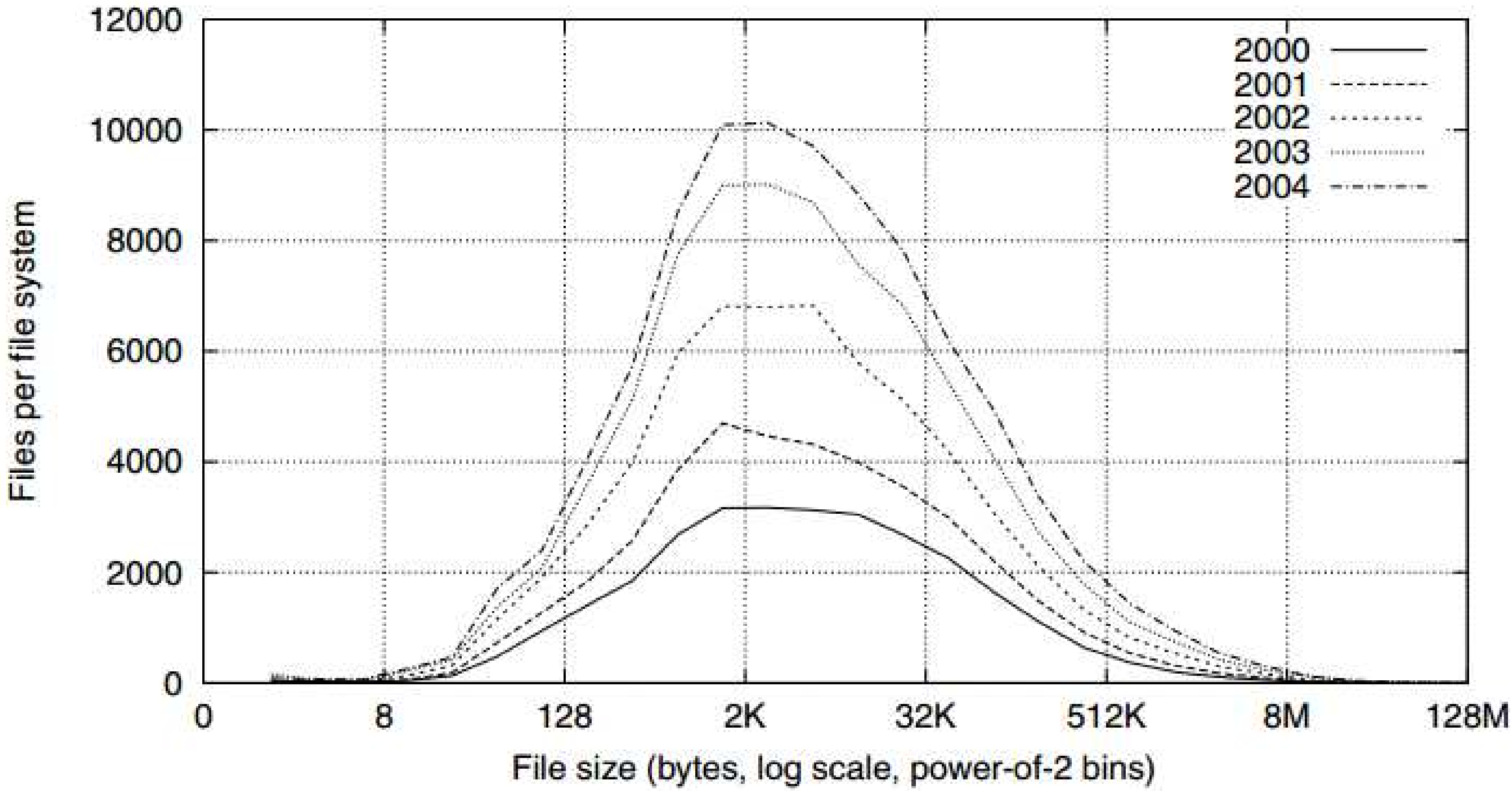
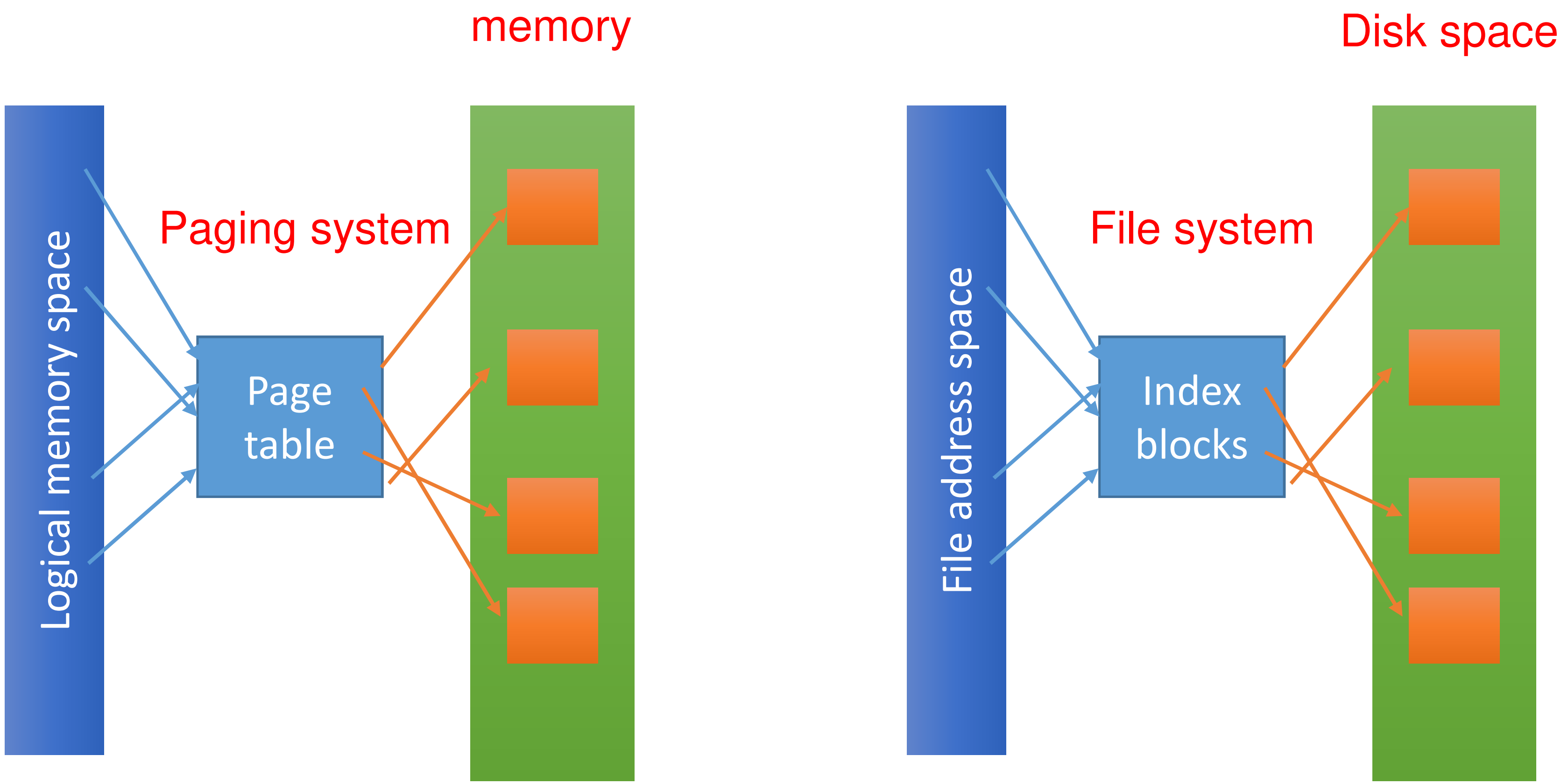


Fig. 2. Histograms of files by size.

A. Agrawal, "A Five-Year Study of File-System Metadata"

Indirection, indirection, indirection …



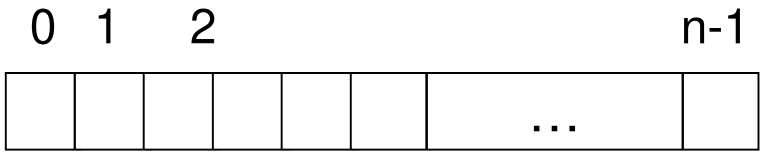
"All problems in computer science can be solved by another level of indirection" -- David Wheeler

Extent-Based Systems

- A hybrid of contiguous allocation and linked/indexed allocation
- Extent-based file systems allocate disk blocks in extents
- An extent is a set of contiguous disk blocks
  - Extents are allocated upon file space allocation, but they are usually larger than the demanded size
  - Sequential access within extents
  - All extents of a file need not be of the same size
- Example: Linux ext4 file system

Free-Space Management

- Bit vector (*n* blocks)



$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of all-0-value words) +
offset of first 1 bit

- First check whether a DWORD is not 0xffffffff
  - If not, scan for the zero bits

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

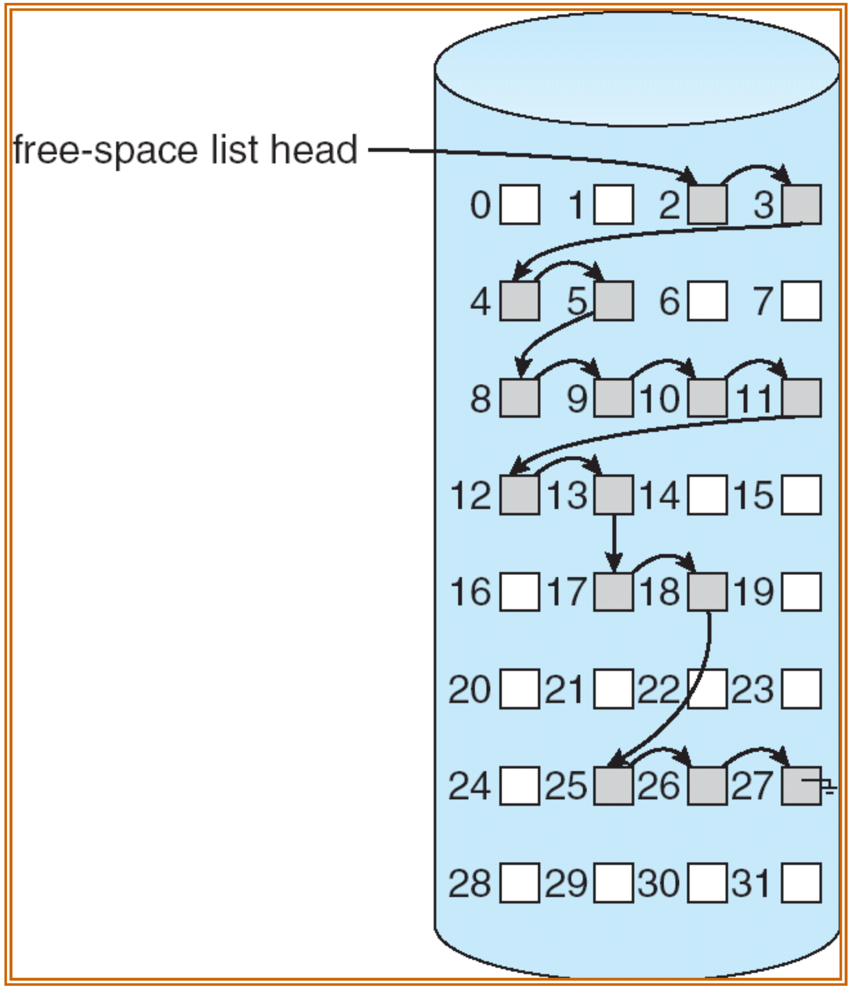
    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    n = $2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Scanning for 0's to find free blocks
- Easy to get contiguous files
  - Check whether a DWORD is zero (0x00000000)
- Used by UNIX FFS, Ext family, ...

Linked Free Space List on Disk

- Allocating and deallocating free blocks in a constant time
- No waste of free space
- But cannot get contiguous space easily, prone to fragmentation
- Used by FAT



File Fragmentation

- File system "ages" after many creation and deletion of files
  - Free space is fragmented into small holes
  - File system cannot find contiguous free space for a new file or for an existing file to grow
- Degree of Fragmentation (DoF) of a file

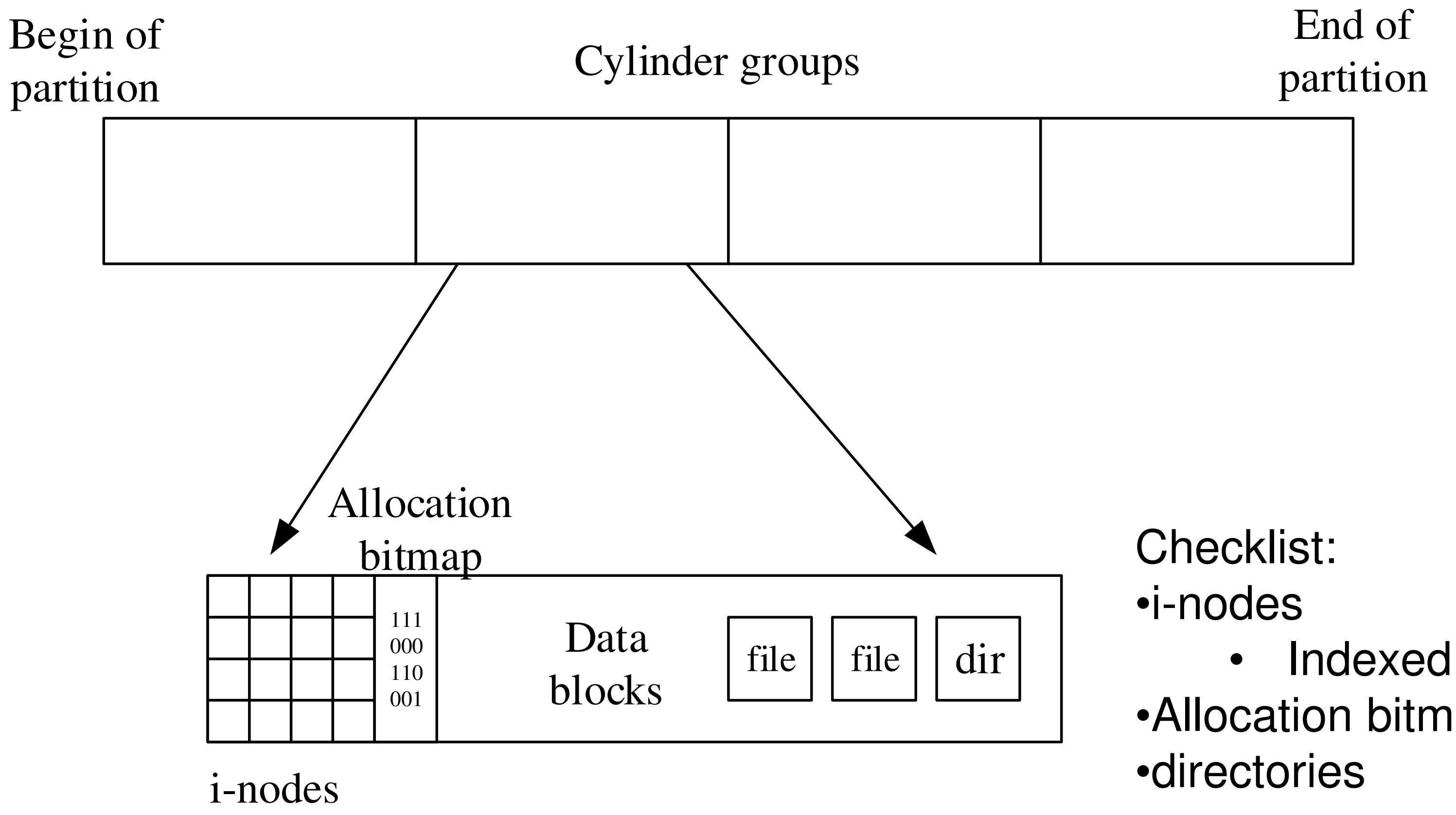
$$DoF = \frac{\text{\# of extents of the file}}{\text{the ideal \# of extents for the file}}$$

  - The higher the DoF of a file is, the more disk seeks are required to access the file

Comparison

- Directory Implementation
  - Plain table: FAT, Ext2/3
  - B-tree: XFS, NTFS, Ext4
- Allocation methods
  - Linked list: FAT
  - Indexed allocation: Ext2/3/4
- Free space management
  - Linked list: FAT
  - Bitmap: Ext

Review: ext4 file system



Checklist:
- i-nodes
    - Indexed allocation
- Allocation bitmaps
- directories

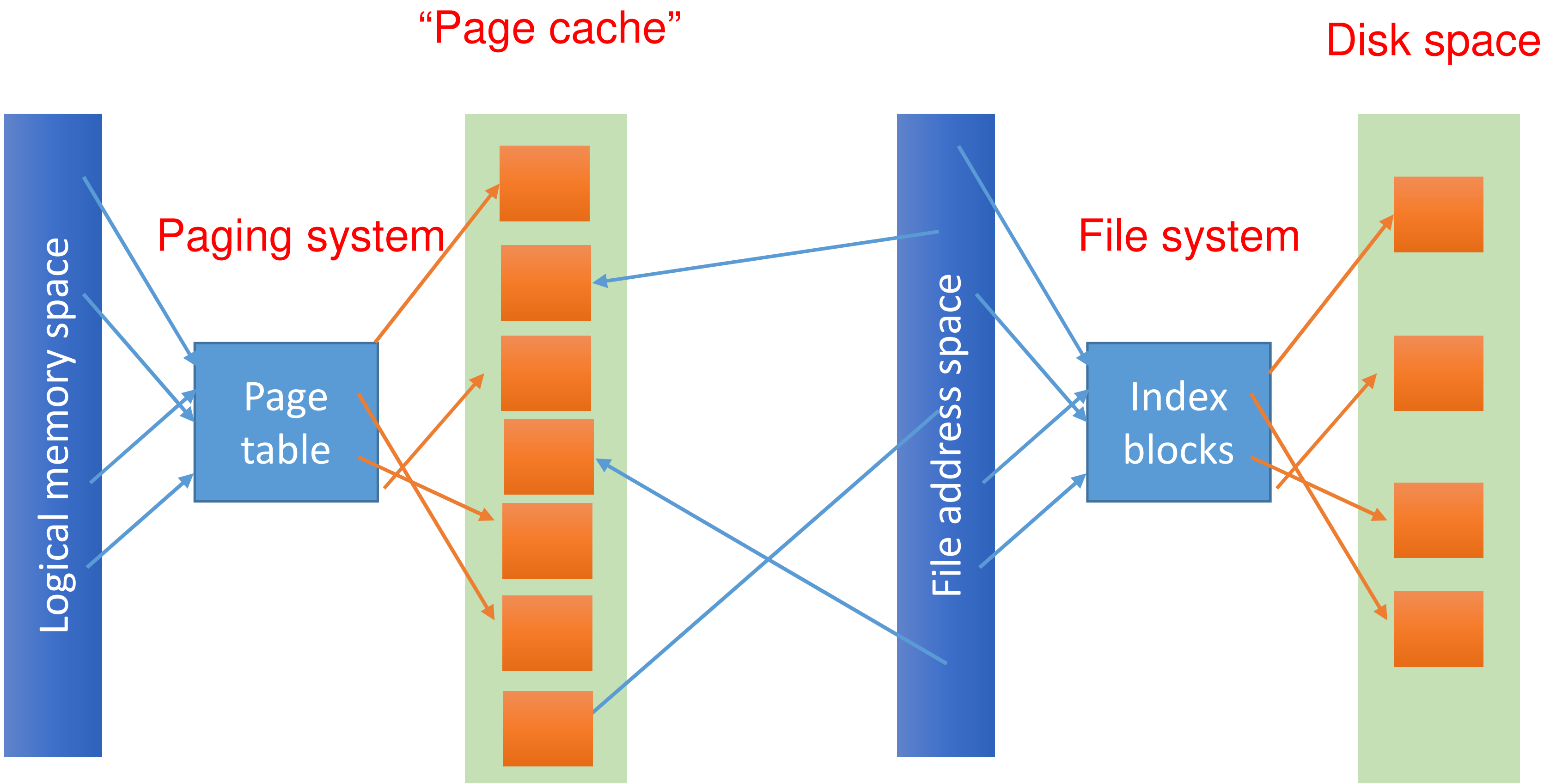
Efficiency and Performance (FS dependent)

- Different file systems have their own optimizations
- Ext4-specific optimizations:
  - Dividing disk space into cylinder groups to make inodes appear near to their associated data blocks
  - Embedding small files into directories (<60 bytes)
  - Using extents to take advantage of sequential disk accesses

Efficiency and Performance (FS independent)

- Kernel-level optimizations shared by file systems
- Disk cache – separate section of main memory for frequently used blocks (temporal locality)
  - Implemented by page cache
- Read-ahead (prefetch) – technique to optimize sequential access
  - Exploting spatial locality of file access
  - Like pre-paging

Page Cache



"Page cache"
Disk space
Logical memory space
Paging system
Page table
File address space
File system
Index blocks

Recovery

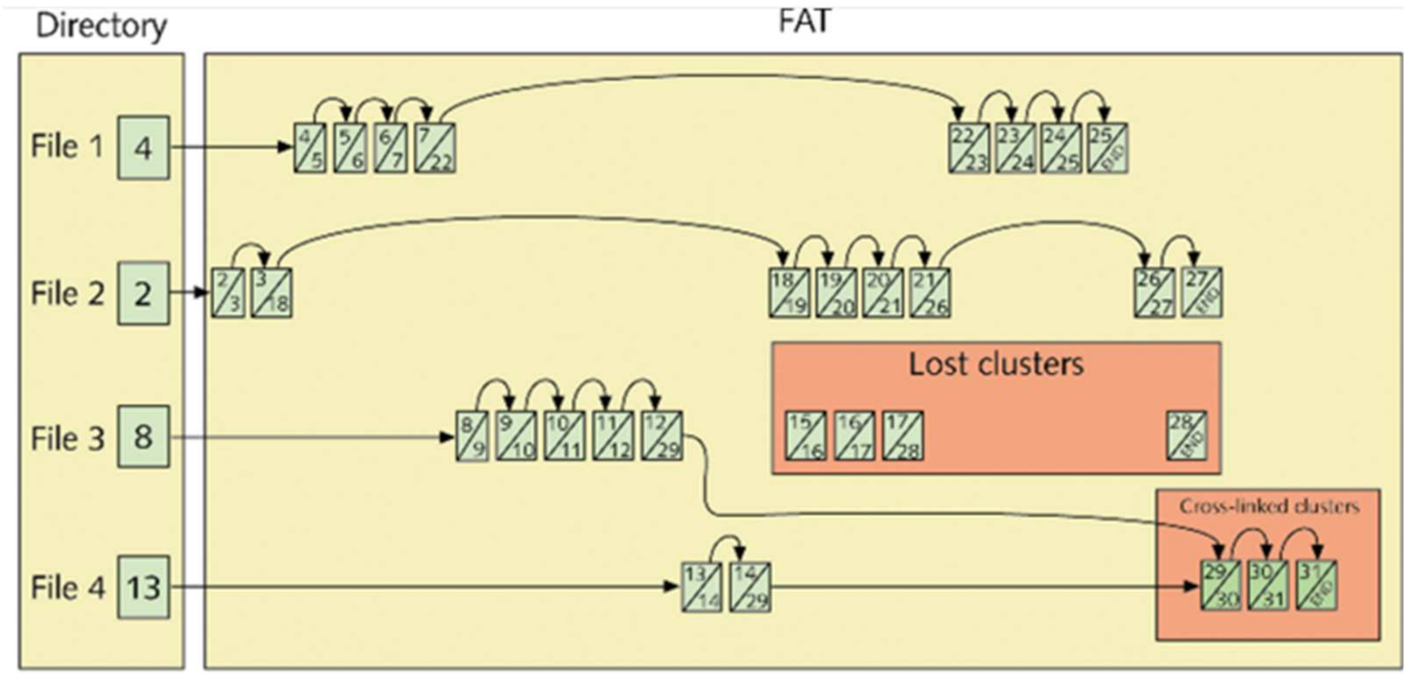
- A file operation modifies multiple blocks
  - Some of them have been written and some are not
  - Unwritten data are lost if the system crashes

Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies

- Loss of metadata: strucrural inconsistency
- Loss of user data: partially written file

Sutrctural Inconsistency Examples

- Ext file systems
  - A bitmap indicates that an inode has been allocated but the inode is not written yet (and vice versa)
  - A hard link is created to a file but the file's reference count has not been incremented yet
- FAT file systems
  - A list of blocks are freed and re-allocated to another file, but the link list table has not been updated yet (cross-linked lists in FAT)



Lost and cross-linked clusters

http://faculty.salina.k-state.edu/tim/ossg/File_sys/file_system_errors.html

Recovery Utilities

- Usually a dirty bit in the super block can tell whether a volume is cleanly unmounted
- Run file system consistency check on dirty volumes
  - `fsck` (UNIX) `scandisk` (Windows)
  - A lengthy process, takes up to 1 hour on a 1 GB disk

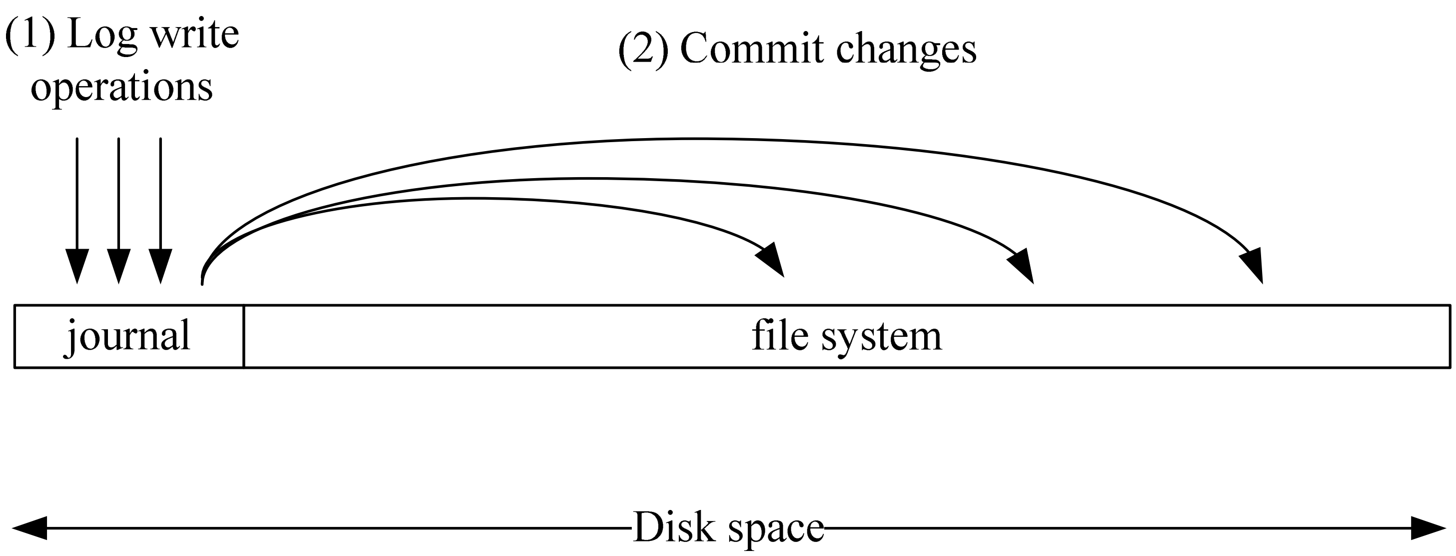
Journaling File Systems

- Guarantee the atomicity of file system operations
  - Atomicity: all or none
  - Structural consistency
- Journaling file systems collect the dirty data produced by a set of (completed) file-system operations into a transaction
  - Write transactions to the journal first, and then modify the file system
  - Journal is a reserved disk space
- If system crashes, upon system reboot, the file system re-do the transactions in the journal
  - partial transitions are discarded

Journaling File Systems

- Transactions
  - An idea borrowed from database systems
  - ACID properties (Atomicity, Consistency, Isolation, Durable)
  - All or none (no partial)
- Journaling
  - Based on write-ahead logging (WAL)
  - Guarantees that file systems are structurally consistent
  - Does not guarantee no loss of data
- When powering up after crash
  - Scan the journal
  - Found a complete transaction → redo
  - Found a partial transaction → discard

Write-Ahead Logging (WAL)



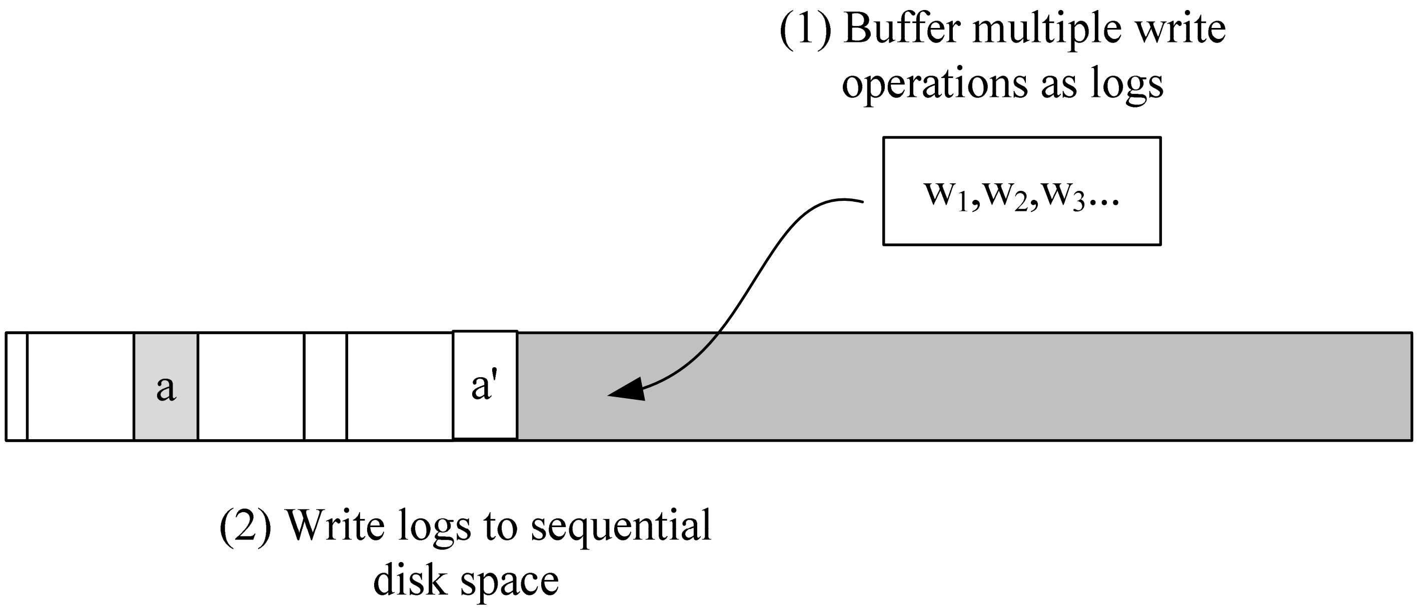
Journaling File systems -- Summary

- Motivation
  - Preventing power interruptions from corrupting file systems
- Method
  - Adding a journal space to the file system
  - Collecting a series of writes as a transaction
  - Write transactions to the journal
  - Apply transactions in the journal to the file system (in background)
  - Incomplete transactions (in disk journal) are discarded
- Benefit
  - On crash, replay the transactions in the journal, no need to scan/fix the file system
- Problem
  - Degraded performance→ amplifying write traffic

Log-Structured File Systems

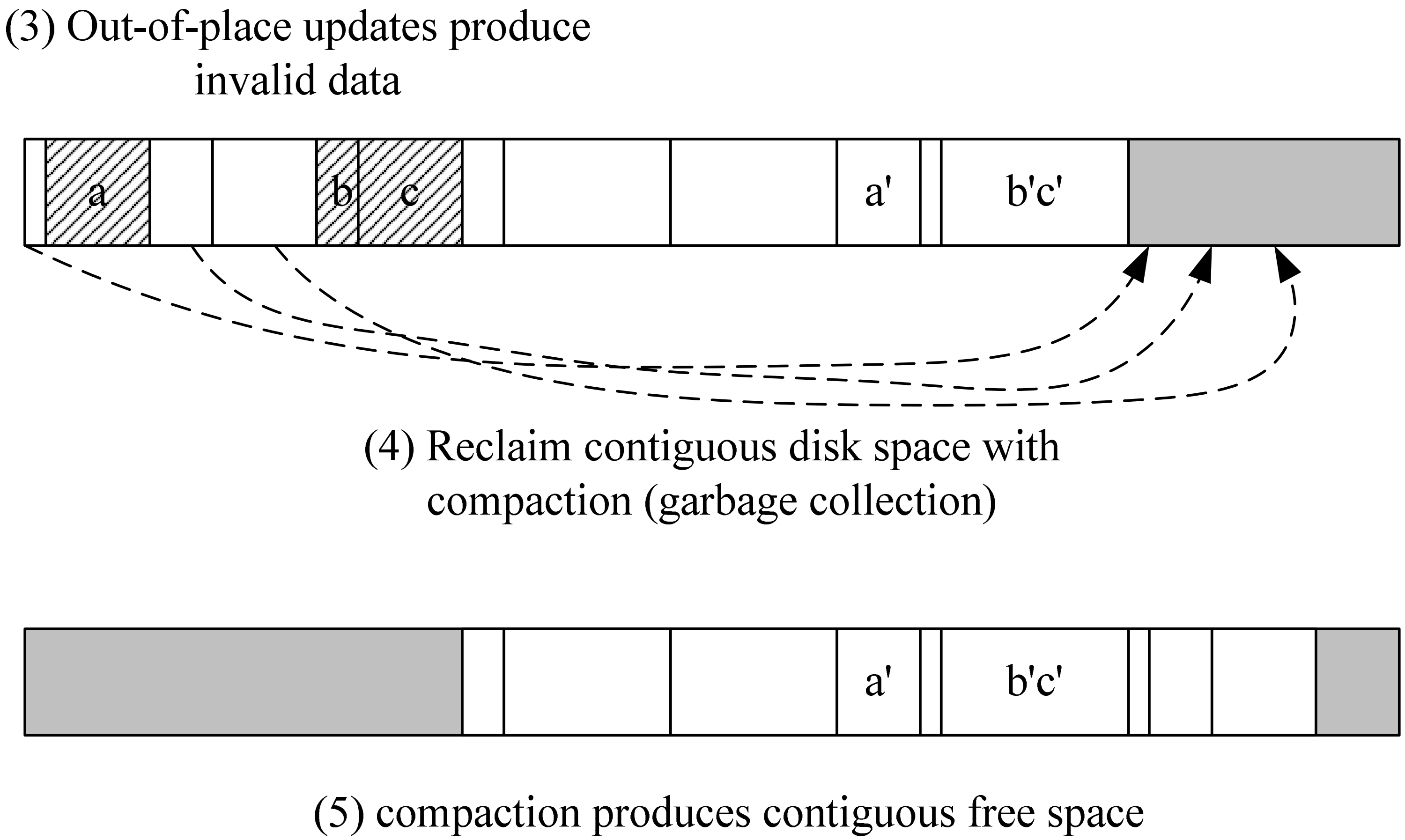
- Log-structured file systems are similar to journaling file systems in many aspects; but they are different
  - LFSs treat the entire disk space as a single logging area
  - No need to "copy back"
- The main idea is to optimize random write
  - Convert random writes into sequential writes
  - Out-of-place updates
- Examples
  - NILFS2
  - F2FS for Android devices

The Concept of Log-Structured File Systems

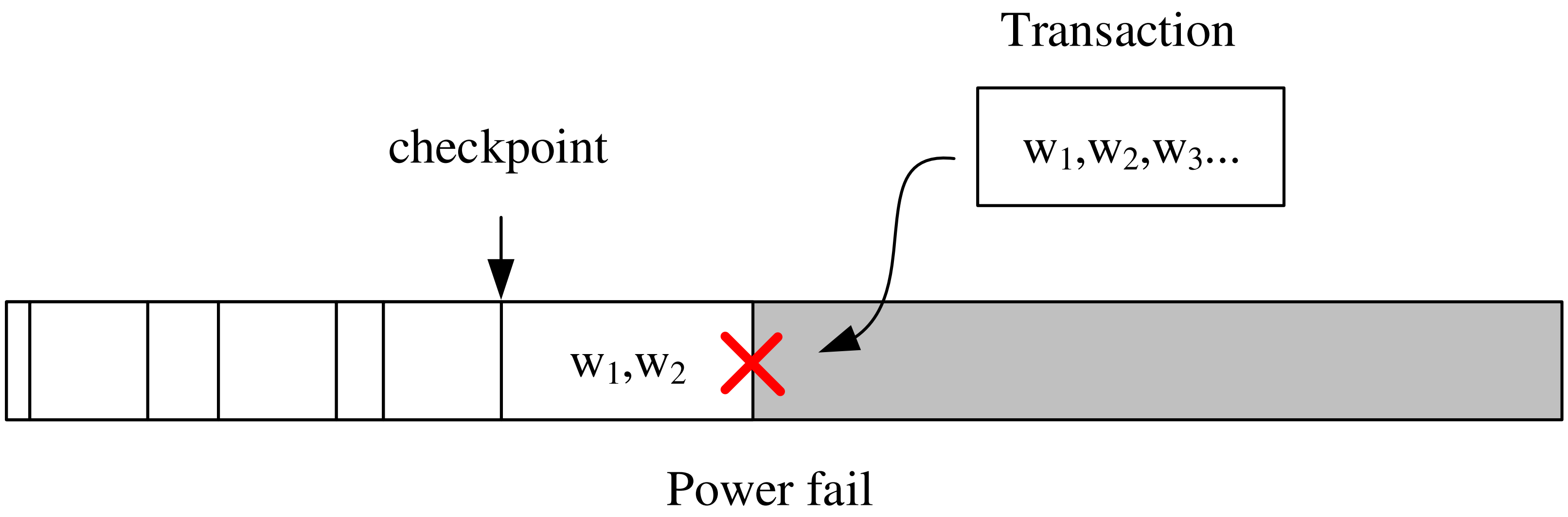


Updates are out of place

Compaction (Garbage Collection) in LFS



Recovery in LFS



Log-Structured File Systems -- Summary

- Motivation:
  - RAM is cheap and a large disk cache can handle read accesses
  - Write requests eventually arrive at the disk
  - Random write is slow
- Methods:
  - Convert random writes into long write bursts (logs)
  - Out-of-place updates
- Benefits:
  - Optimized random write performance
  - Easy recovery
- Problems:
  - Need compaction (garbage collection) to produce sequential space for new writes

End of Chapter 11