Chapter 11: File System Implementation

Prof. Li-Pin Chang
CS@NYCU

Chapter 11: File System Implementation

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

Objectives

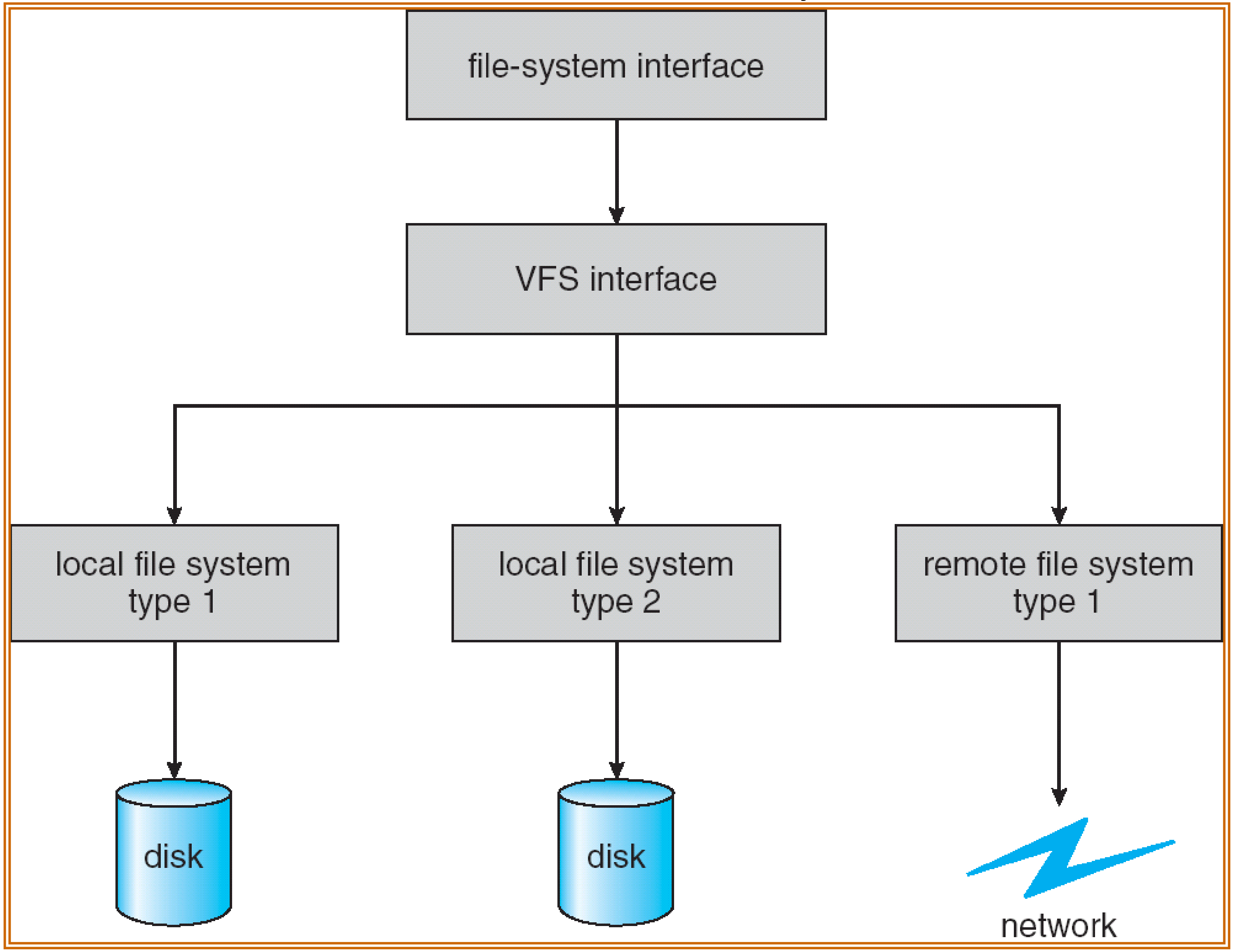
- To describe the details of implementing local file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

File System Structure and Abstraction

Layered File System

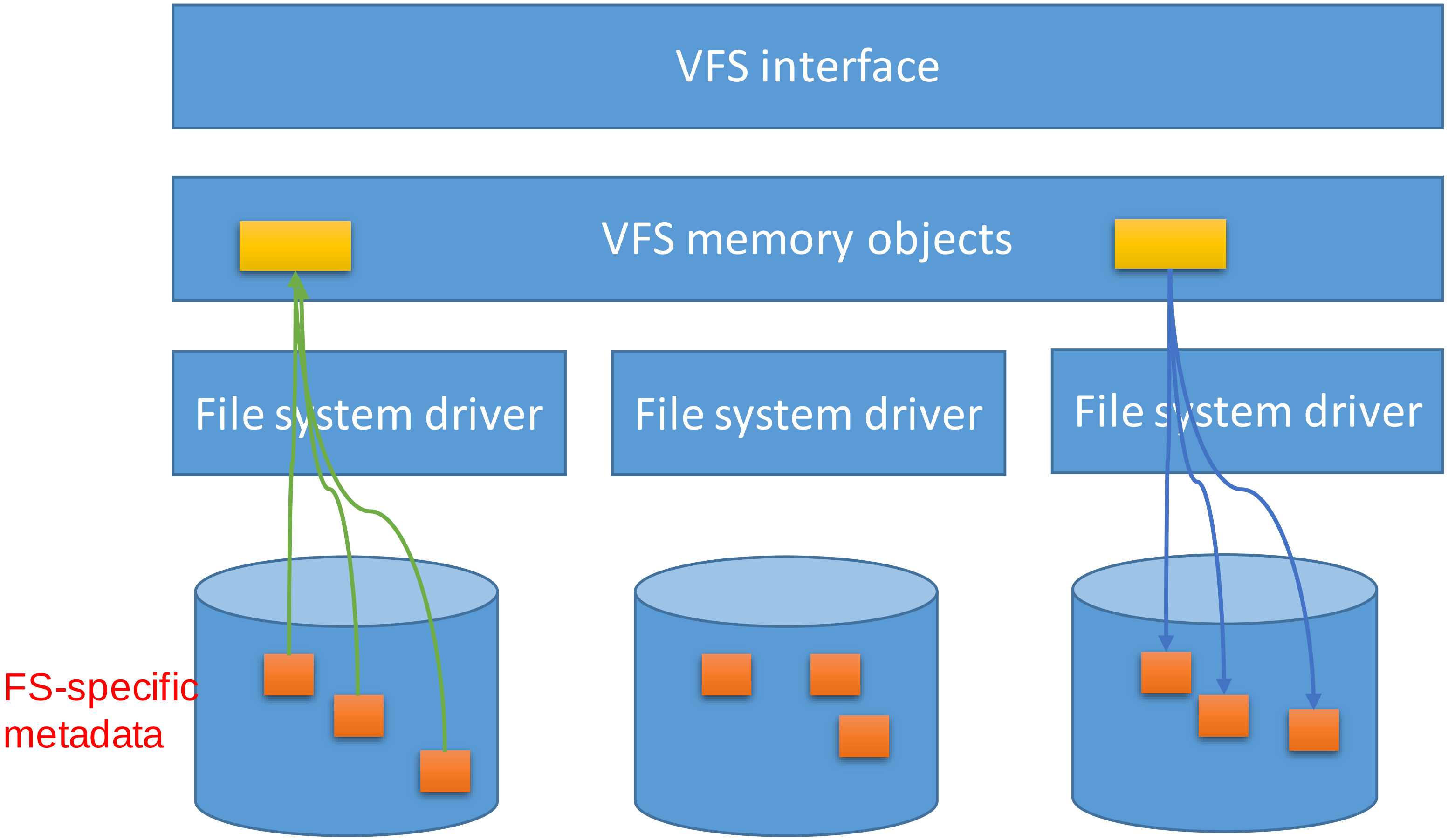
fread() / fwrite()	Applications
fs->read, fs->write	File-system interface
	File-system implementation
Read write to page cache	Page cache
	I/O scheduler
Block read, block writes	Disk driver
Control signals	Disk

Schematic View of Virtual File System



Linux Virtual File System Architecture

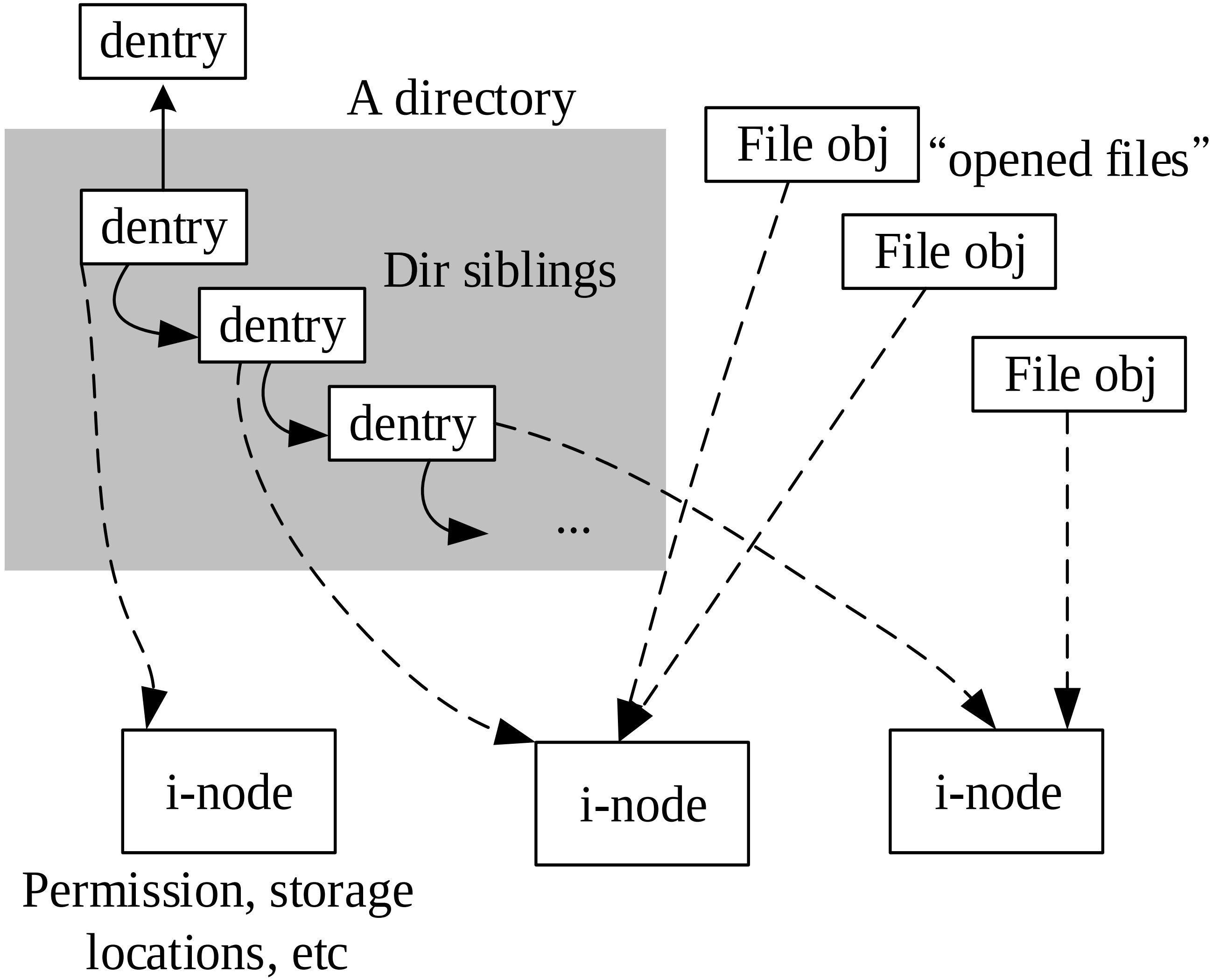
- File system drivers translate between kernel VFS memory objects and disk metadata



In-memory Kernel Objects of Linux VFS

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

In-memory objects of Linux VFS

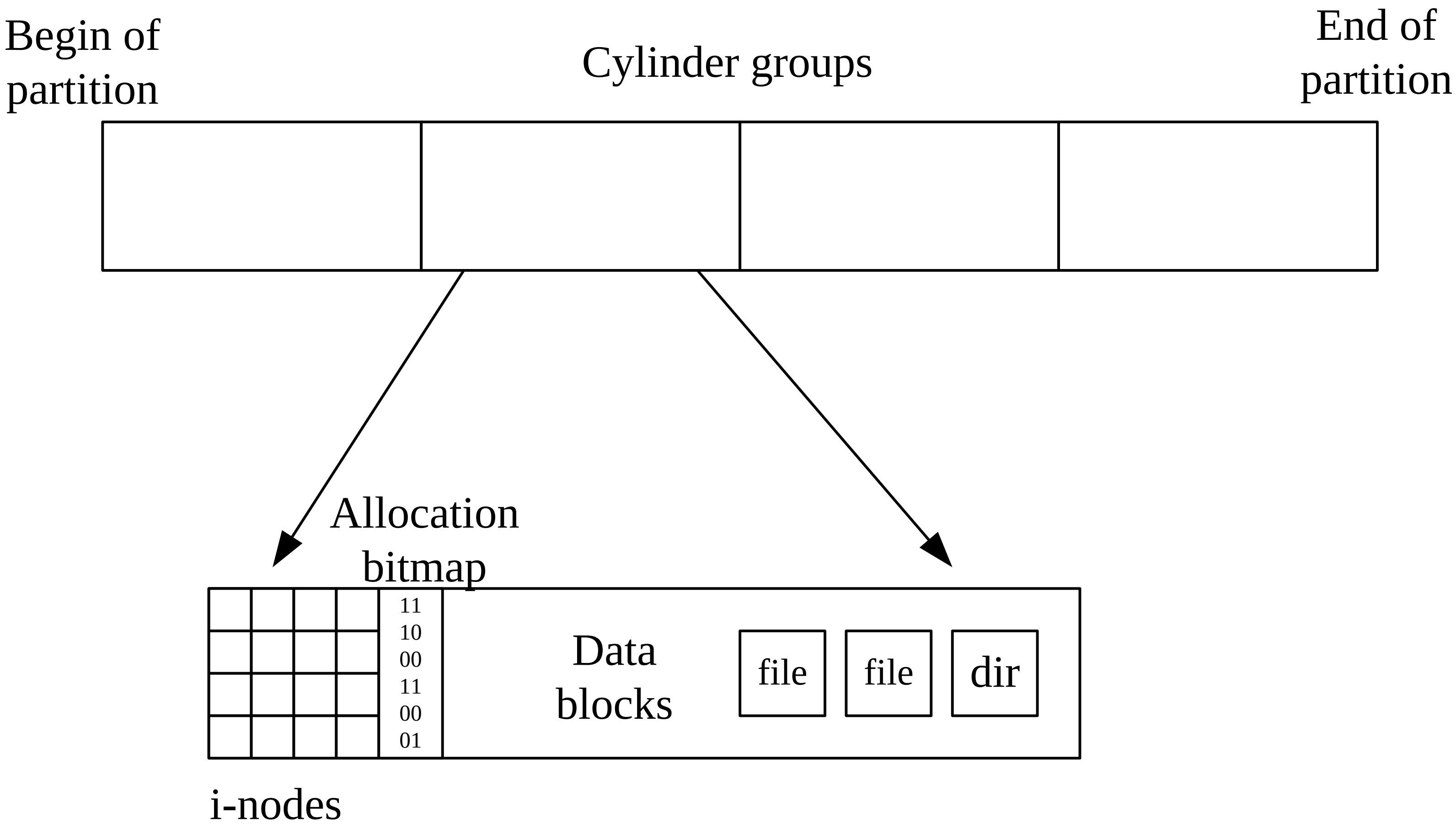


Disk Metadata

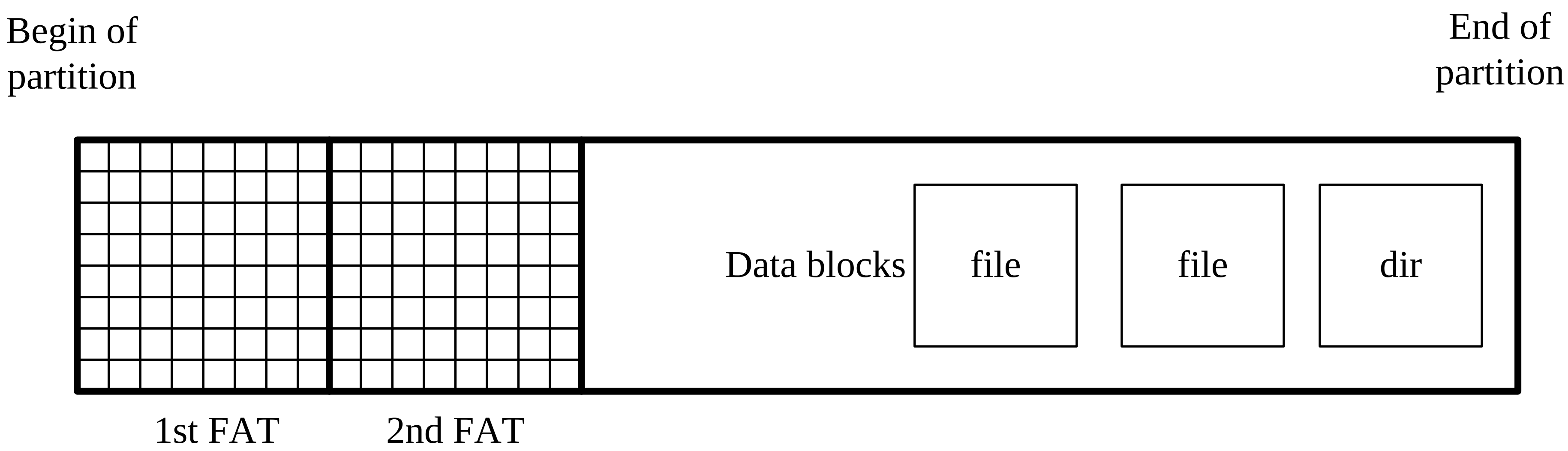
- File-system-specific; vary from file system to file system
- Linux ext file system
  - Super block, Inodes, Allocation bitmaps
- Microsoft FAT file system
  - File allocation tables, Directories

- File system driver must fill the in-memory objects with the information in disk metadata
  - May not be one-to-one mapped, e.g.,
    Ext file system has i-node on disk; FAT file system does not

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



Disk layout of FAT 12/16/32 file systems



File System Key Design Issues

Key Design Issues

1. Directory implementation
2. Allocation (index) methods
3. Free-space management

Issue 1: 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
  - XFS, NTFS, ext4 (H-tree, fixed 2 levels)
  - Scaling well for large directories

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.." ..

Issue 2: Allocation/Index Methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
  - Extent-based 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, unless files are migrated to larger spaces
- Efficient access; perfect for I/O overhead reduction
  - file offset can be directly translated into sector block #
  - Less I/Os involved
  - Always sequential disk read/write
- Wasteful of space (dynamic storage-allocation problem)
  - File deletion leaves free holes (external fragmentation)
  - Needs compaction, maybe done in background or downtime

Contiguous Allocation

- Mapping from logical to physical
- LA = file offset (bytes); 1 disk block = 512 bytes

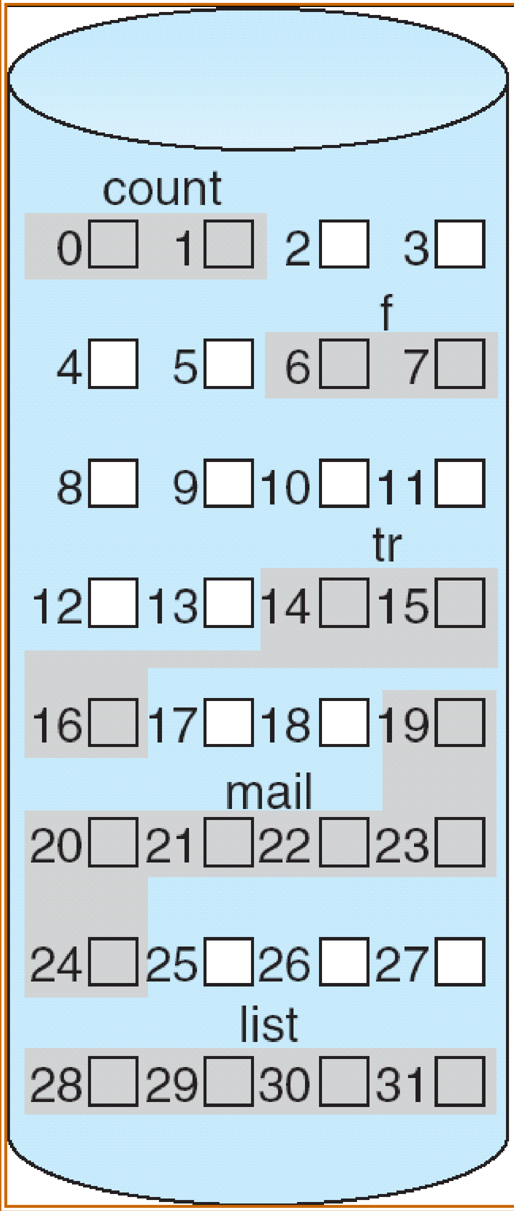
Q

LA/512

R

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

Contiguous Allocation of Disk Space

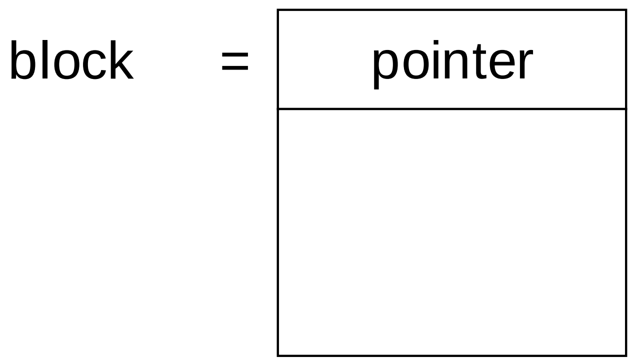


directory

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
- Physical contiguity of the disk blocks is not absolutely necessary because file data are copied to sequential memory before use

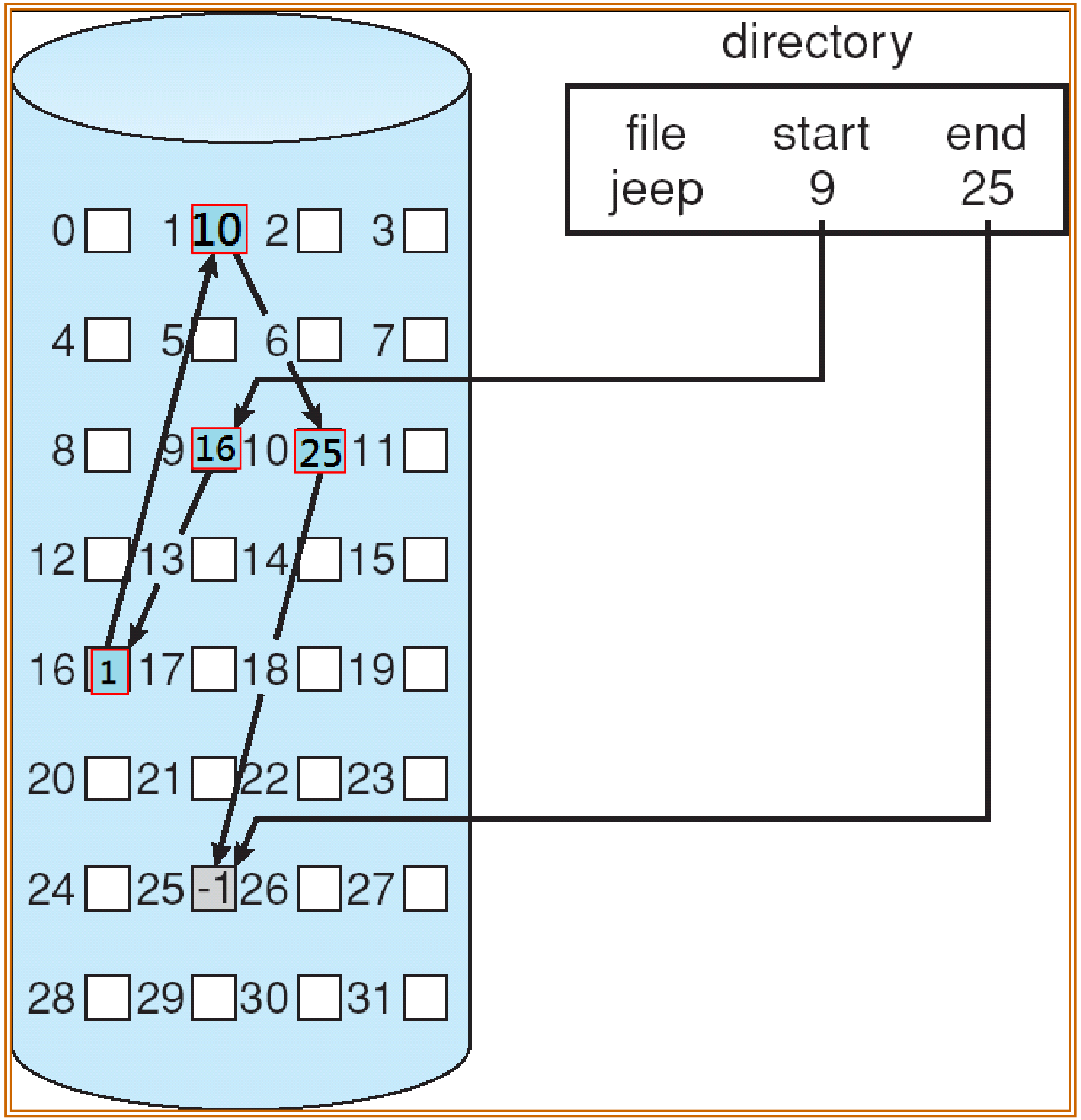


Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system
  - no waste of space (no external fragmentation)
  - However, no random access (need to traverse the linked blocks)
- Mapping
  - 1 byte for pointer, so 511 bytes for user data
  - Block to be accessed = the Qth block in the file's linked list
  - Displacement into block = R + 1 (the 0th byte is for pointer)

$$LA/511 \begin{cases} Q \\ R \end{cases}$$

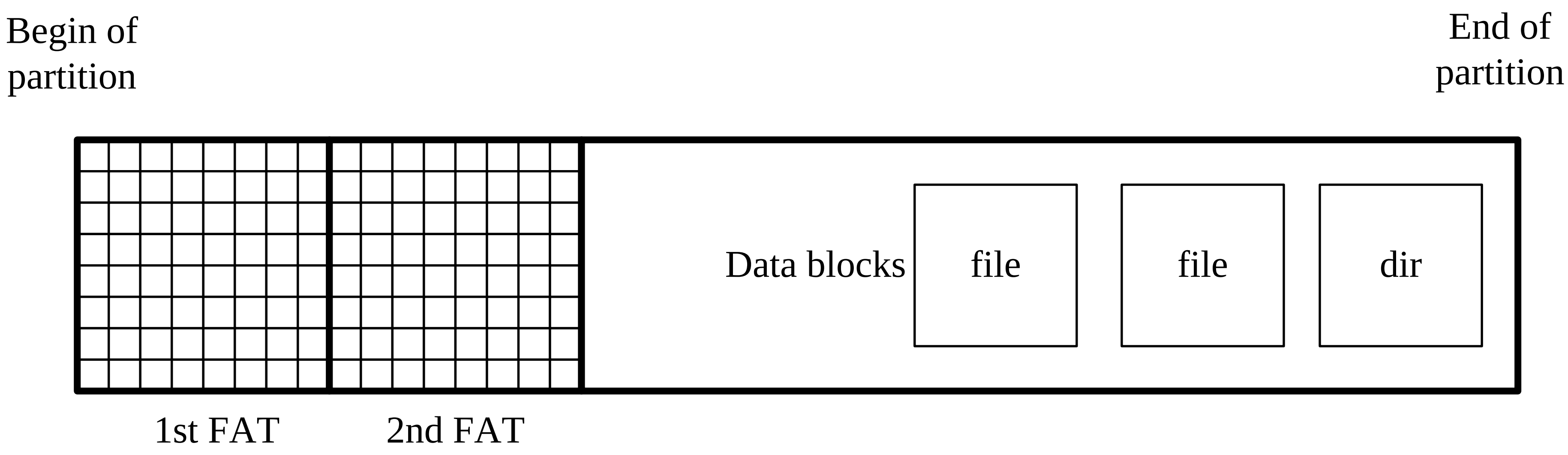
Linked Allocation



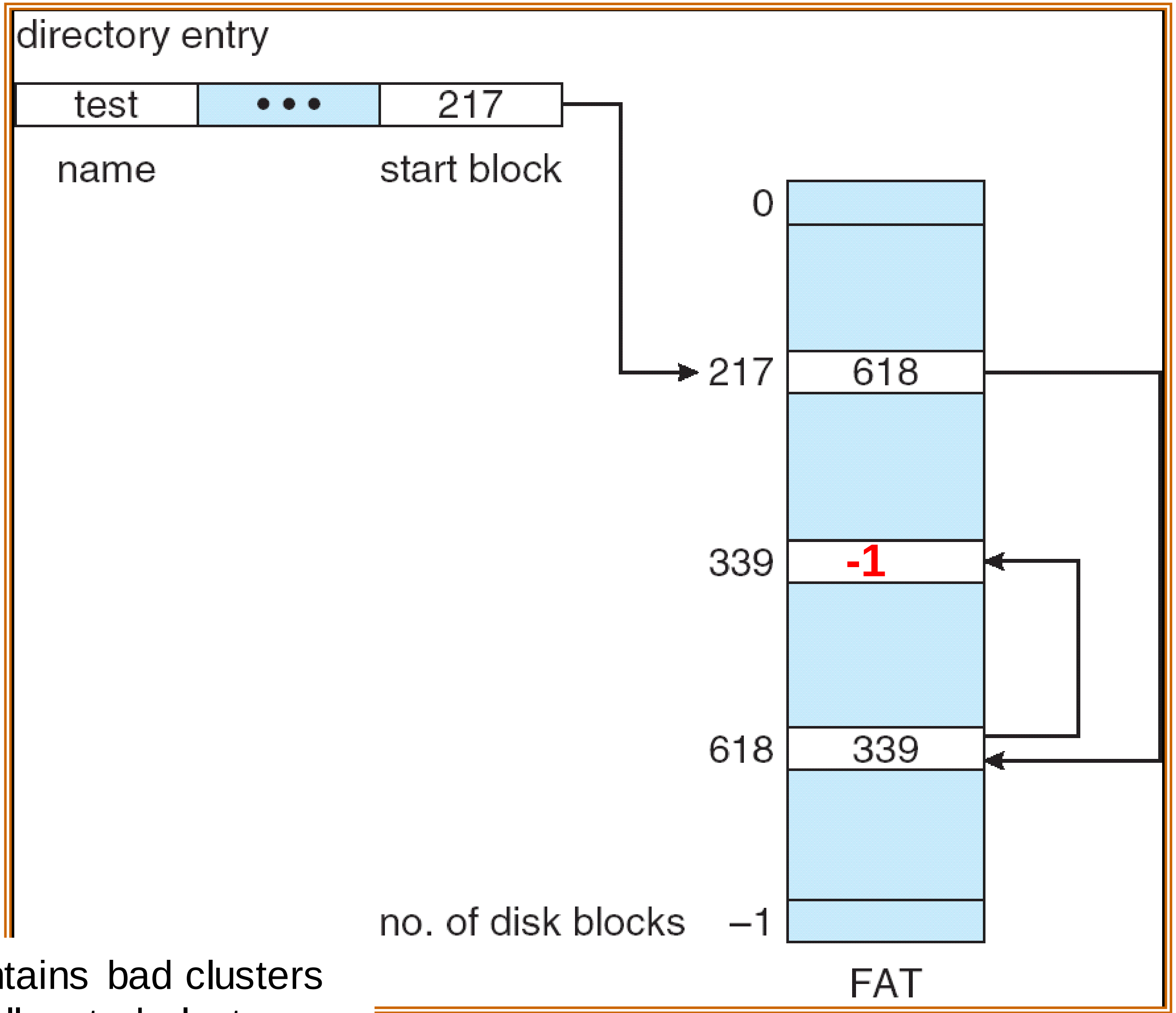
Linked Allocation

- Separating pointers from data blocks
  - Making data size a power of 2; easier to manage
- Example: FAT file system

The layout of FAT 12/16/32 file system



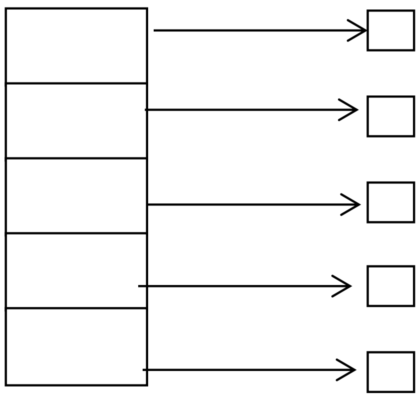
File-Allocation Table



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

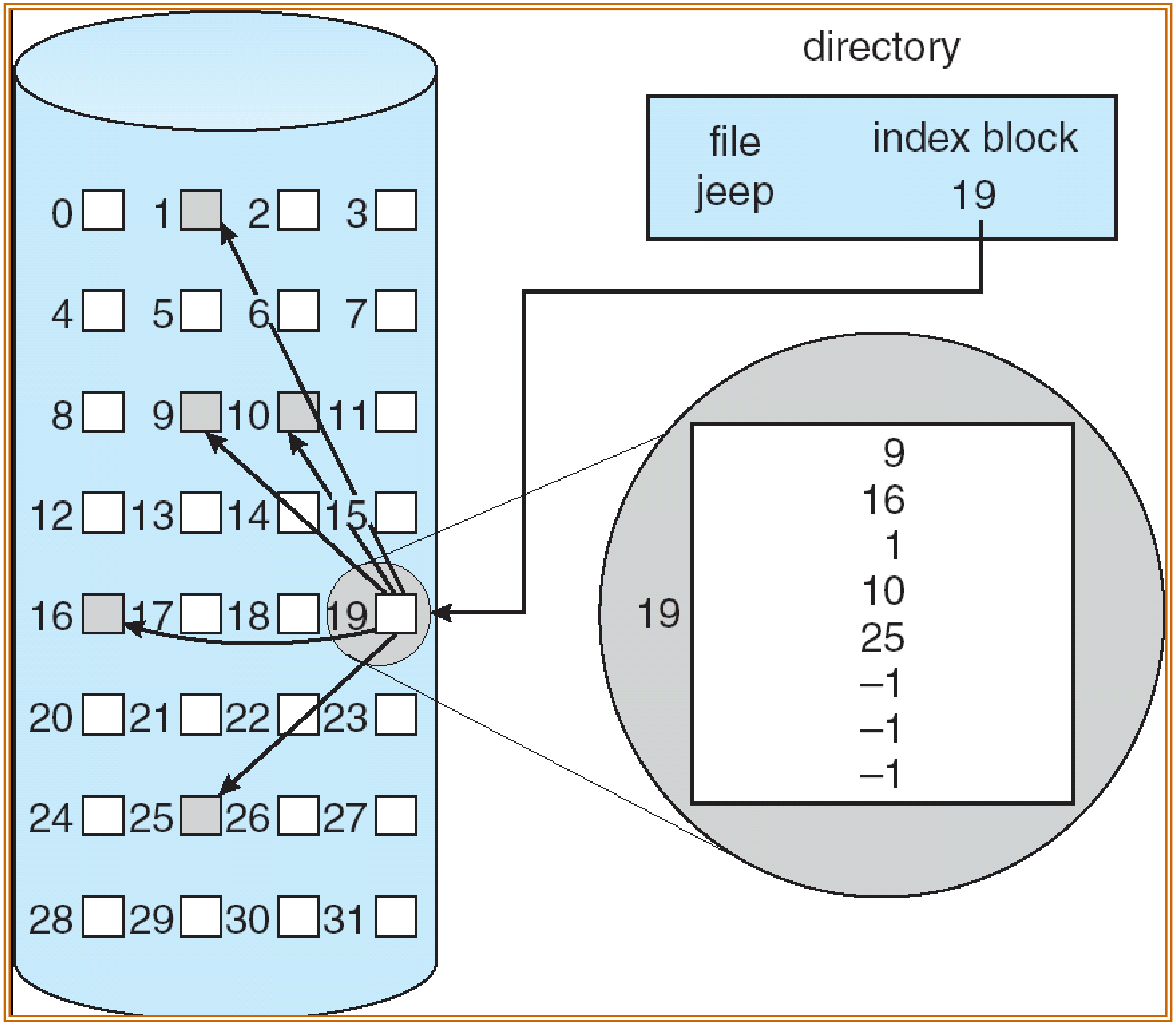
Indexed Allocation

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



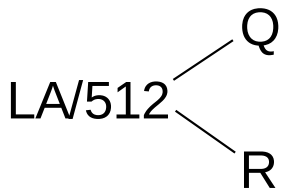
index table

Example of Indexed Allocation



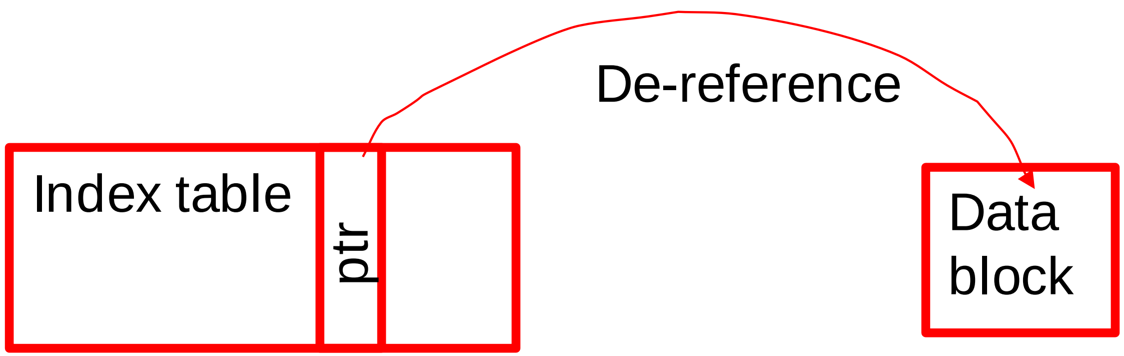
Indexed Allocation (Cont.)

- Need a index table
- Capable of “random” access; no list traversing
- Per-file overhead of an index table (block)



Q = displacement into index table (entry #)

R = displacement into the referred block



Indexed Allocation – Mapping

- Assuming two-level index

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

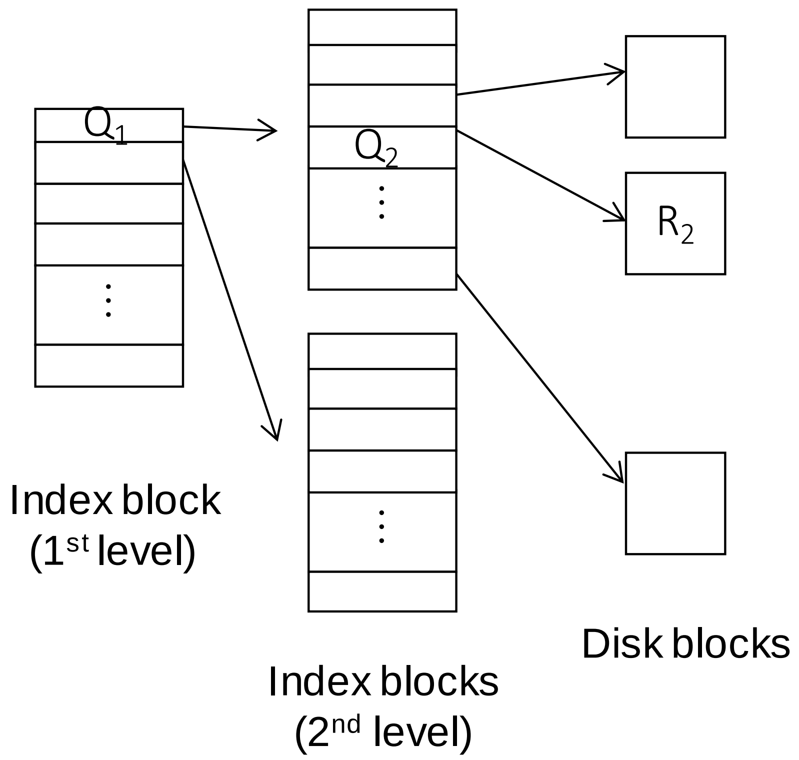
Q_1 = displacement into outer-index

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

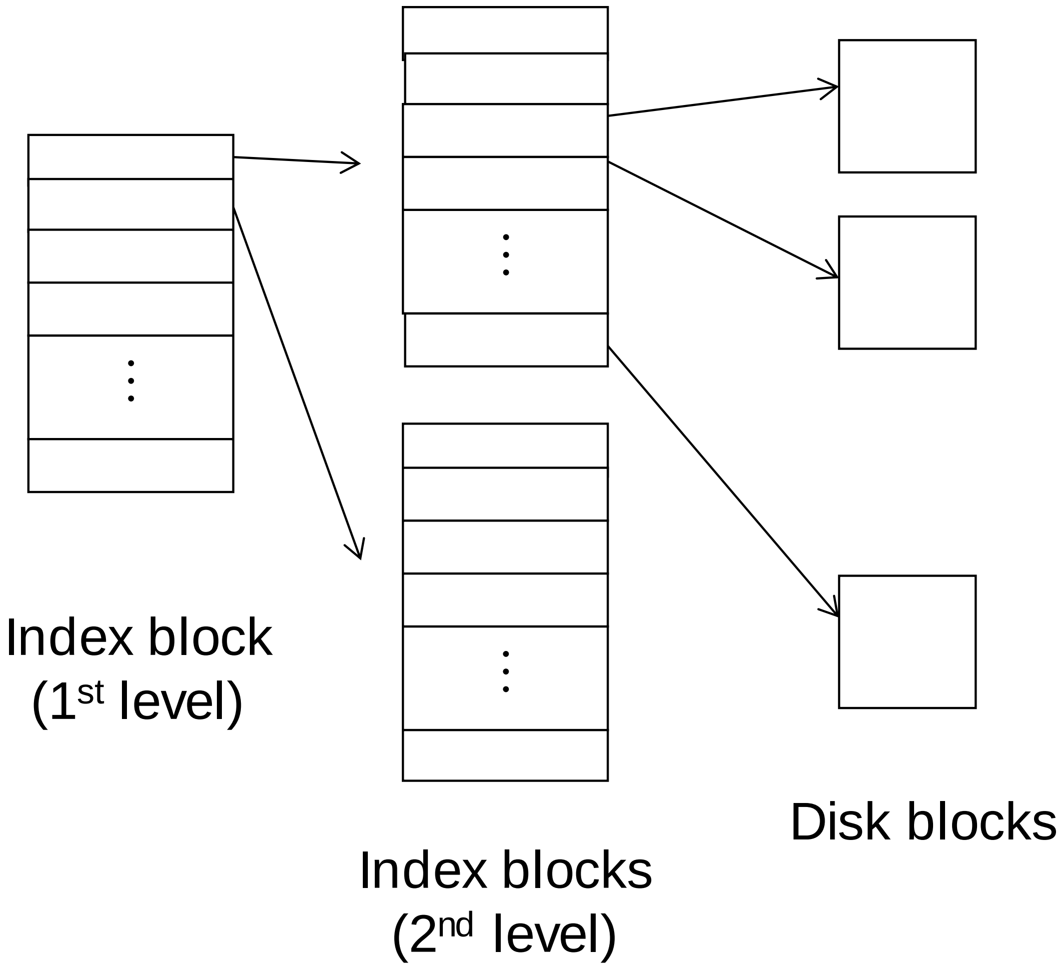


Indexed Allocation – Mapping (Cont.)

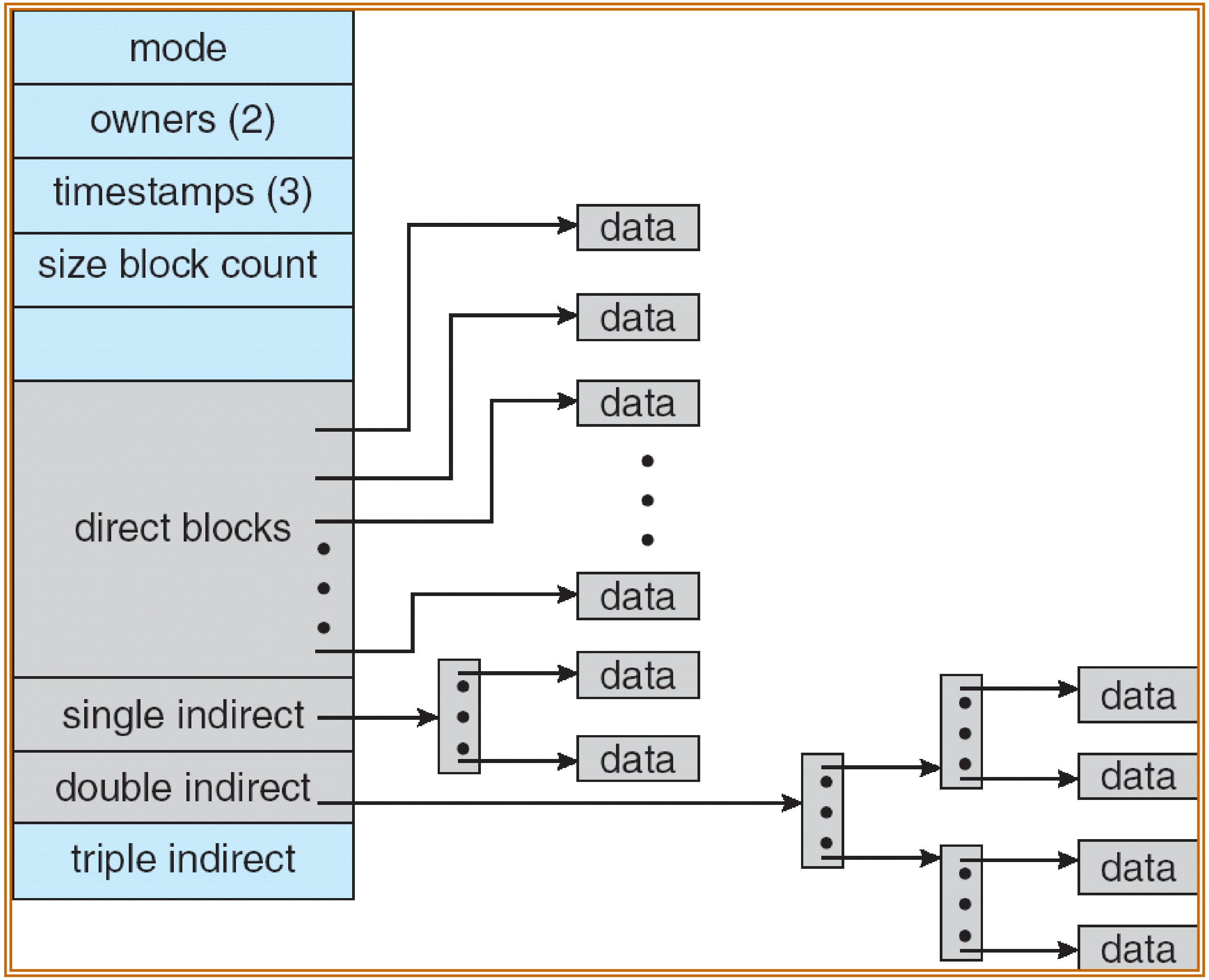
- 1block=512B, 1ptr=1B
- 1 idx. block has 512 ptrs

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

- Max. file size = 512*512*512 bytes
- Isn't it similar to two-level page tables?



Example: UNIX inode



Small files use only direct blocks

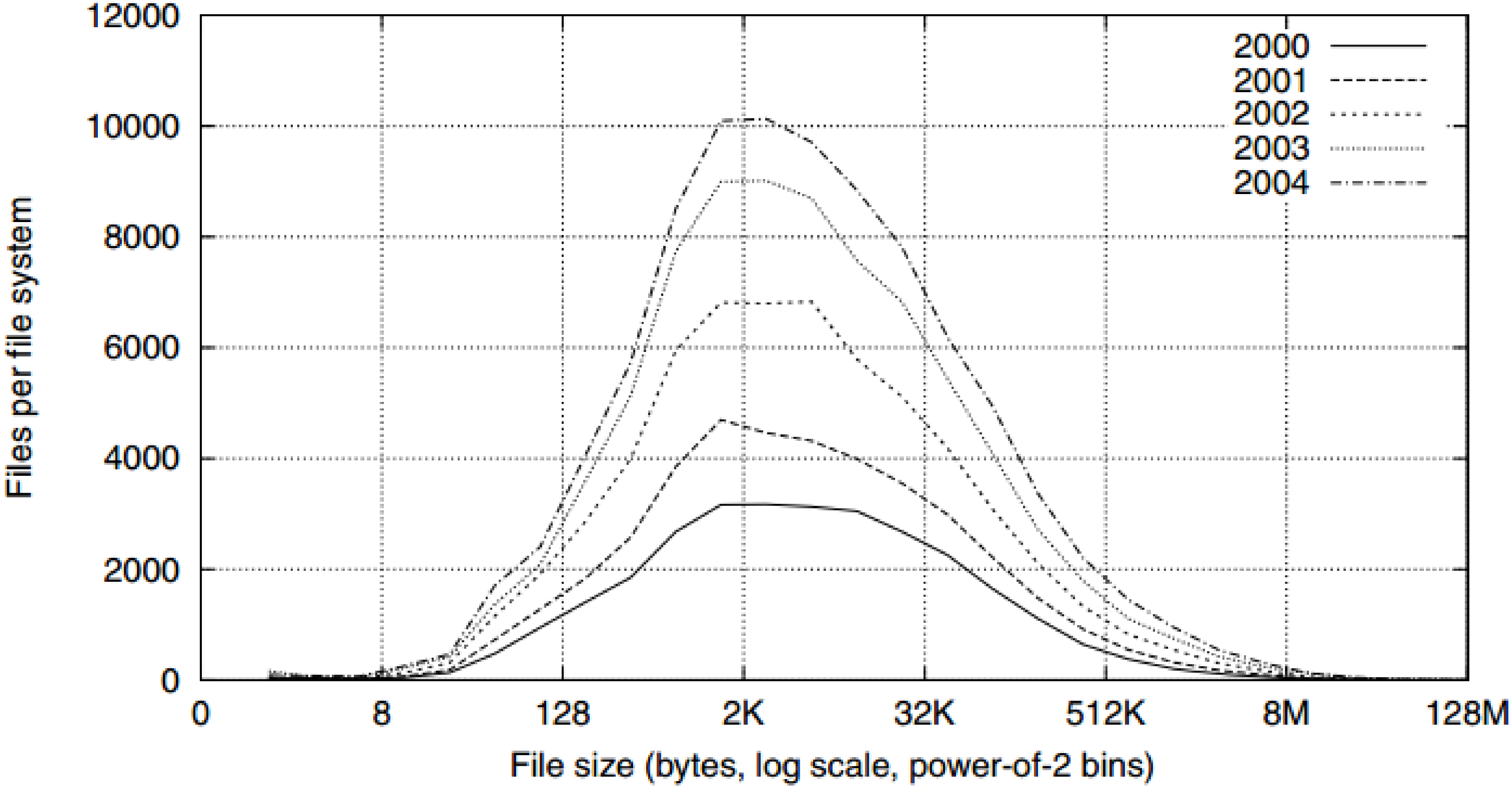
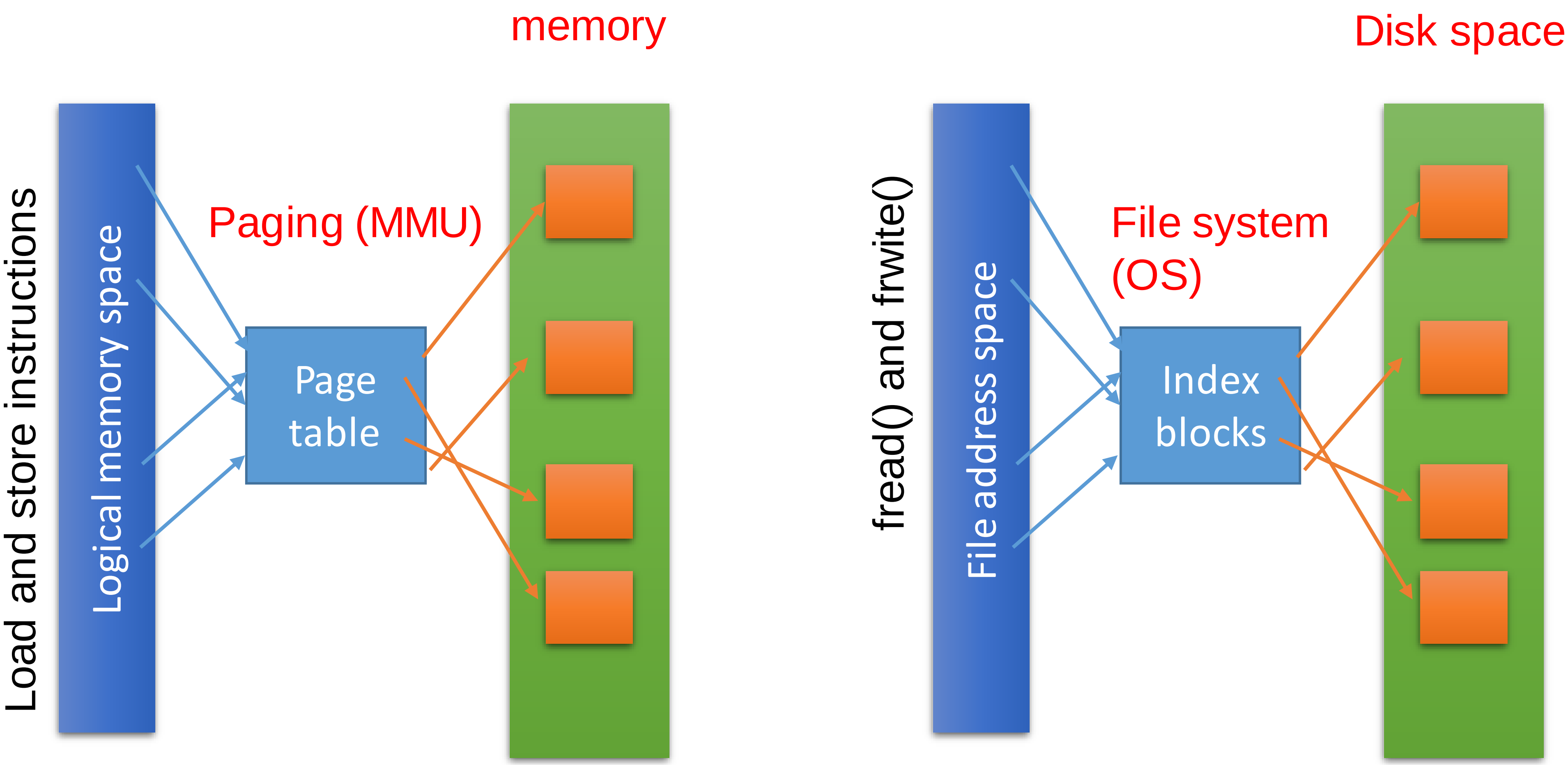


Fig. 2. Histograms of files by size.

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

Indirection, indirection, indirection …



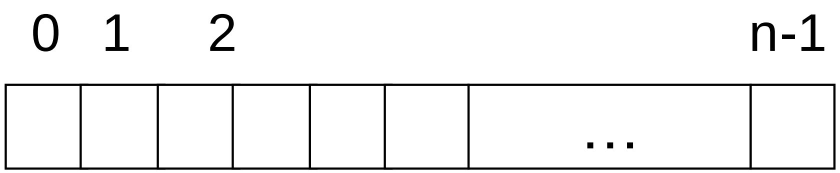
"All problems in computer science can be solved by another level of indirection" -- David Wheeler

Extent-Based Allocation

- 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

Issue 3: 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:

$$\text{block size} = 2^{12} \text{ bytes}$$

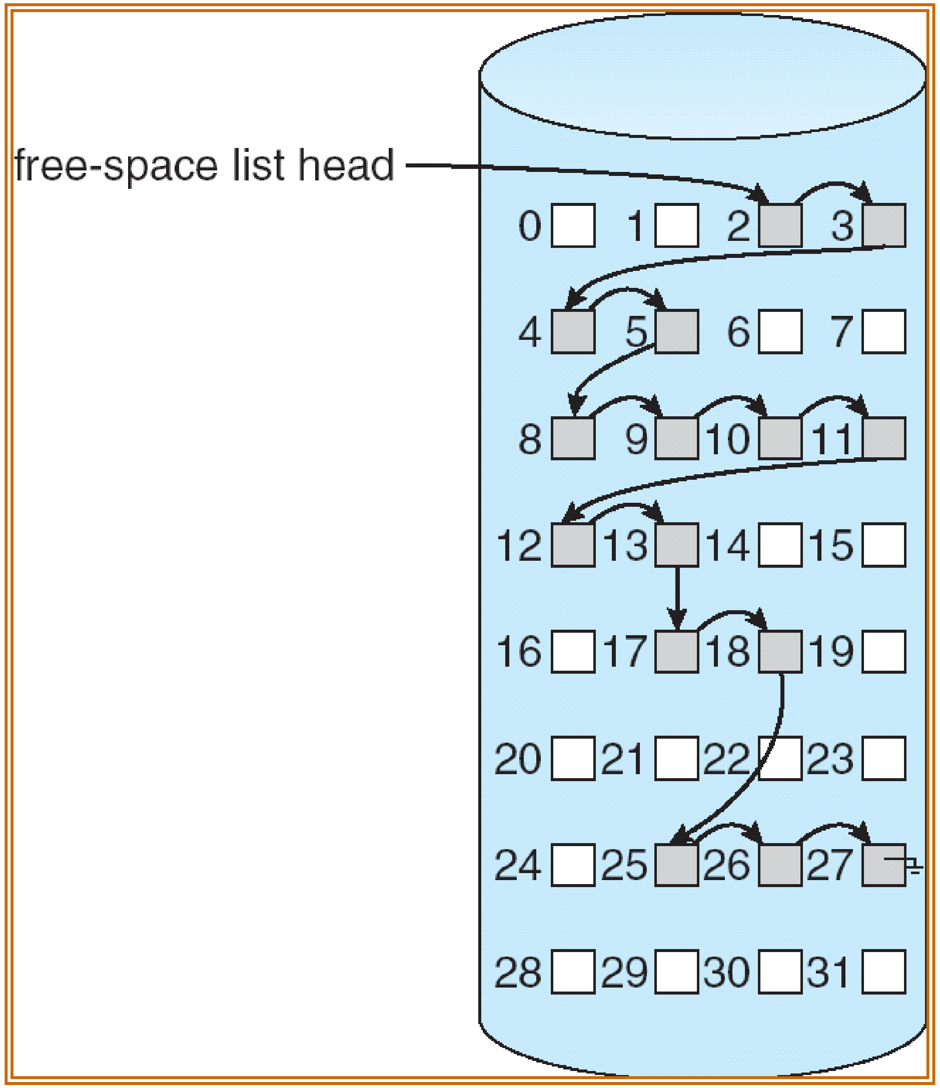
$$\text{disk size} = 2^{30} \text{ bytes (1 gigabyte)}$$

$$n = 2^{30}/2^{12} = 2^{18} \text{ bits (or 32K bytes)}$$

- 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

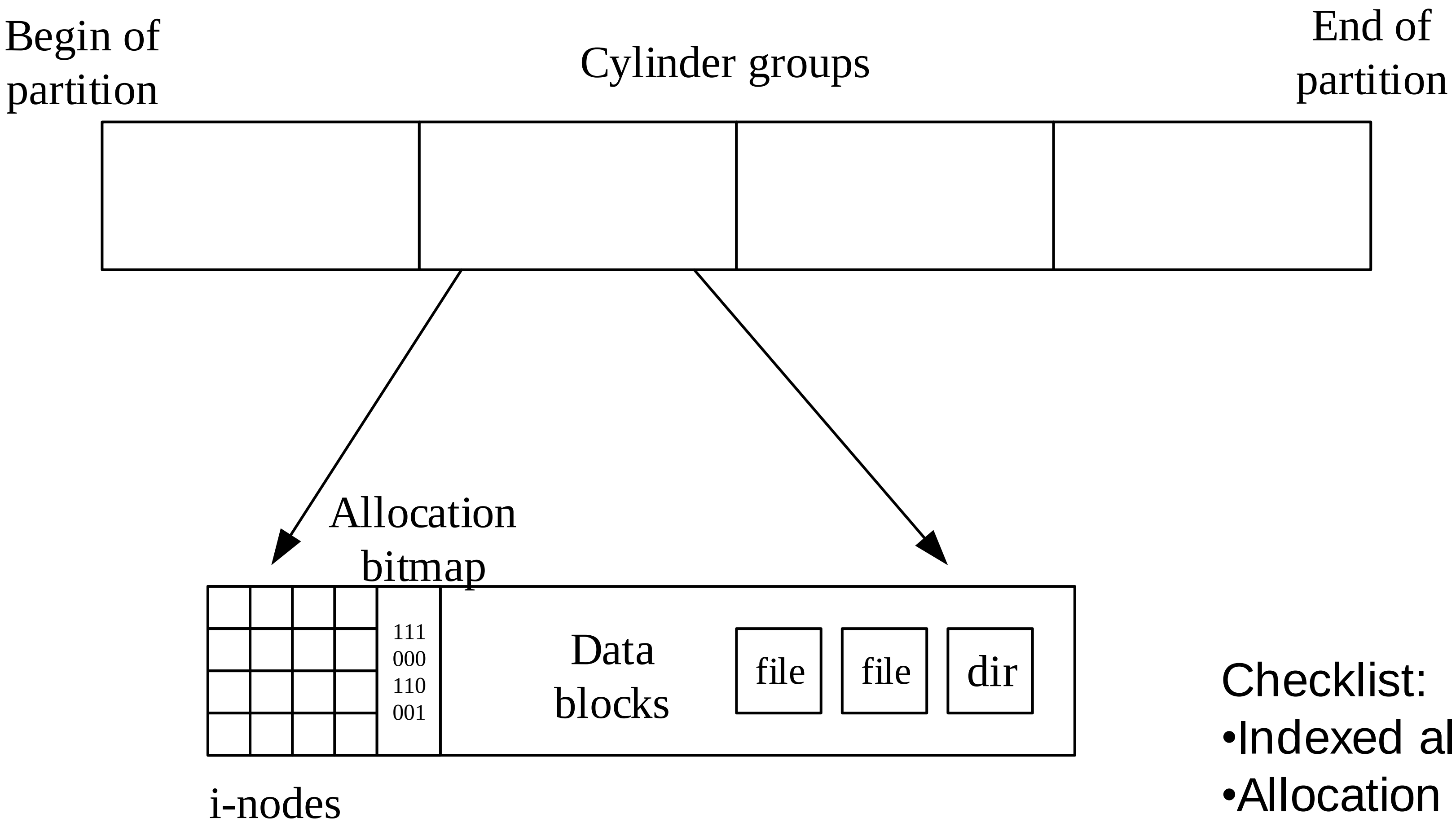
- Allocating and deallocating free blocks in a constant time
- No waste of free space
- But cannot get contiguous space easily, prone to fragmentation
- Not seen in modern file systems



Comparison

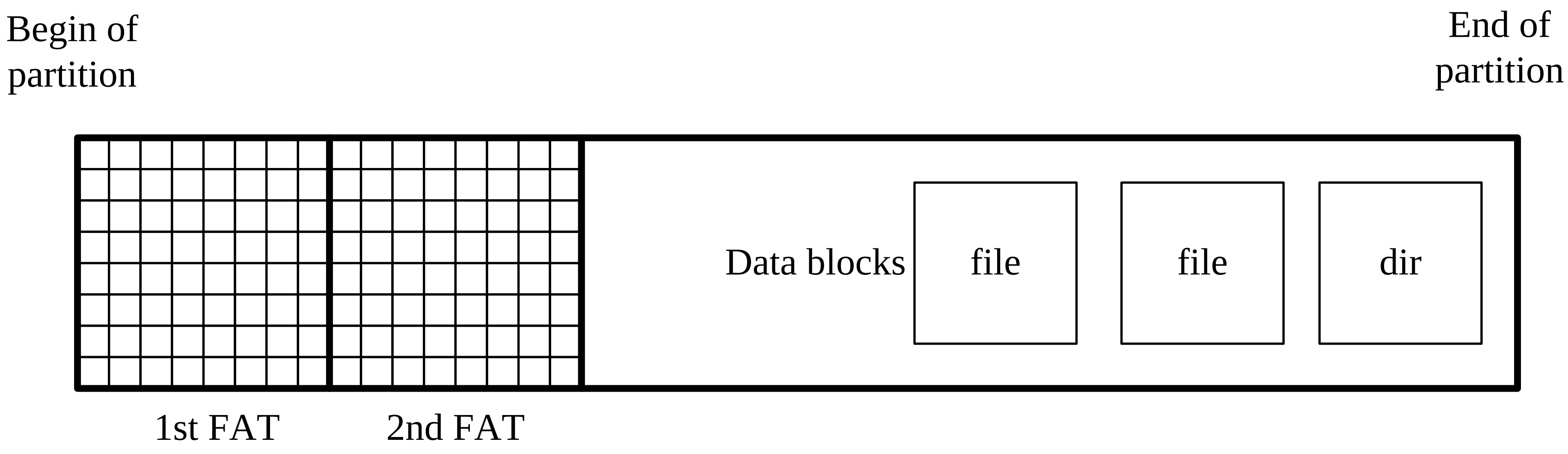
- Directory Implementation
  - Plain table: FAT, Ext2
  - B-tree: XFS, NTFS, Ext3/4
- Allocation methods
  - Linked list: FAT
  - Indexed allocation: Ext2/3/4
- Free space management
  - Linked list: ?
  - Bitmap: Ext

Review: ext4 file system



Checklist:
- Indexed allocation
- Allocation bitmaps
- Directories (H-tree)

Review: FAT file system



Check list
- Linear directory table
- Linked allocation
- Scan 0 in FAT for free space (similar to bitmap)

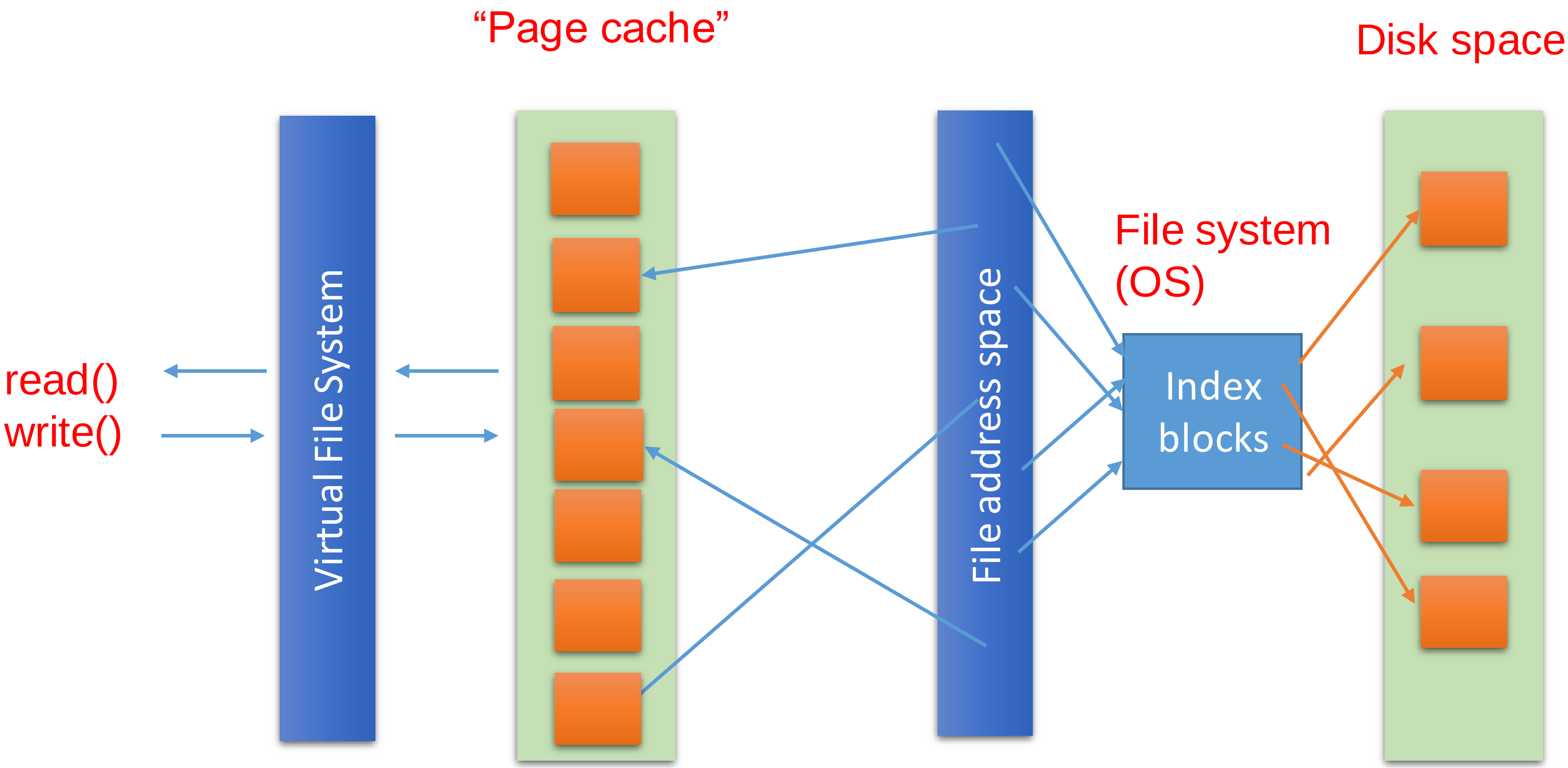
Efficiency and Performance

Generic Optimization

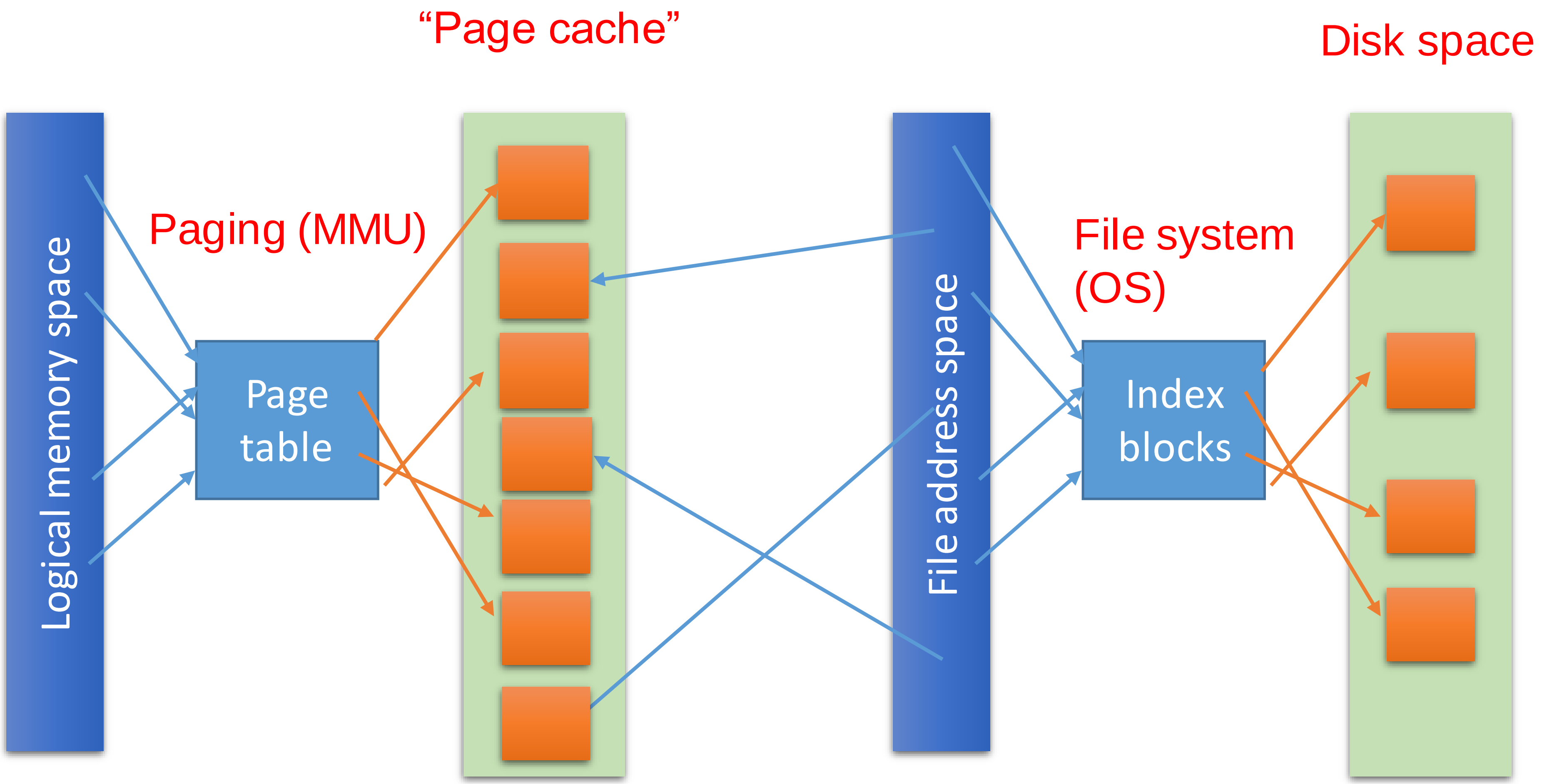
- The OS kernel provides generic optimization methods that all file systems can use

- Disk cache (page caching) – separate section of main memory for frequently used blocks (temporal locality)
- File read-ahead (prefetching)– technique to optimize sequential access (spatial locality)
  - Similar to pre-paging. Difference: file read-ahead size doubles if prefetched data are used.
  - Applications uses `fadvise()` to tell the kernel about how aggressive prefetching should be

Page Caching: Regular Files



Page Caching: mmap()'ed Files



FS-Specific Optimizations

- File systems have their unique techniques for performance optimization

For example, Ext4 employ the following 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

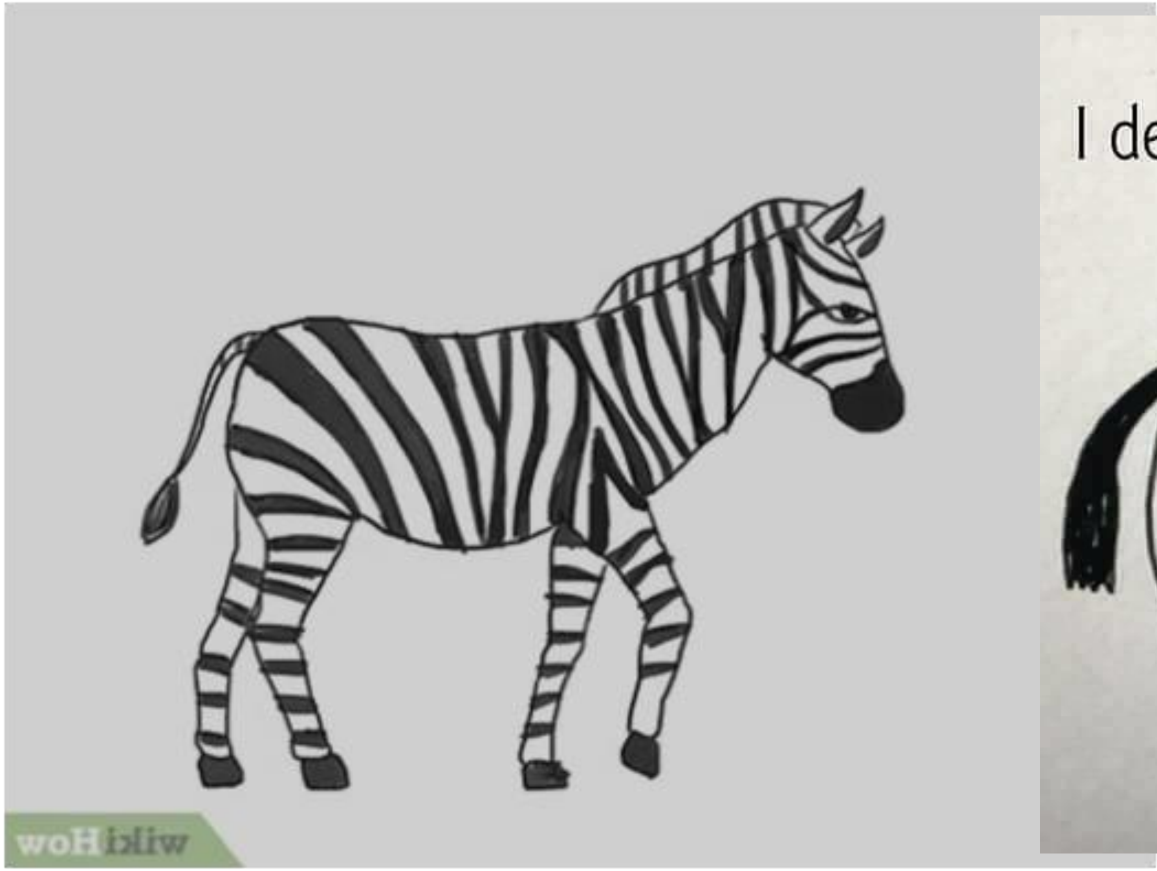
File Fragmentation

- 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

File Defragmentation



Making fragmented files sequential.
→Reducing I/O count and disk head movement on file access.

Consistency and Recovery

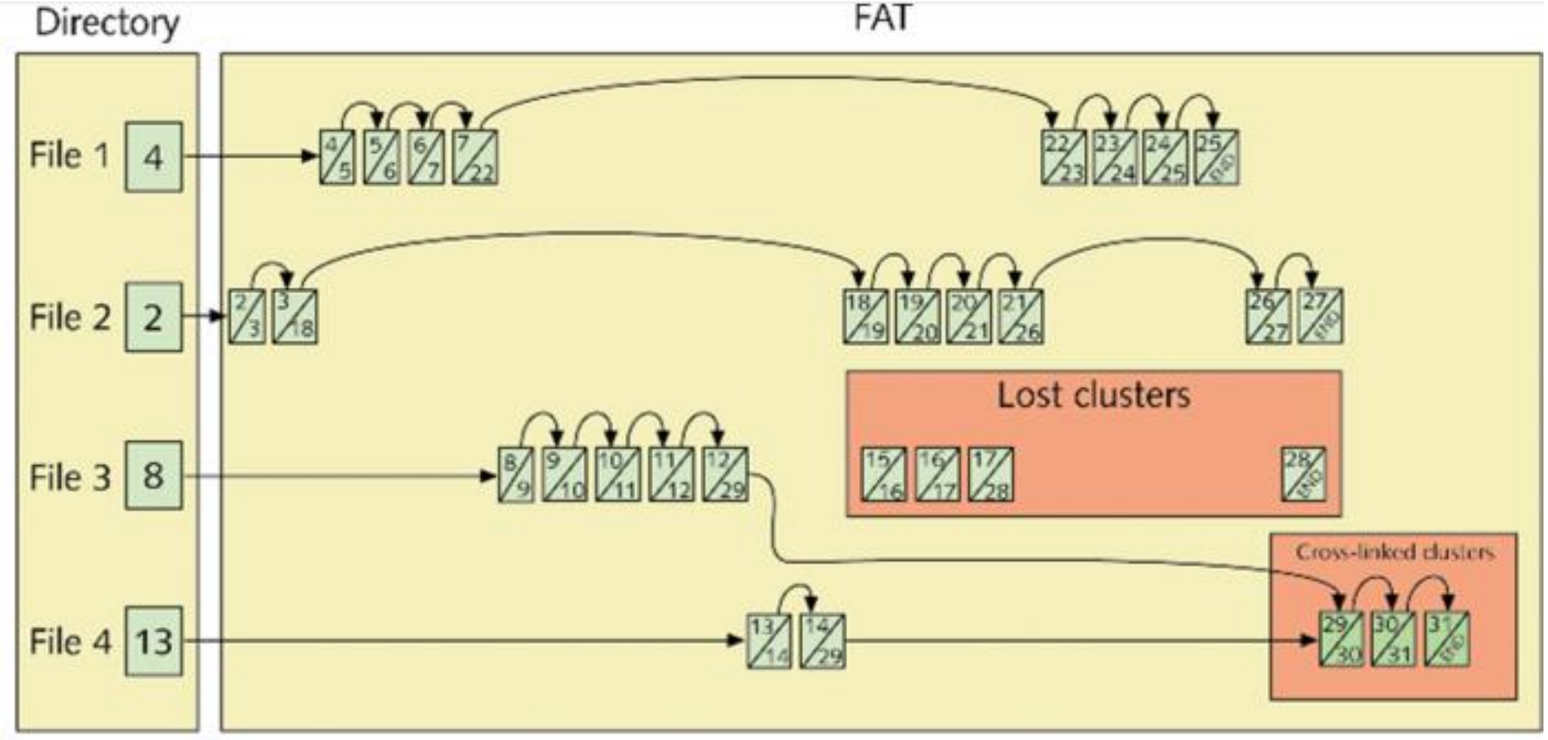
Inconsistency and Recovery

- A file operation involves multiple block modifications
  - To create a file in ext4 will need to modify: allocation bitmap, inode, directory, data block
  - What if power fails in the middle of file creation?
- Unwritten data/metadata are lost
  - Loss of metadata: structural inconsistency
  - Loss of user data: partially written file

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

Sutrctural Inconsistency Examples

- Ext file systems
  - A bitmap indicates that an inode has been allocated but the inode has not yet been written (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 yet been updated (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

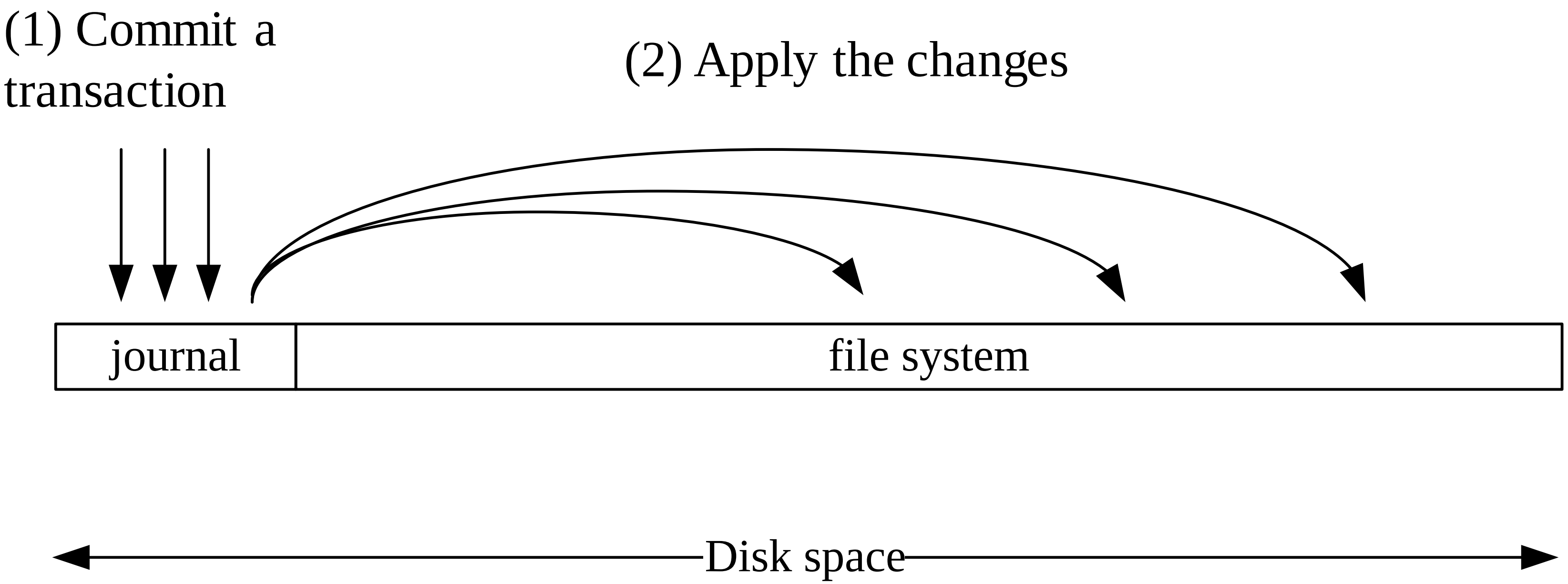
Journaling File Systems

- The root cause of file system inconsistency
  - A file operation, which involves to modify multiple disk blocks, is interrupted
- Transactions
  - An idea borrowed from database systems
  - A set of self-contained disk block modifications
- Protecting the file system against inconsistency
  - To guarantee the atomicity of file transactions
  - Atomicity: all are done or nothing is done (all or none)

Journaling File Systems

- Journaling file systems often employ Write-Ahead Logging, WAL, to guarantee the atomicity of transactions
- WAL requires a reserved space as the journal
- Two-step approach
  - The file system commits a transaction (to the journal)
  - The file system applies the changes (to the file system)

Write-Ahead Logging (WAL)



Crash Recovery with WAL

1. Scan the journal
2. Found a complete transaction → redo
3. Found a partial transaction → discard

- Transaction atomicity is thus guaranteed

Journaling File systems -- Summary

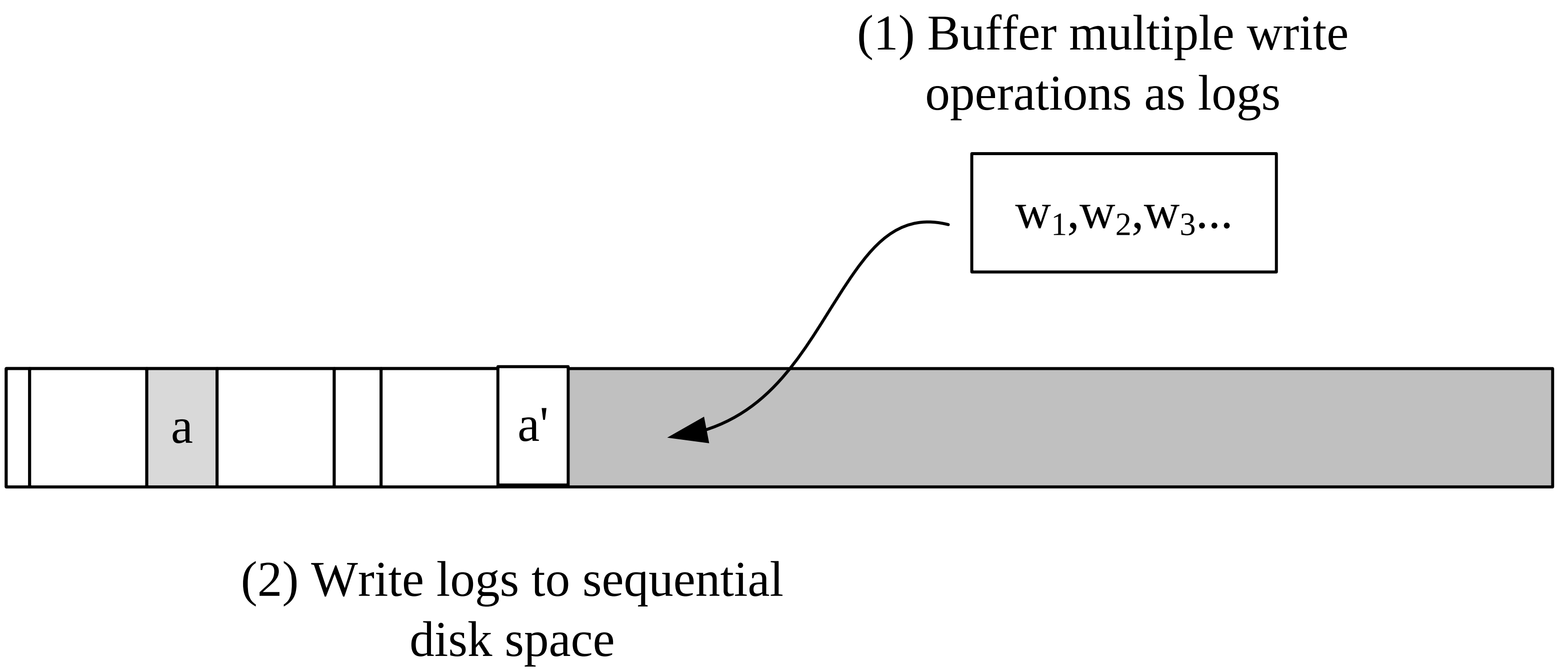
- Motivation
  - Preventing power interruptions from corrupting file systems
- Method
  - Creating a journal space for the file system
  - Collecting a set of self-contained writes as a transaction
  - Write transactions to the journal
  - Apply changes to the file system
  - On recovery, scan the disk journal. Re-do legit transactions; incomplete transactions will be discarded
- Benefit
  - Crash recovery is very fast
- Problem
  - Amplifying the write traffic

Log-Structured File System: sequential writing always

Log-Structured File Systems

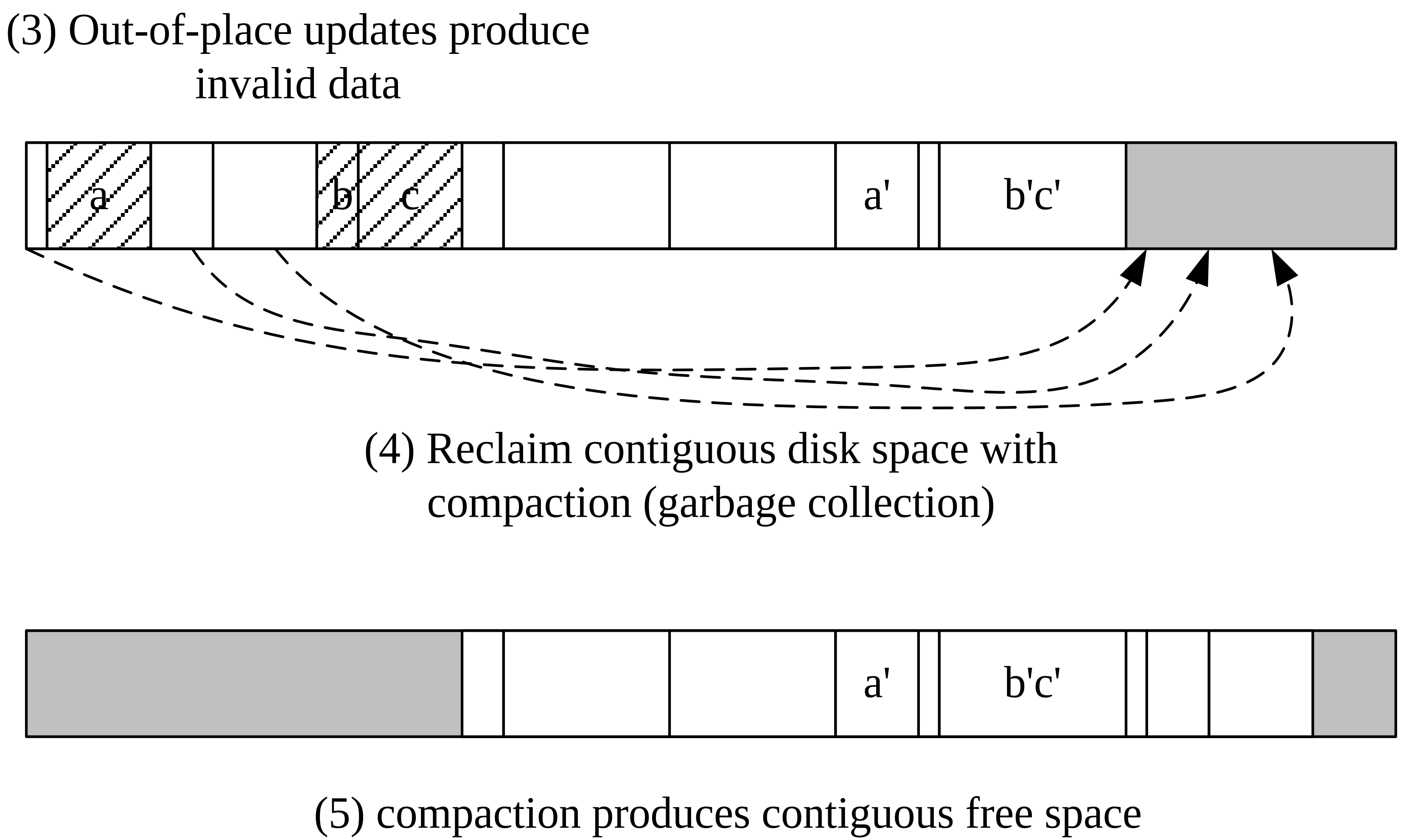
- Performance bottleneck of modern file systems
  - Read performance: not a problem with a large disk cache
  - Write performance: random writes are slow
- Key idea: out-of-place update
  - Random updates need not occur in place, they are converted to sequential writes
  - A log-structured file system can be imagined as a huge journal space without the “file system”
- Examples
  - NILFS2 (servers), F2FS (Android devices), NOVA (NVRAM)

The Concept of Log-Structured File Systems



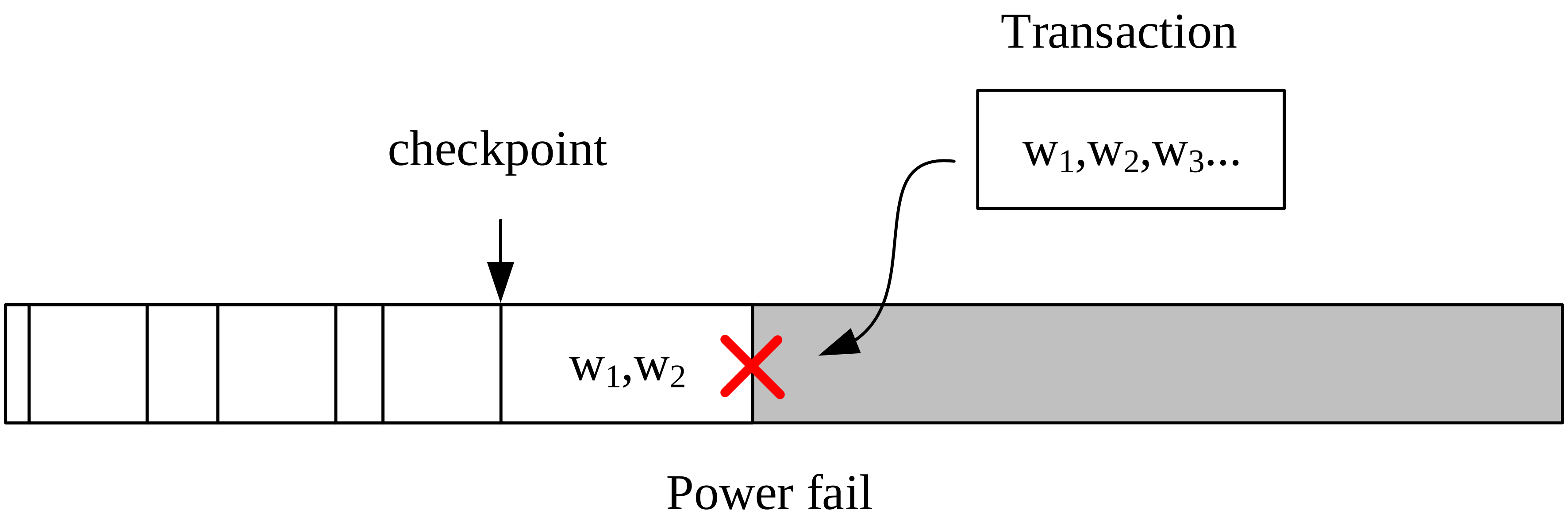
- Writes are always sequential and thus are highly efficient
- Out-of-place updates leaves garbage in the storage

LFS Cleaning



Also known as Compaction or Garbage Collection

LFS Checkpoint amd Recovery



- Recovery is surprisingly simple because writes are chronologically ordered

Log-Structured File Systems -- Summary

- Motivation:
  - RAM will be cheap and random reads are not a problem with a large page cache
  - Random writes must eventually hit the disk and they are slow
- Methods:
  - Collecting random writes (updates) into a long write burst
  - Out-of-place updates via sequential writing
- Benefits:
  - Great random write performance
  - Easy recovery
- Problems:
  - Need cleaning (i.e., compaction or garbage collection) to re-generate sequential space for new writes

End of Chapter 11