

Chapter 11: File System Implementation

Prof. Li-Pin Chang
National Chiao Tung University

Chapter 11: File System Implementation

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

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

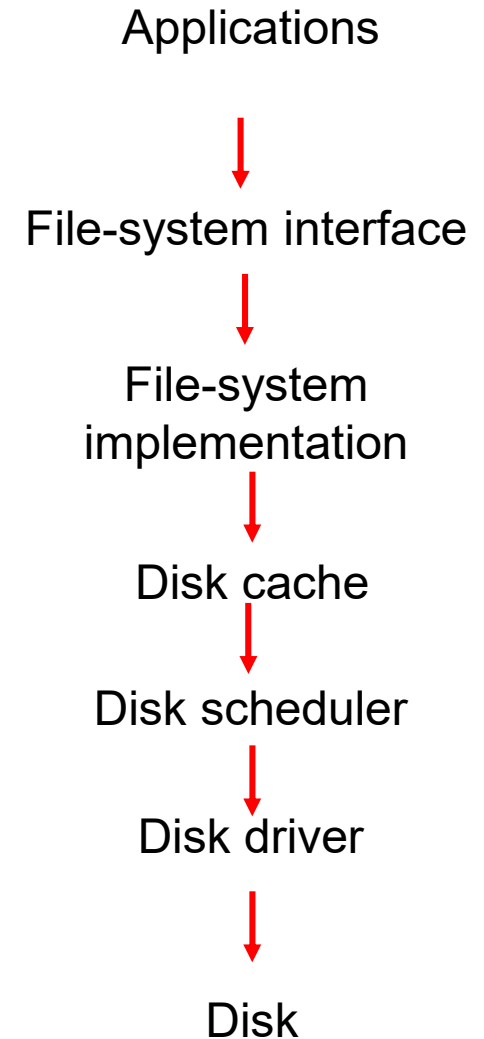
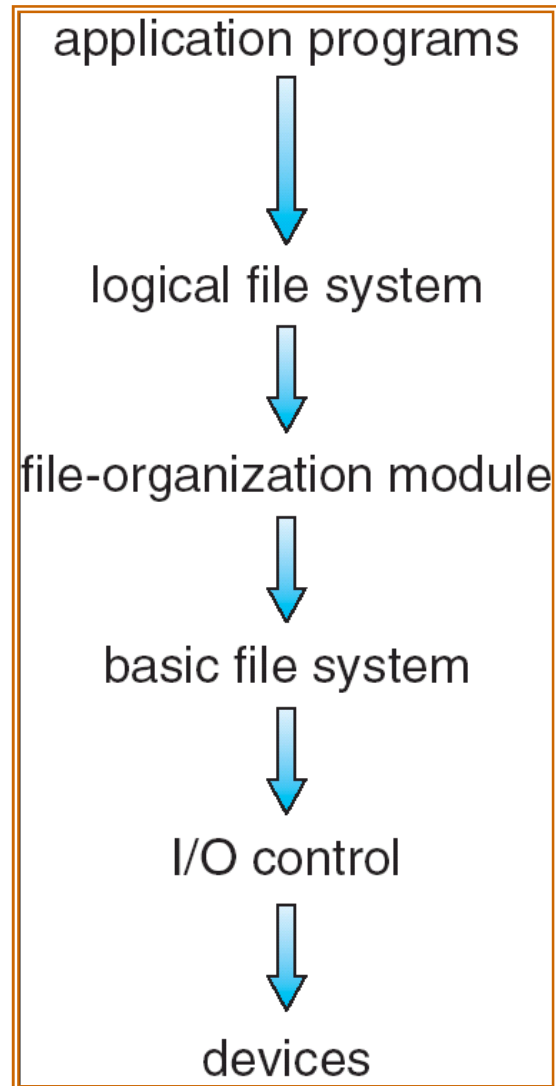
fread() / fwrite()

fs->read, fs->write

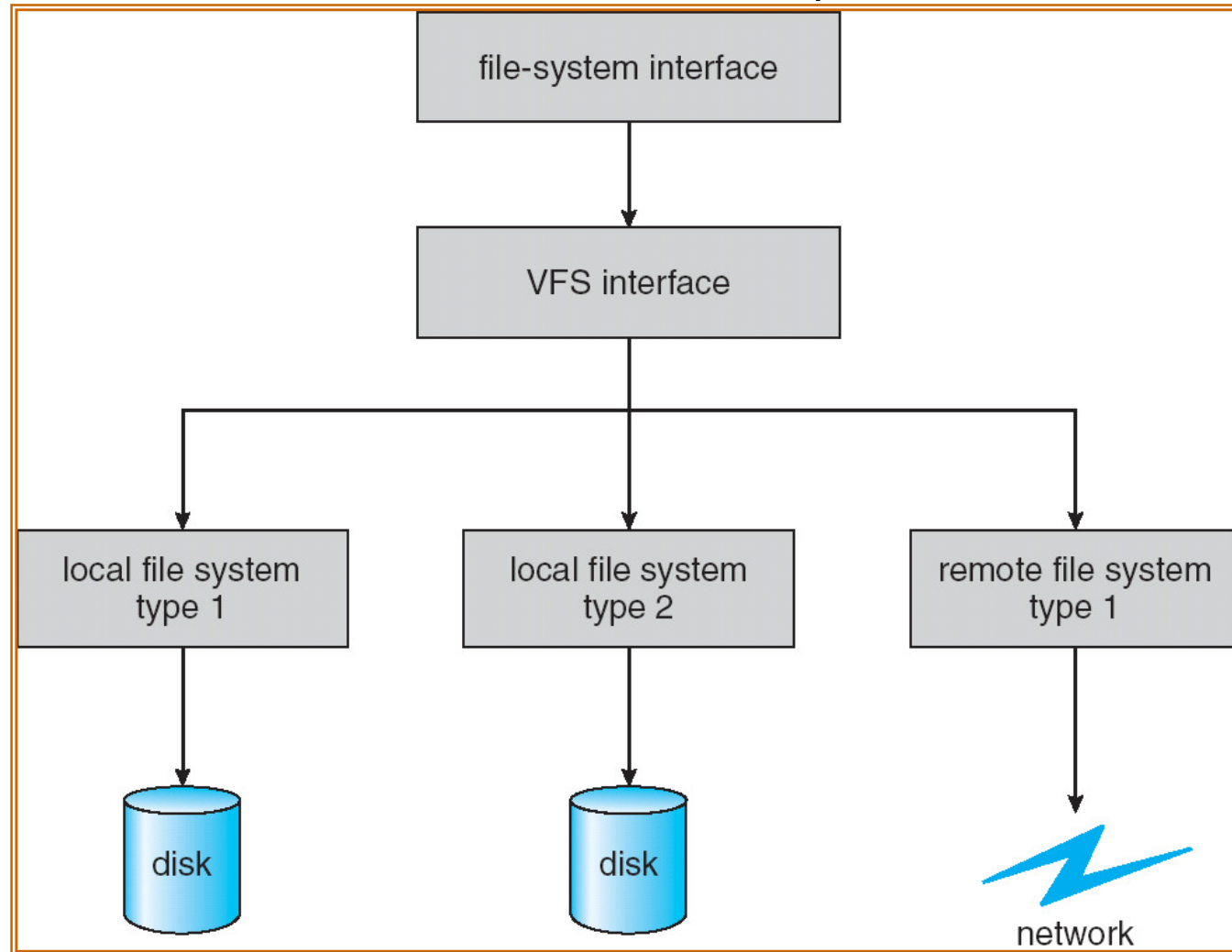
Read write to
Buffer/cache

Disk read/disk write

Control signals

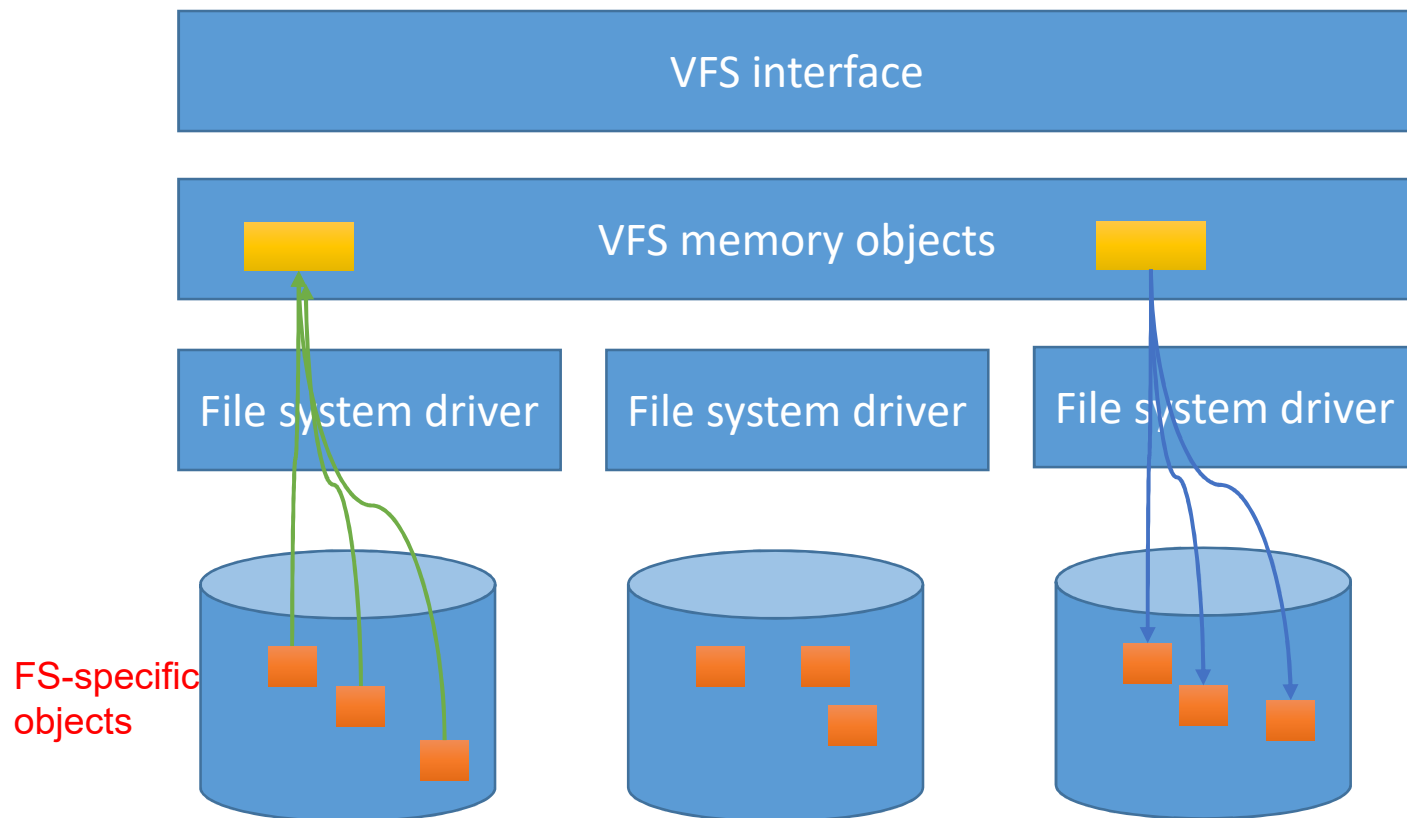


Schematic View of Virtual File System



Linux Virtual File System Architecture

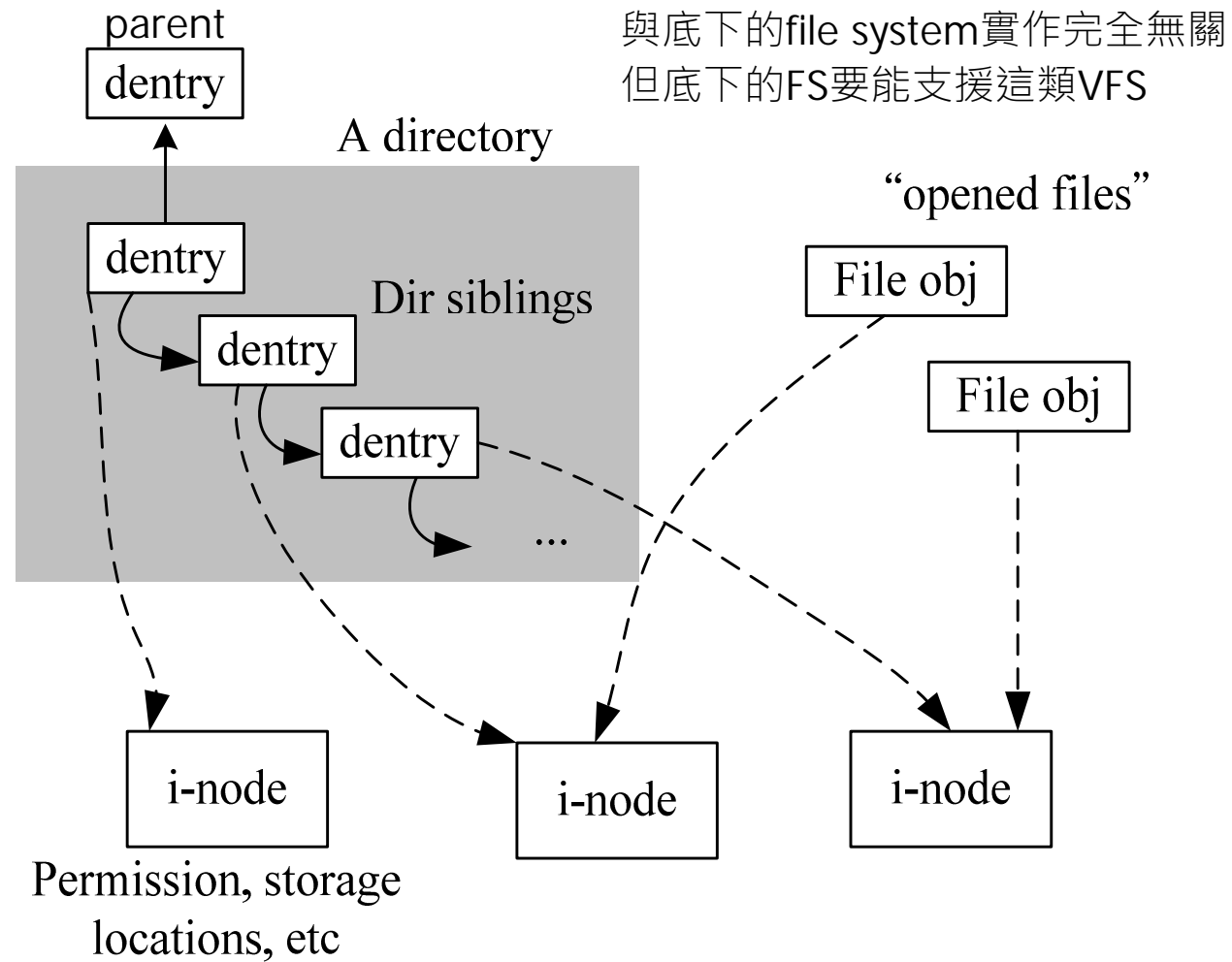
- File system drivers fill VFS objects with information in their disk data structures



In-memory Objects in Linux VFS

- Inode
 - Uniquely represent an individual file
- File object 開啟檔案後會產生一個file object，能有多個File object，但i-node只有一個
 - Represent an opened file, one for each fopen instance
- Superblock 表示filesystem的資訊：類型、可用空間、blocksize...
 - Represent the entire filesystem
- Dentry object
 - Represent an individual directory entry
- A collection of operations are defined on each type of object

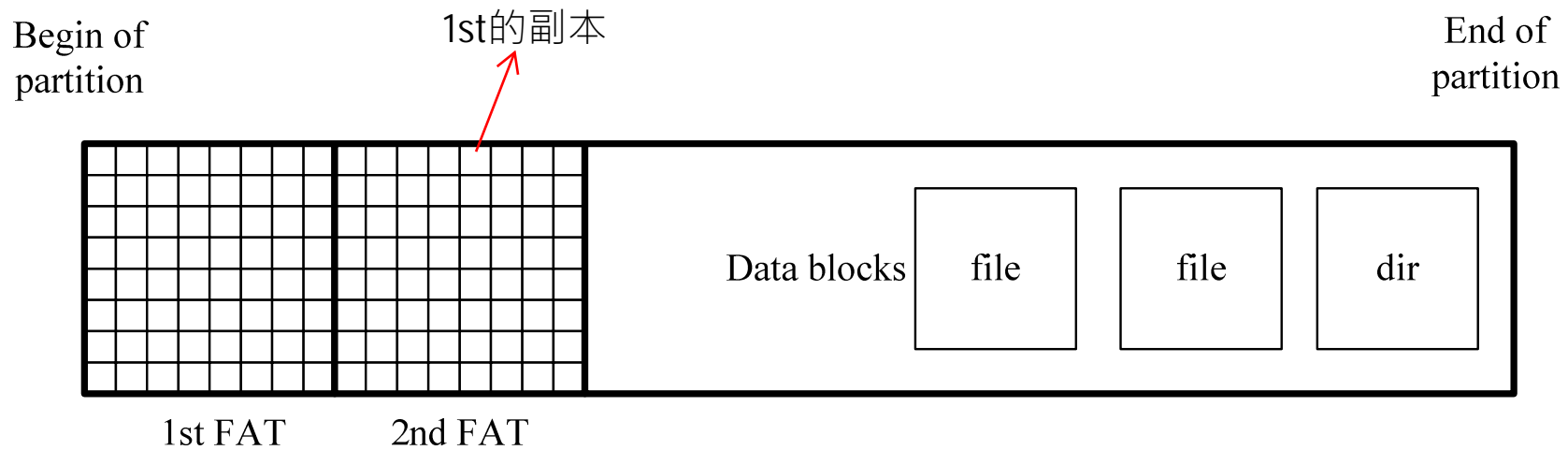
In-memory objects of Linux VFS



On-disk data structures

- File-system-specific
- Linux ext file system 跟unix kernel一起開發的，所以很像
 - 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
VFS會丟一個資料結構叫filesystem填→filesystem要與VFS相容!

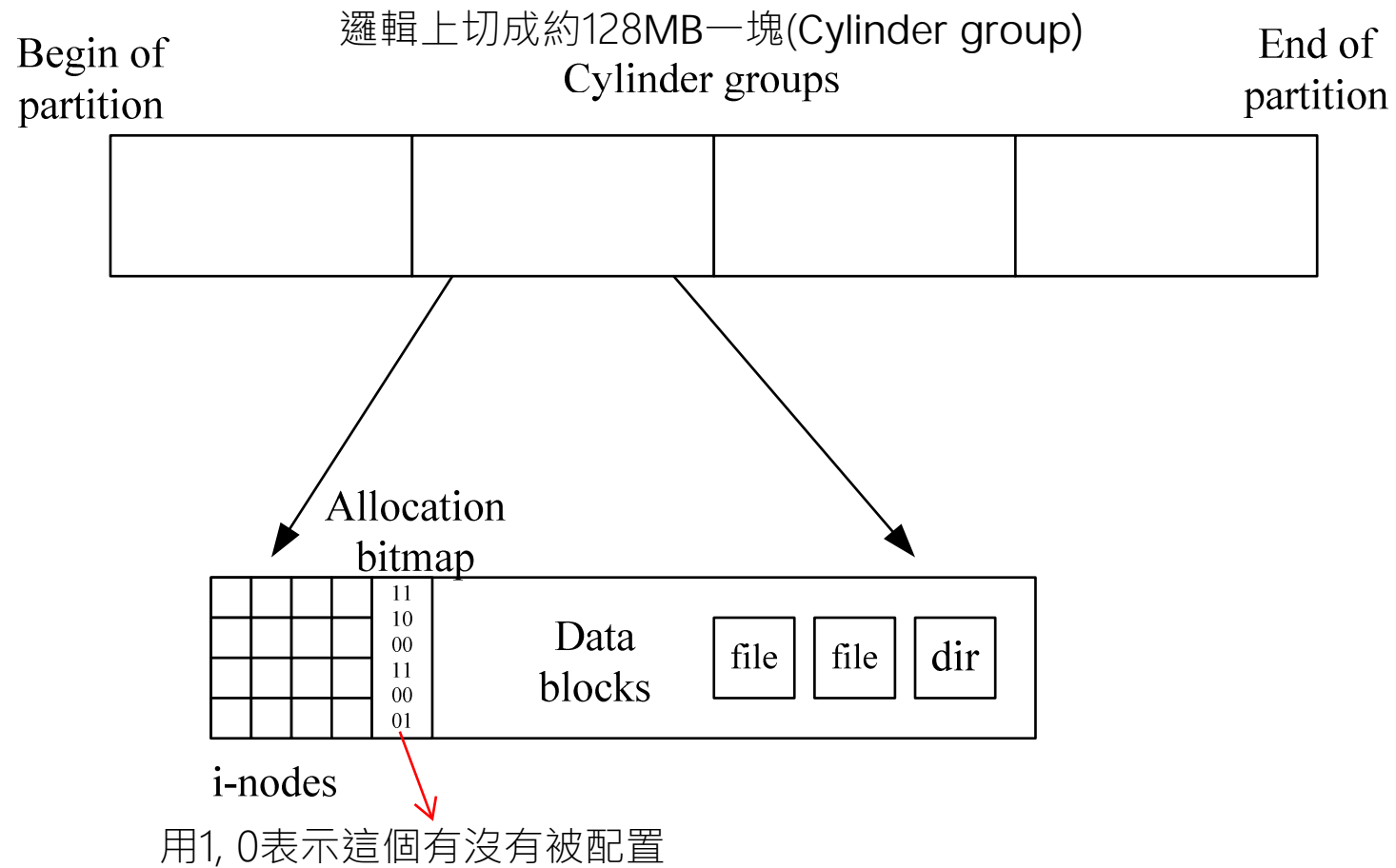
Disk layout of FAT 12/16/32 file systems



存的是link-list

裡面都是pointer只到後面的datablock(平行對應)

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



FileSystem最基本的重點

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

Directory Implementation

資料夾內如何實作

- **Linear list** of file names with pointer to the data blocks.
 - simple design
 - time-consuming operations dir內檔案很多就開很慢，要一個接一個去找
 - FAT file system
- **Hash Table** – linear list with hash data structure.
 - Efficient search
 - **collisions** – situations where two file names hash to the same location
 - fixed size
- **B-trees (or variants)** 每一個dir內就是b-tree，以name當key
 - Efficient search, variable size
 - XFS, NTFS, ext4
 - For very large file systems

Allocation Methods

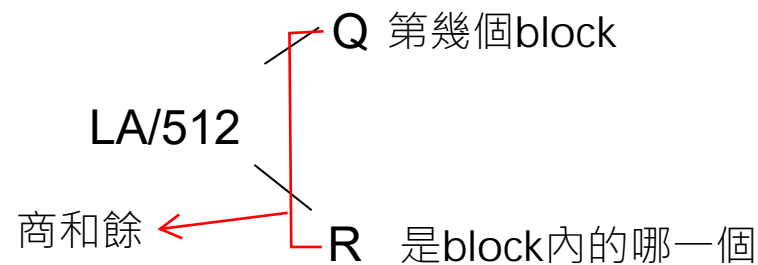
- An allocation method refers to how disk blocks are allocated for files:
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

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) 會有碎片
 - File deletion leaves holes (external fragmentation) in the file system
 - Needs compaction, maybe done in background or downtime
常常要搬來搬去來找到足夠的連續空間

Contiguous Allocation

- Mapping from logical to physical



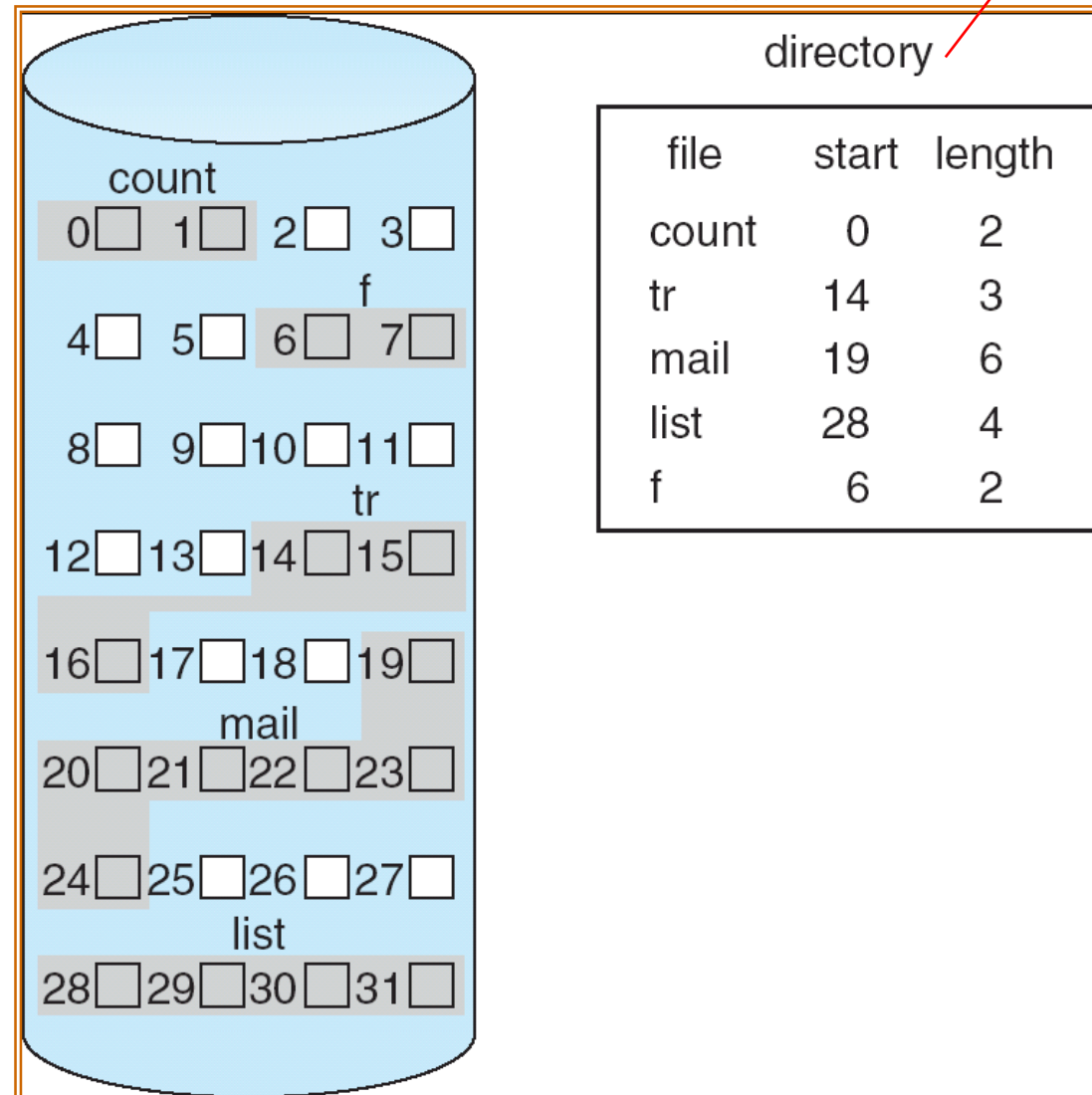
Block to be accessed = Q + starting address (block)

Displacement into block = R

LA=byte address
1 block=512B

Contiguous Allocation of Disk Space

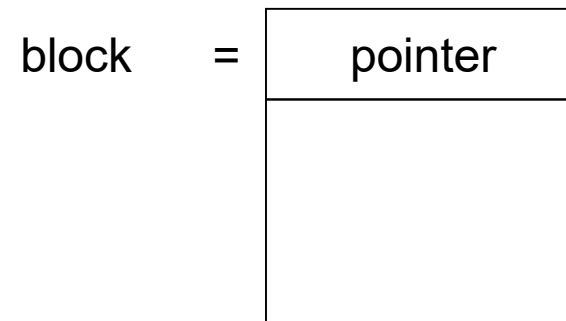
也會放在磁碟的某個地方



Linked Allocation

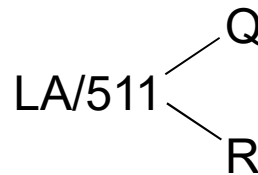
用link-list的方式來存data

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



Linked Allocation (Cont.)

- Simple – need only starting address 基本上不怕fragmentation
但位置會比較分散，讀寫頭要跑來跑去
→效能較差
- Free-space management system
 - no waste of space (no external fragmentation)
 - No random access (need to traverse the linked blocks) 無法隨意access某塊block
- Mapping



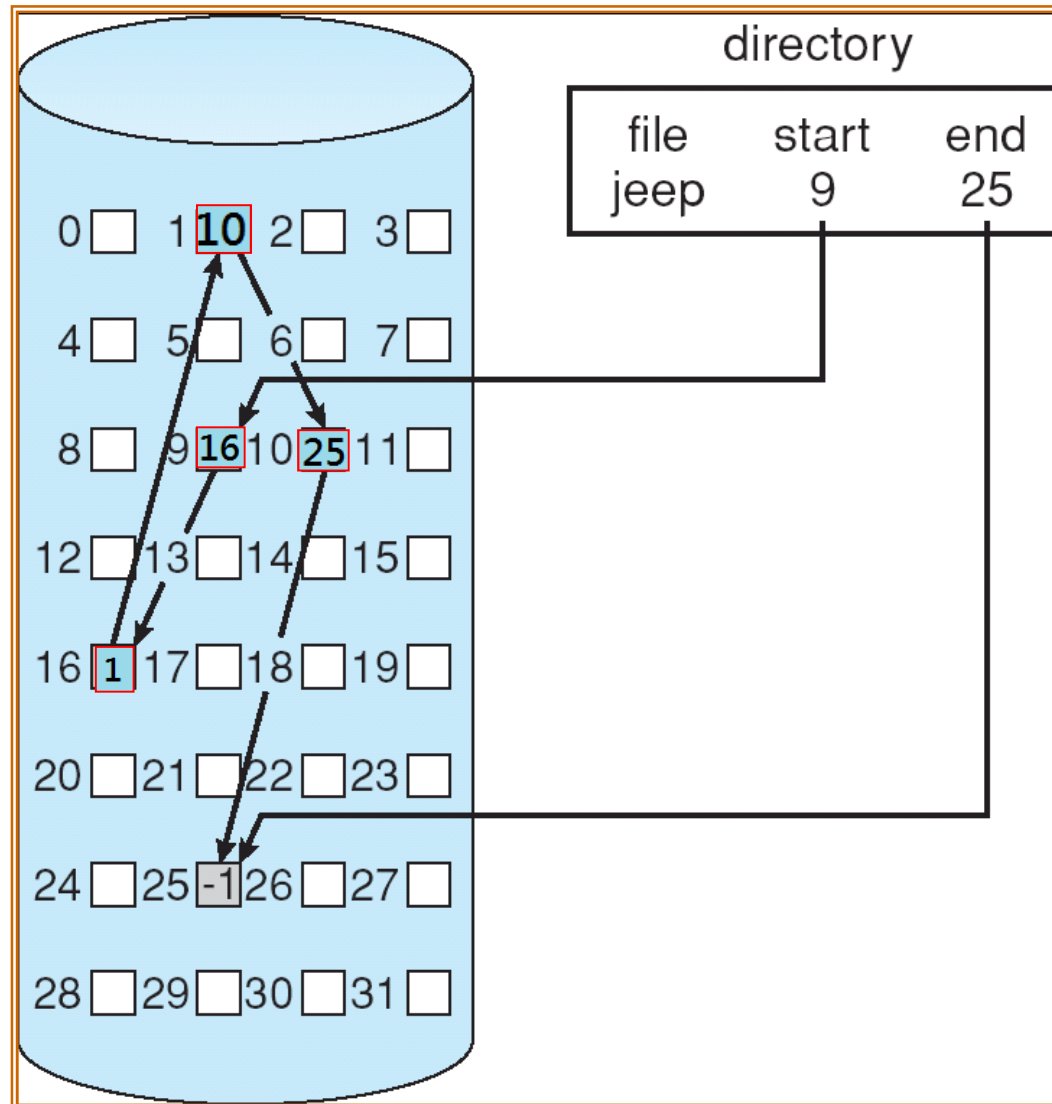
1 data block = 512B, 1 for ptr,
So 511B for user data

Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = $R + 1$ (the 0th byte is for pointer)

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

Linked Allocation



優點：只要有空間就能用，
不用管破碎問題
缺點：檔案分布破碎，讀取
效率不佳

Linked Allocation

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

FAT : File Allocation Table

The layout of FAT 12/16/32 file system

把partition割成 2^{16} 個cluster
以cluster為配置單位

很浪費空間

1G disk $\rightarrow 2^{30}$ bytes

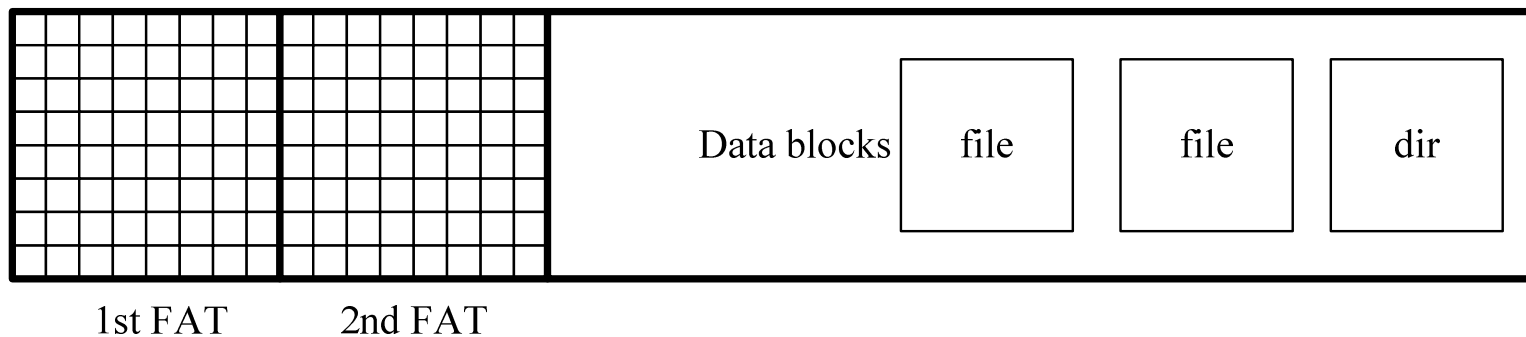
$2^{14} = 16\text{KB}$ per cluster in FAT16

2KB的檔案依舊要佔到16KB

\rightarrow internal fragmentation

Begin of
partition

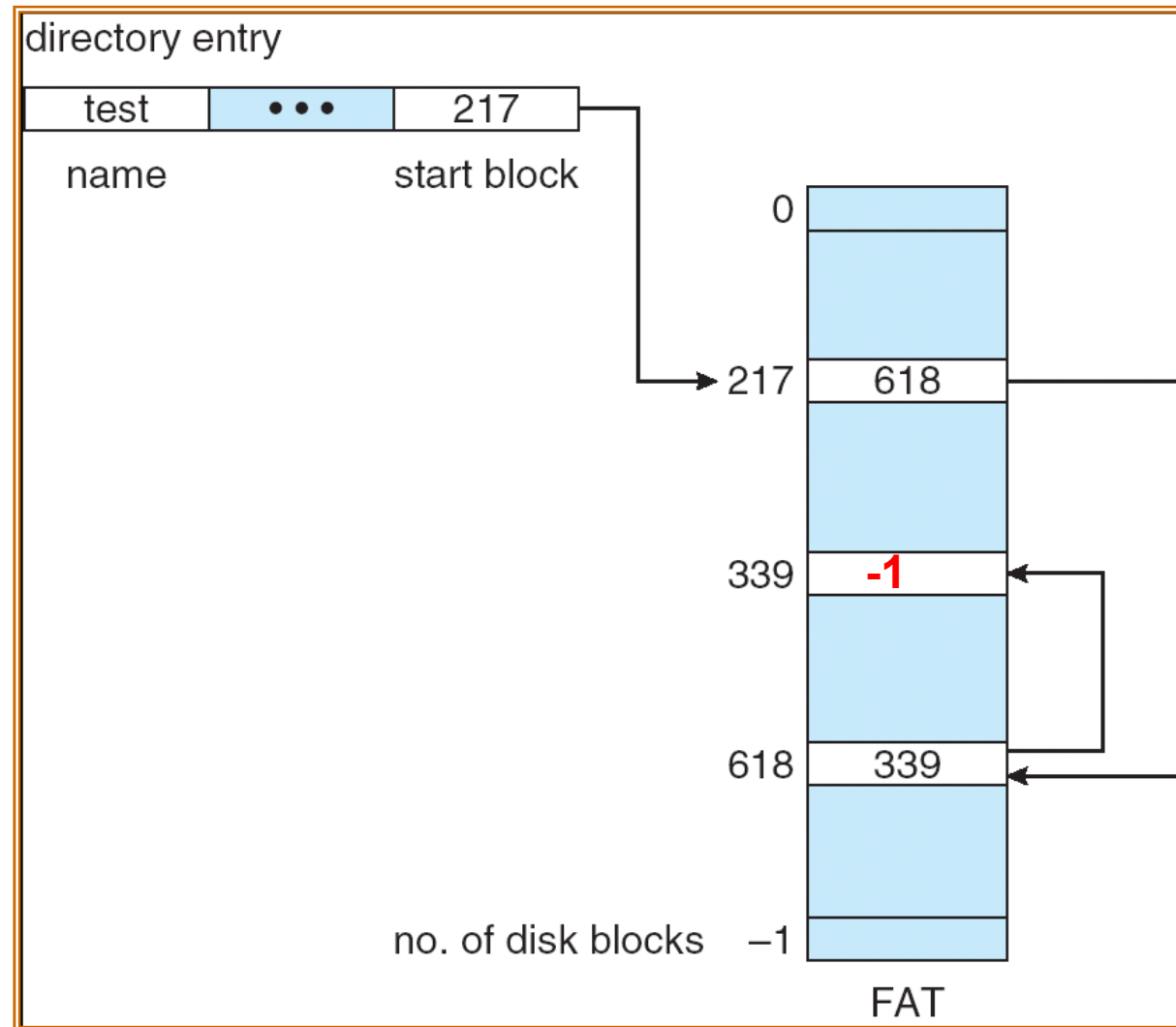
End of
partition



FAT12 \rightarrow 16 \rightarrow 32

為了避免越來越嚴重的internal fragmentation

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 is a read/write bottleneck

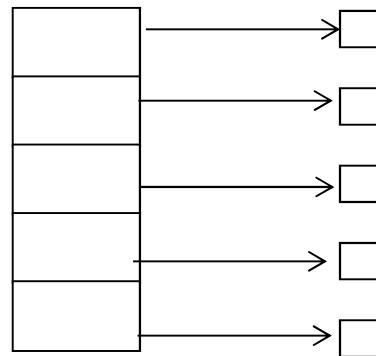
要找下一個data block磁頭必須轉到megadata那塊去讀資料
- FAT table can also be a single –point-of-failure
 - Two identical copies

這部分有壞掉整個FAT系統就GG

透過多複製一份來保障FAT table
出錯也不會出大事

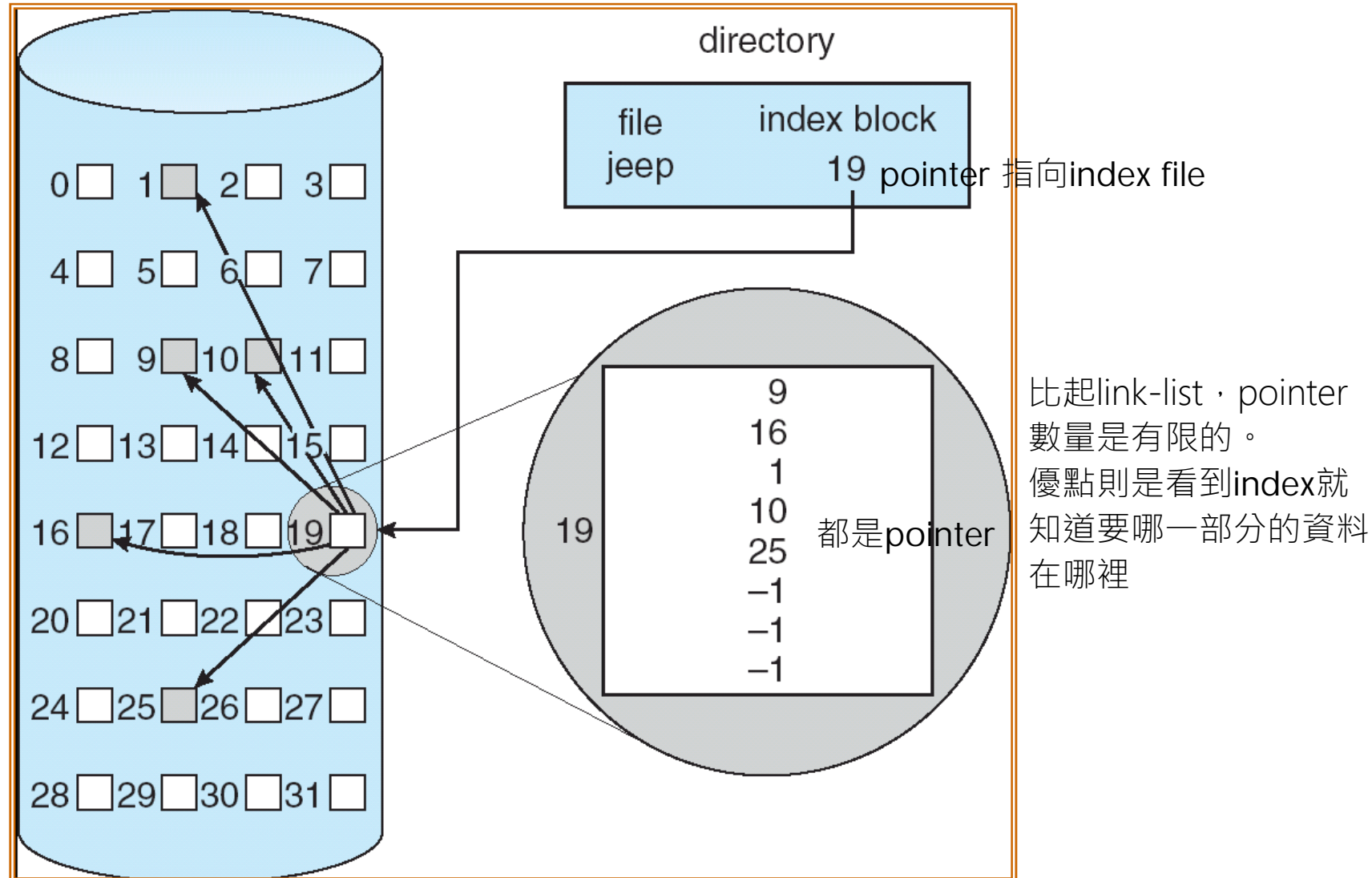
Indexed Allocation

- Brings all pointers together into the index block.
- **Logical view.** 把所有pointer塞在一個index-block裡面
減少尋找block所需的disk access數



index table

Example of Indexed Allocation



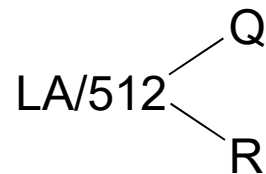
Indexed Allocation (Cont.)

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

不用管空間連續性的問題



但index block自身要佔一些空間

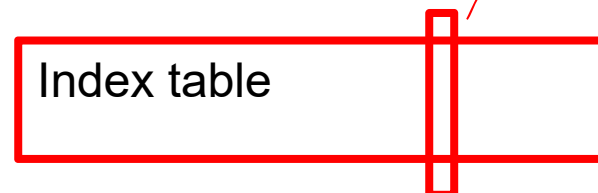


LA=byte address
A block is of 512B

Q = displacement into index table (entry #)

R = displacement into block

•



De-reference



Indexed Allocation – Mapping

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

層數多了，要讀一塊data至少要查兩層index
→每層都要一次disk access(5~10ms很貴)

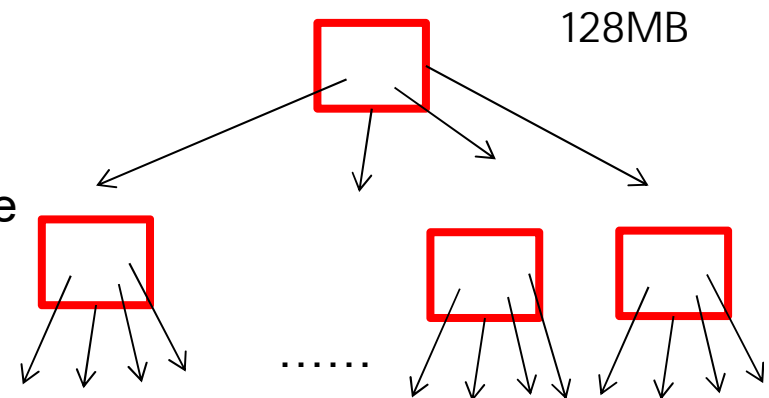
$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

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

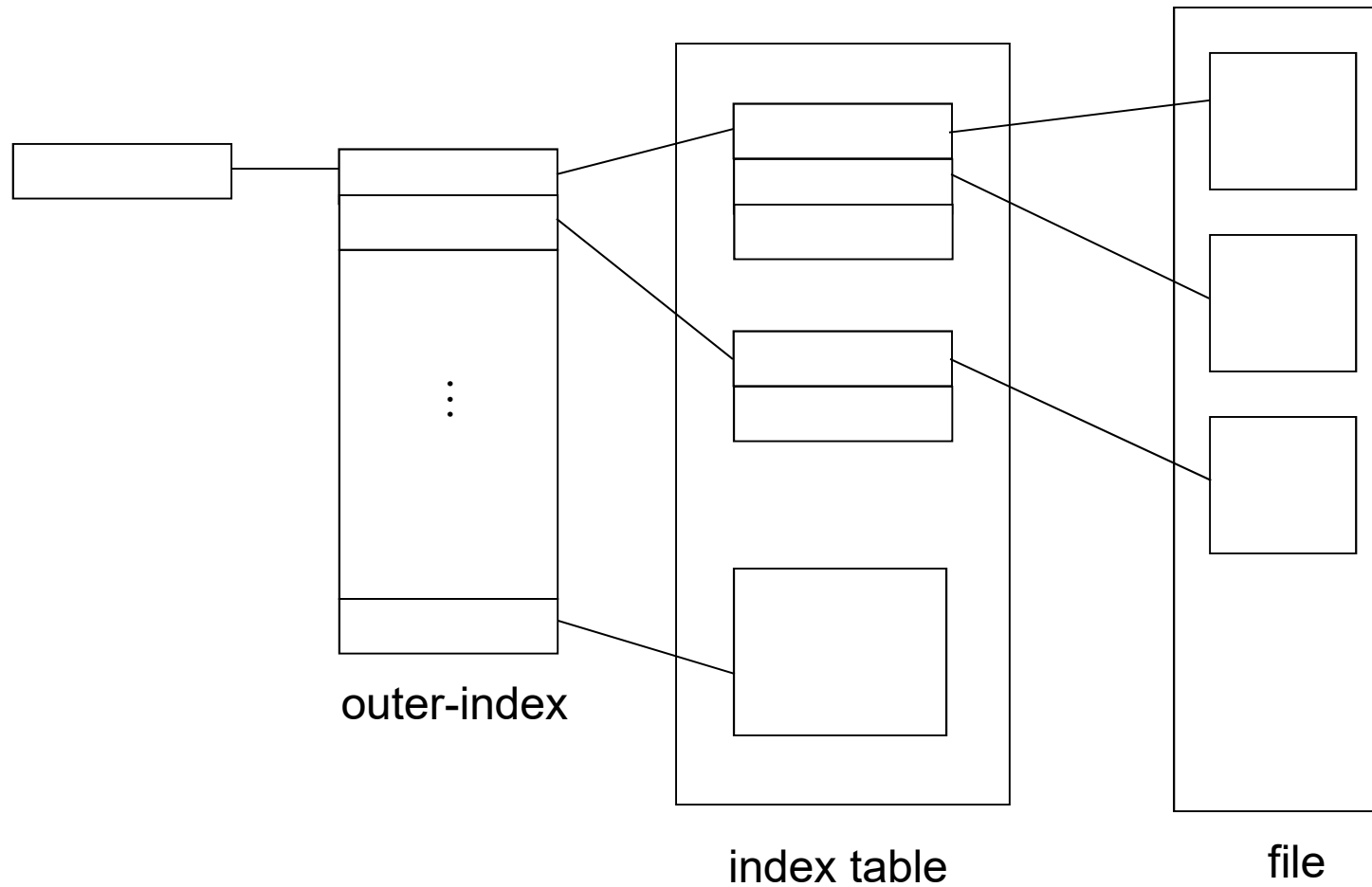
$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

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

- **1st level: pointers to index tables**
- **2nd level: pointers to data blocks**
- 1block=512B, 1ptr=1B
- 1 index block has 512 ptr,
- Total addressable size=512*512*512 bytes

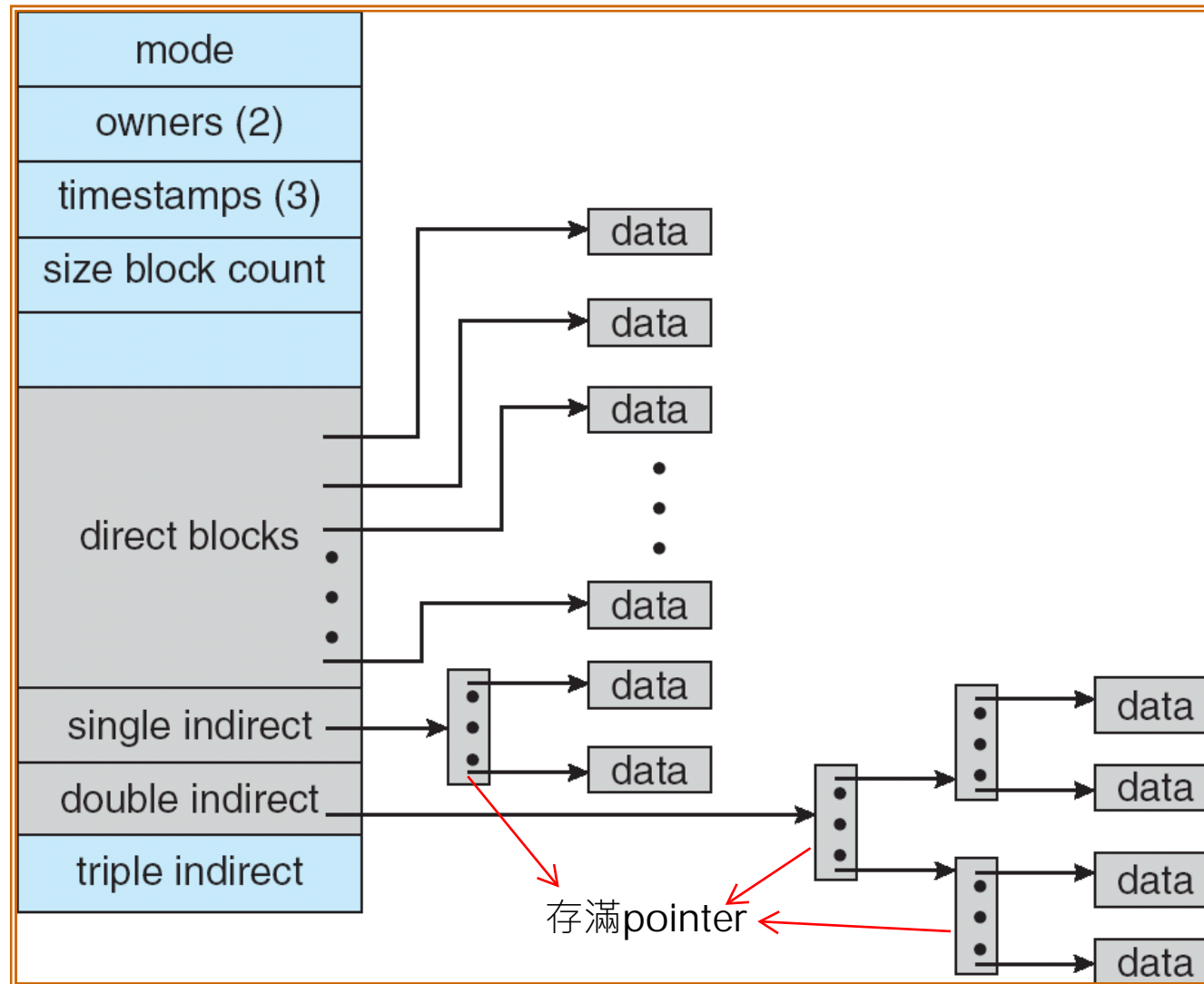


Indexed Allocation – Mapping (Cont.)



UNIX inode

An i-node. Small files use only direct blocks



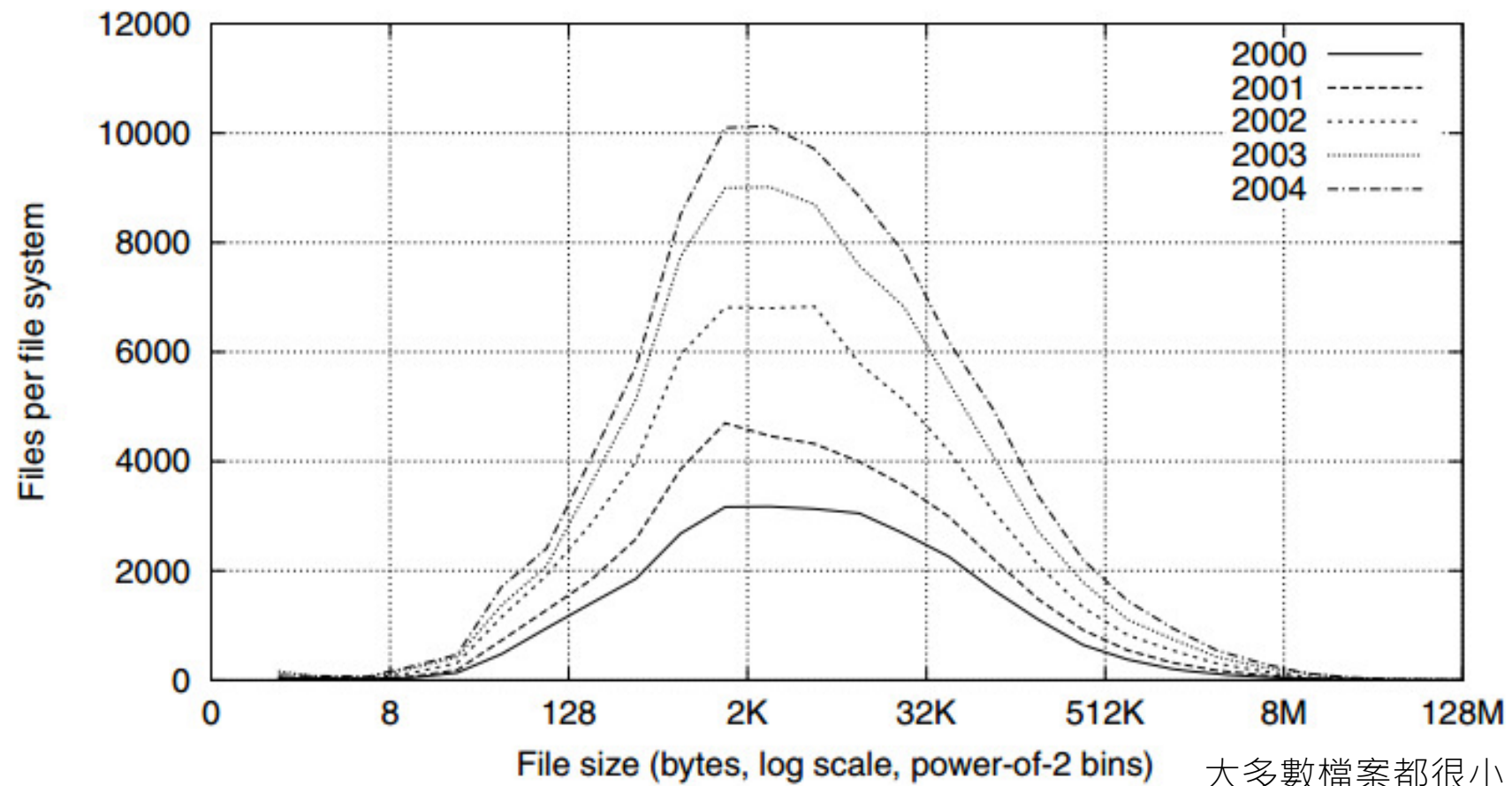


Fig. 2. Histograms of files by size.

大多數檔案都很小
→對小檔案做最佳化更有效率

[A. Agrawal, "A Five-Year Study of File-System Metadata"](#)

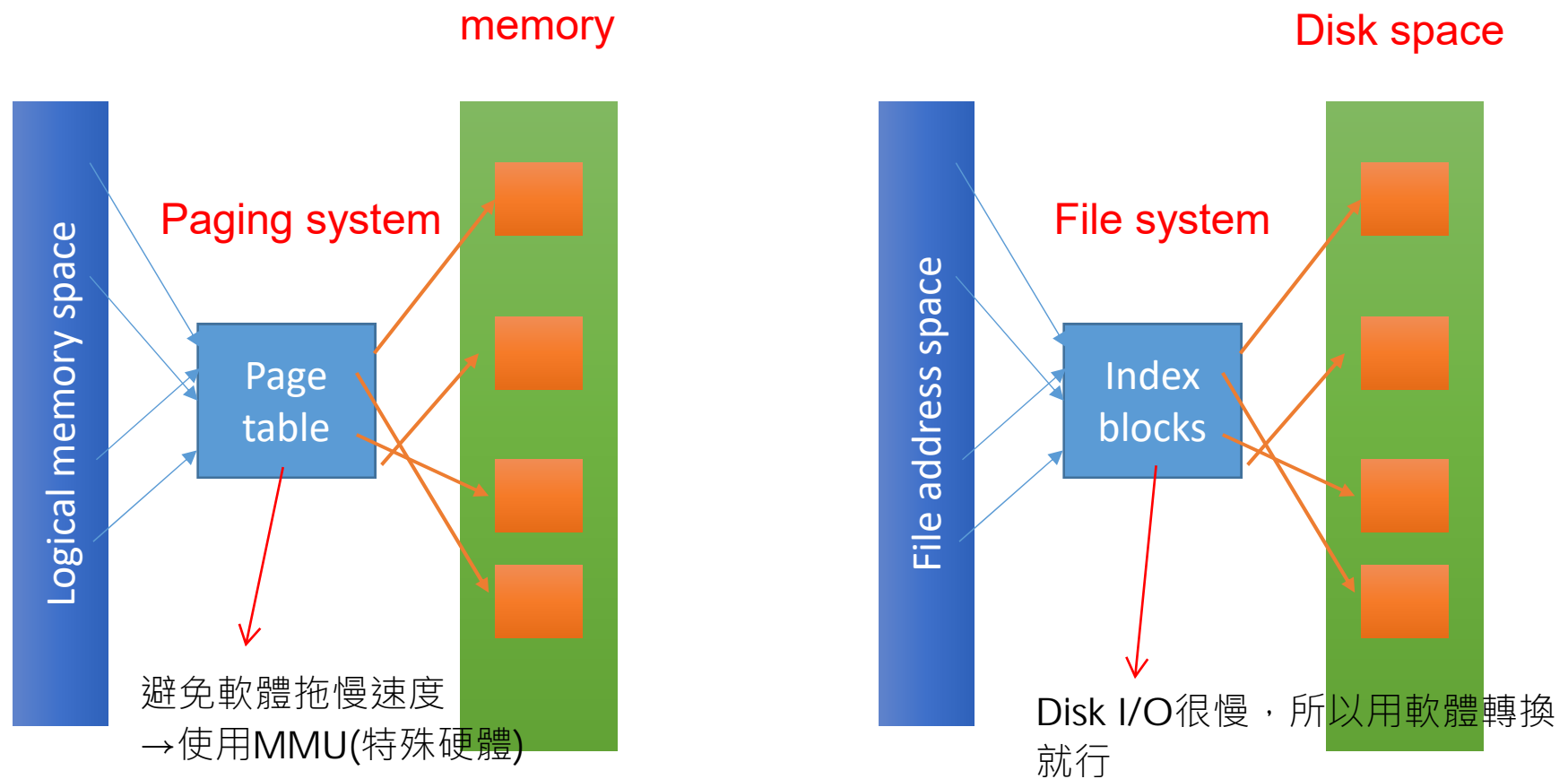
連續

Extent-Based Systems

盡量找連續的空間，預測一下file成長的趨勢來看要配給多少空間。如果檔案持續成長超過配置→將多的部分分配到另一塊連續空間。

- 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

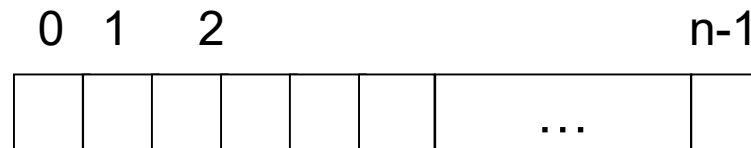
Indirection, indirection, indirection ...



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

Free-Space Management

- Bit vector (n blocks)



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

bit map要 $O(n)$ 來找free block→慢
link-list management則是 $O(1)$

bit map可以輕鬆發現連續space
link-list則是完全沒辦法做到(幾乎
看不出來有沒有連續)

Block number calculation 如何快速抓第一個free的block

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

double word(32bits)

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

應該是 (number of all-**1**-value words) +
offset of first **0** bit

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)

- Easy to get contiguous files

- Check whether a DWORD is zero (0x00000000)

Free-Space Management (Cont.)

- Need to protect:
 - Pointer to free list
 - Bit map
 - Must be kept on disk
 - Copy in memory and disk may differ
 - Cannot allow for block[i] to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk
 - (power-cycling issue) 斷電重啟的問題
- Solution:
 - Set bit[i] = 1 in disk 這樣即使斷電，這塊block如果寫了東西也不會
 - Allocate block[i] 被當作不存在
 - Set bit[i] = 1 in memory

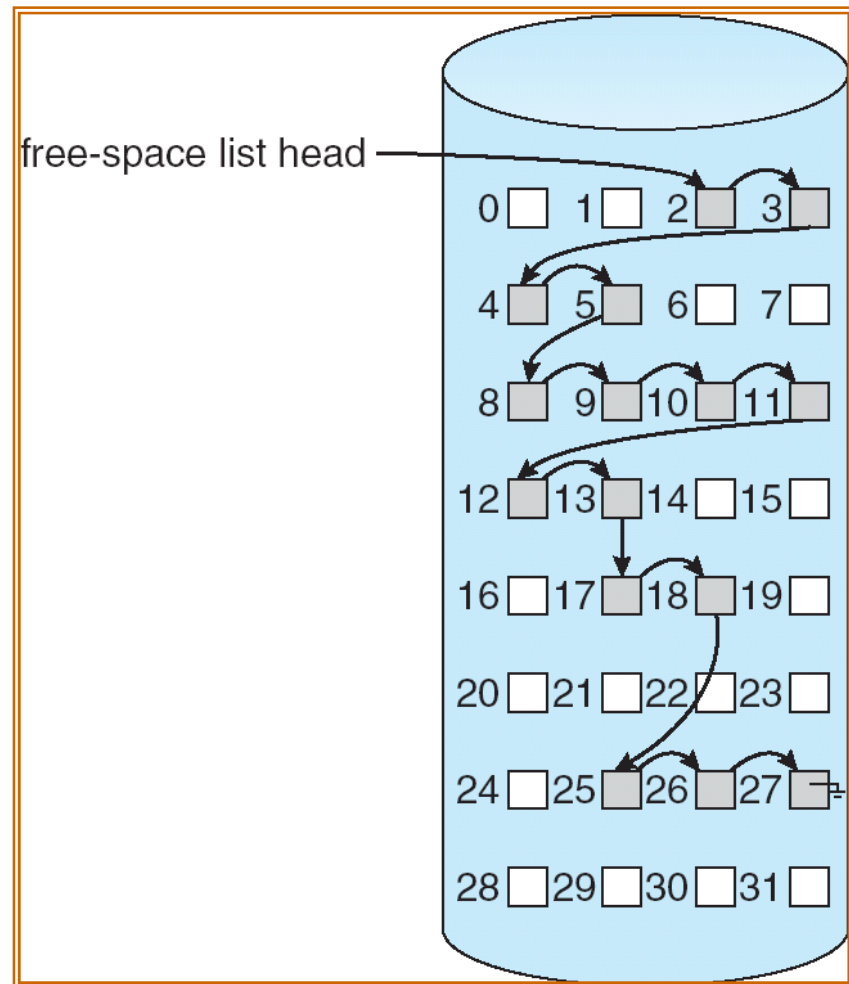
$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Linked Free Space List on Disk

用link-串起所有free spacelist

- Easy to allocate and deallocate free blocks
- No waste of free space
- But cannot get contiguous space easily, prone to fragmentation

很容易讓檔案碎開來



File Fragmentation

用久了之後檔案會慢慢變的破碎

- 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


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

計算Tool : filefrag(linux)

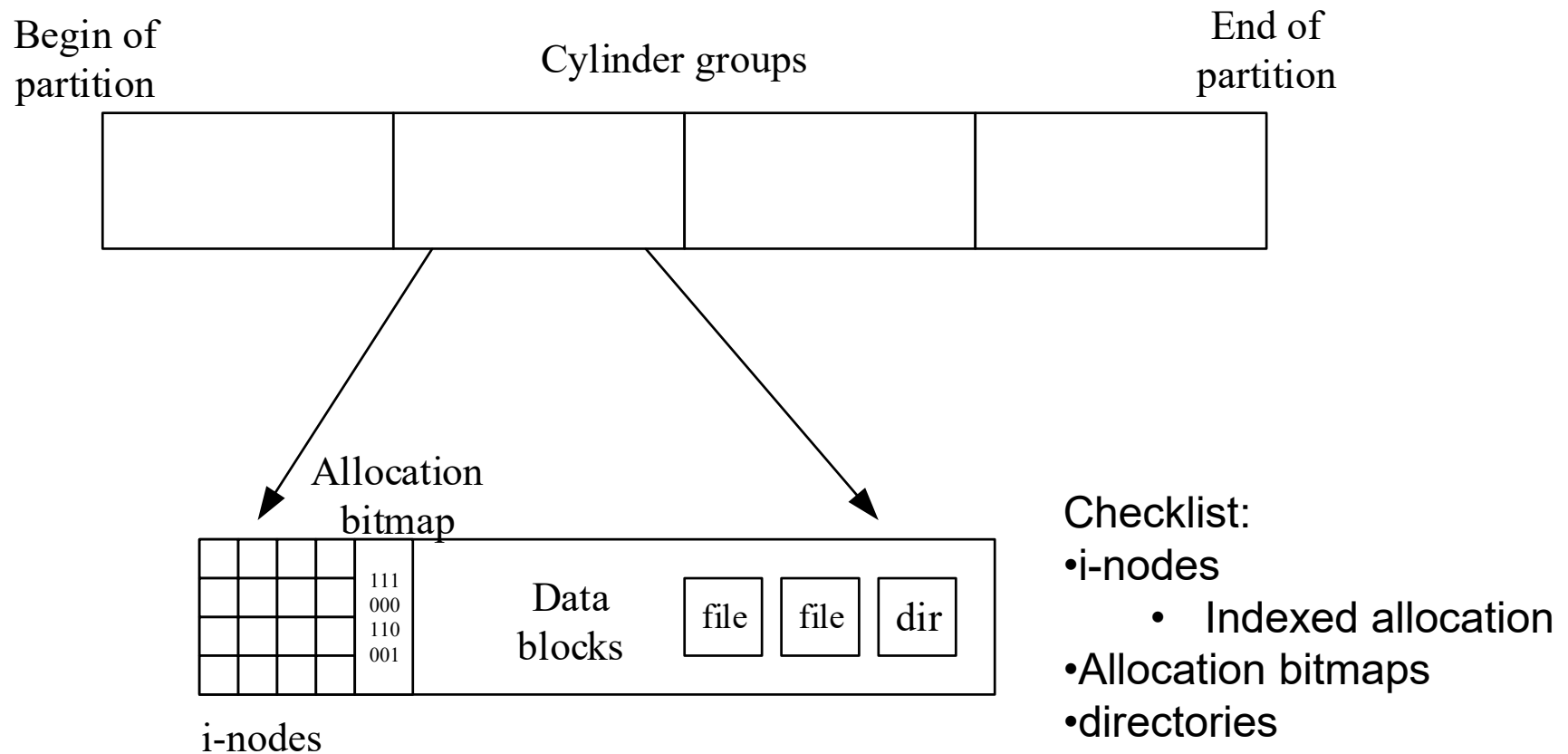
- The higher the DoF of a file is, the more disk seeks are required to access the file

解方 : (defragmentation)磁碟重組

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  extent allocation
 - Linked list: FAT
 - Bitmap: Ext

Review: ext4 file system



切割成Cylinder就能讓megadata較靠近datablocks

Efficiency and Performance

- Optimizations for file systems
 - disk allocation algorithms
 - directory algorithms
 - Free space management
- Optimizations taken by existing file systems 對小file很方便
 - UNIX file systems put a file's inode near to its data blocks
 - ReiserFS embeds small files into directories
 - Ext4 uses extents for taking advantage of sequential disk accesses

Efficiency and Performance

- Optimizations independent of file systems 與FS類型無關的最佳化
 - disk cache – separate section of main memory for frequently used blocks (temporal locality)
 - read-ahead – technique to optimize sequential access
 - Exploting spatial locality of file access 讀取某段資料時，會把附近的也一起讀
基於spatial locality
 - Like pre-paging
 - improve PC performance by dedicating section of memory as virtual disk, or RAM disk 奉獻

Efficiency and Performance

- Asynchronous writes
 - Written data are marked as dirty, the writing process immediately returns
 - A daemon writes back dirty disk blocks at proper timing with the most efficient means
- Synchronous writes
 - Writes to safety-critical data, such as file-system metadata or database metadata
 - `fsync()` or `open()` with `O_SYNC`

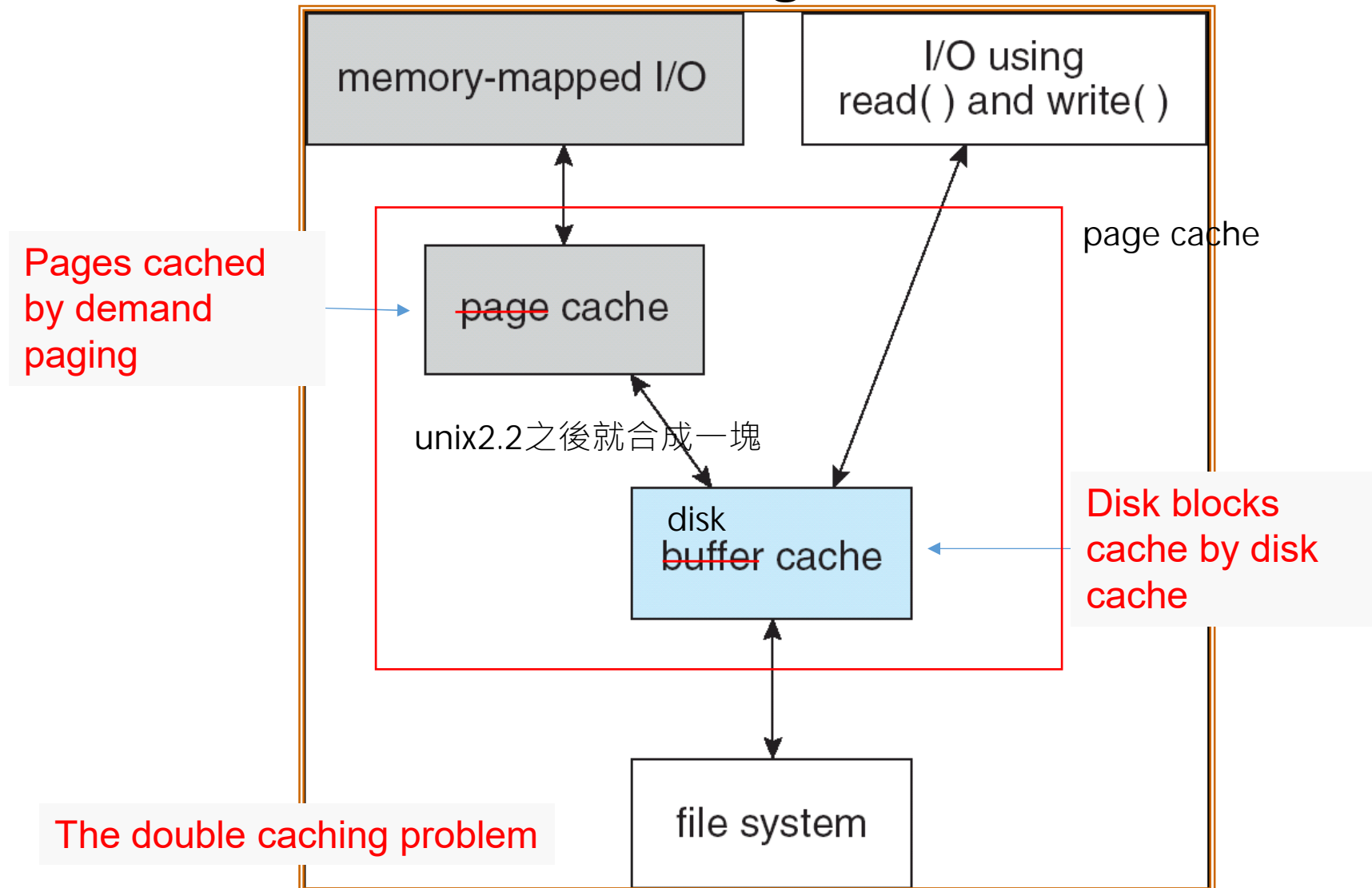
Paging : virtual memory→physical memory
如果不常用→swap out

Page Cache

用page的方式來cache blocks

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O (files) uses a page cache
- **But** routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

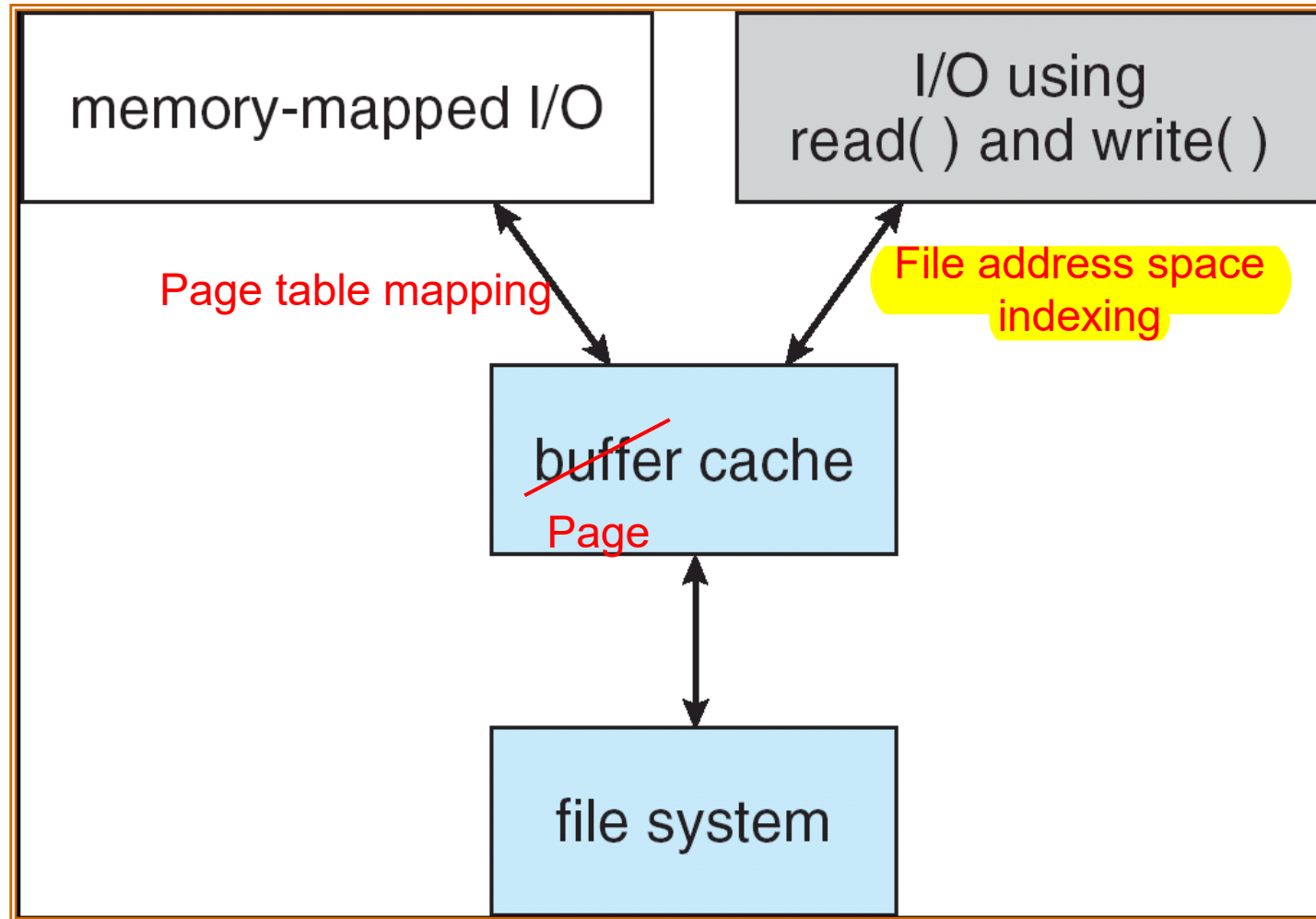
I/O Without a Unified Page Cache



Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

I/O Using a Unified Page Cache



Recovery

- A file operation modifies multiple blocks — 斷電就很容易造成一堆不一致
 - 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

Recovery

- Metadata loss: Structural inconsistency
- 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 可能file被刪除時(count 0) · hard link還存在
- 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)
- User data loss: file contents are lost

Recovery

開機掛載時設成dirty，正常關機時設成clean

- 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
漫長的

Journaling File Systems

inconsistent的原因是file operation改到一半就被斷電

- Guarantee the atomicity of file system operations
 - All or none, structural consistency
- Journaling file systems collect the dirty data produced by a set of (completed) file-system operations into a transaction 一組一組做比較有效率
 - Journal is a reserved disk space 會先寫進journal space(on disk)
 - Write transactions to the journal first, and then modify the file system
- 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) 耐用的
- All or none (no partial)

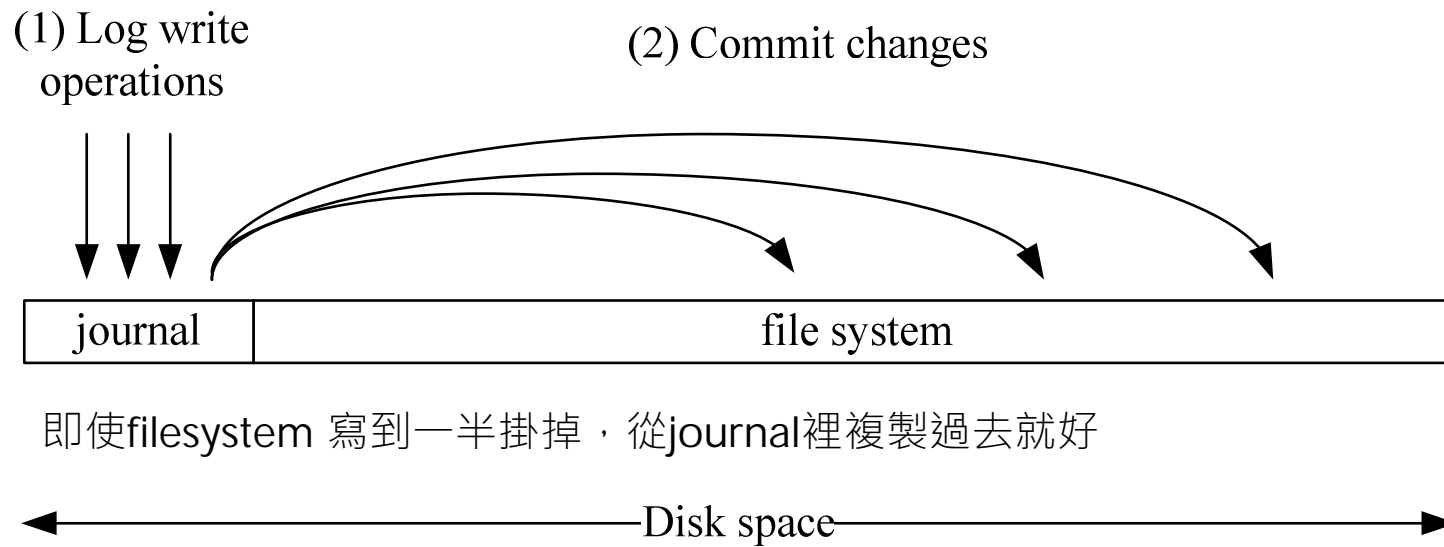
- Journaling, as known as write-ahead logging (WAL)

- 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 無法保護的資料不損失，目的是保證結構的完整
 - Preventing power interruptions from corrupting file systems
- Method
 - Add a journal to the file system (can be a file or a preserved space)
 - Collect a series of write operations 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 the file system
- Problem
 - Degraded performance → amplifying write traffic
有些FS會只journal metadata

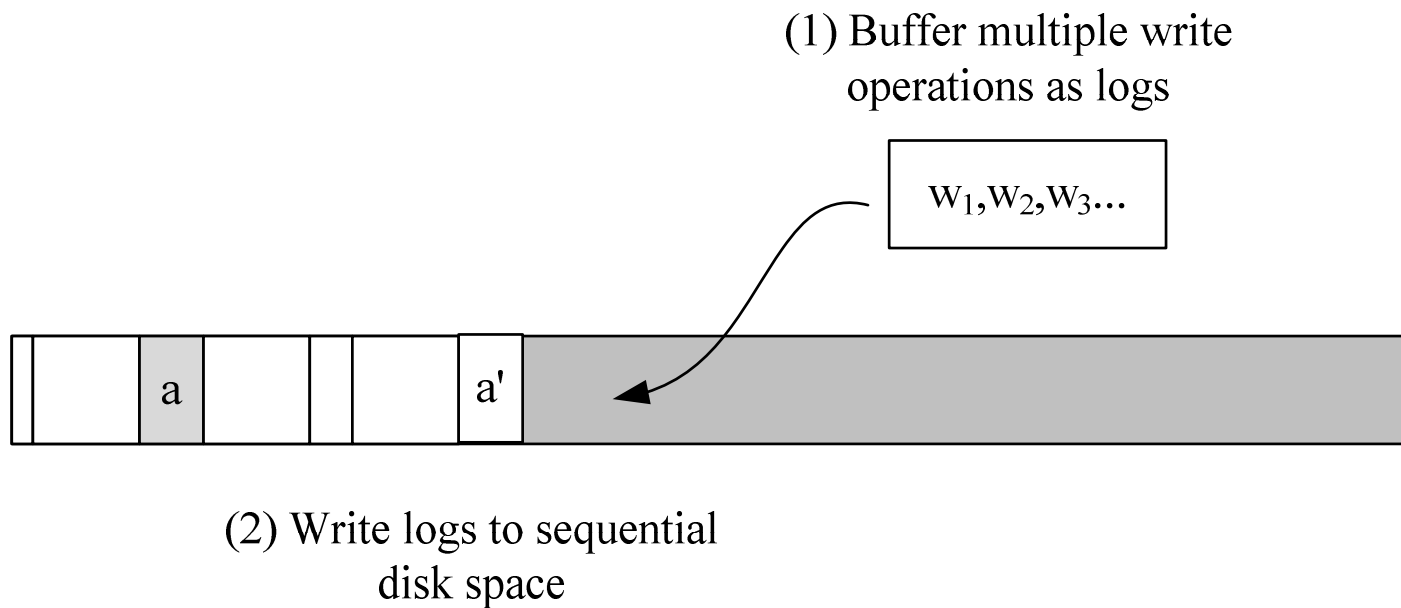
Journaling in Ext3 and Ext4

- Three journaling modes in ext3/4
 - Journal mode
 - Write both user data and metadata to the journal
 - Commit user data/metadata, doubling write traffic
 - Ordered mode
 - Write only metadata to the journal
 - Commit metadata only after user data (of the same transactions) have been written 只有把user data完整寫入disk後才把metadata從journal裡移過去
 - Write-back mode
 - Write only metadata to the journal
 - The order of metadata commit and user data write is arbitrary

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 one single logging area
 - No need to “copy back” (but would require compaction)
- The main idea is to optimize random write
 - Convert random writes into sequential writes
 - Out-of-place updates 並不一定會在資料原來的位置更新
- Examples
 - NILFS2
 - F2FS for Android devices

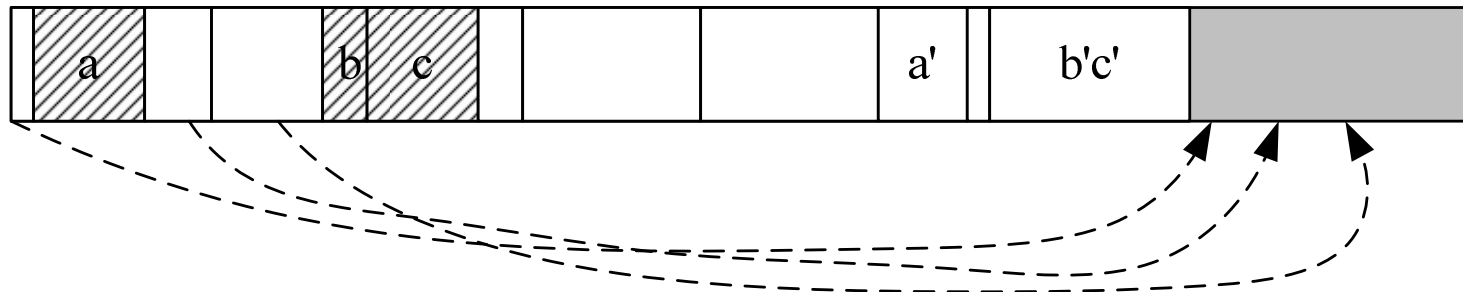
The Concept of Log-Structured File Systems



Updates are out of place

Compaction (Garbage Collection) in LFS

(3) Out-of-place updates produce invalid data



(4) Reclaim contiguous disk space with compaction (garbage collection)



(5) compaction produces contiguous free space

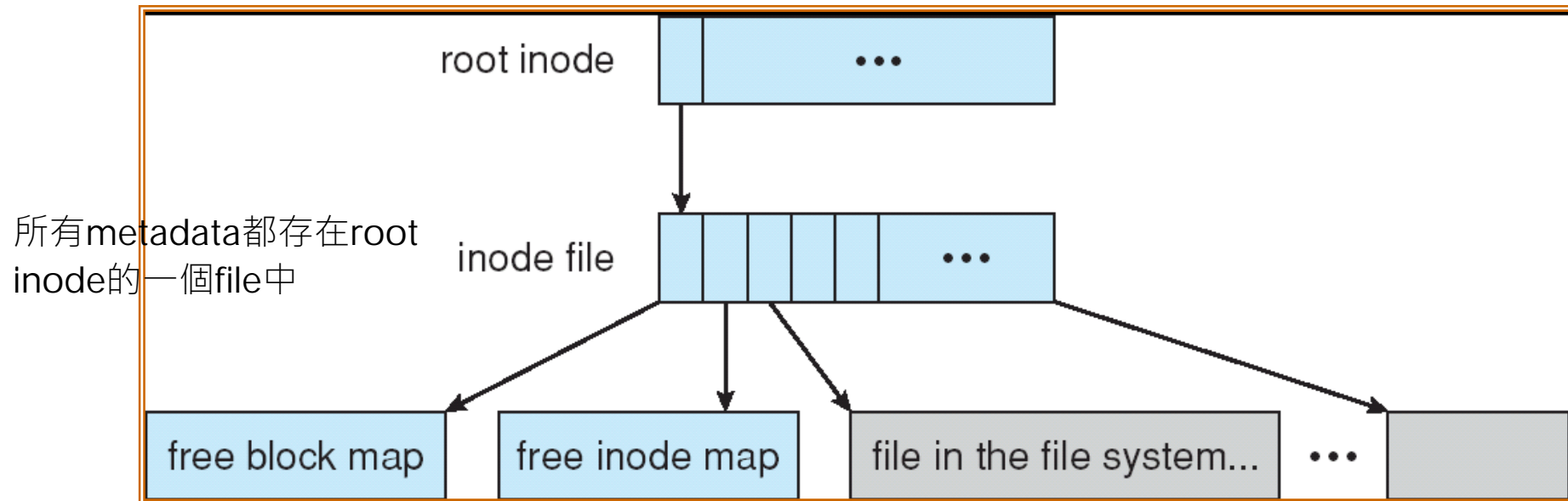
Log-Structured File Systems -- Summary

- 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
- Problems:
 - Need compaction (garbage collection) to produce sequential space for new writes 這樣才能有足夠的連續空間

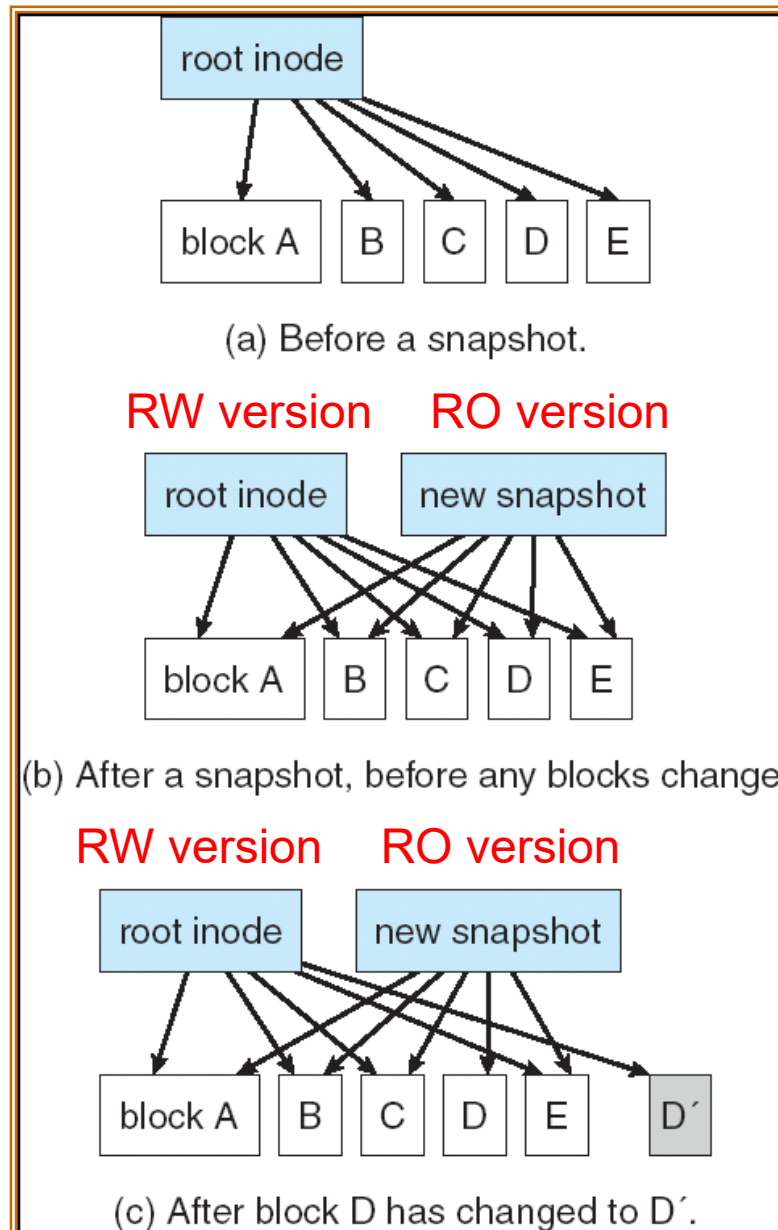
WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
 - NVRAM for write caching 非揮發性的隨機存取記憶體 (Non-Volatile Random-Access Memory)
- Similar to Berkeley Fast File System, with extensive modifications
- We shall focus on **snapshots** in WAFL

The WAFL File Layout



Snapshots in WAFL



Snapshotting is useful in

- On-the-fly backup
- Versioning

真正多出來的只有snapshot需要的root inode
與修改過的blocks

End of Chapter 11