作業系統：File System

目錄

作業系統：File System 1
1. File Concept 2
1.1 開啟與關閉檔案：open / close 2
1.2 檔案鎖定(File Locking) 3
1.3 存取方法（Access Methods） 3
1.4 順序存取（Sequential Access） 3
1.5 直接存取（Direct Access / Random Access） 3
1.6 索引存取方式（Index Access and Beyond） 4
2. 檔案系統保護機制（Protection） 4
2.1 存取控制（Access Control） 4
3. Memory-Mapped Files（記憶體映射檔案） 5
4. 多進程共享記憶體 6
4.1 多個程式如何共享檔案？ 6
4.2 搭配互斥機制（mutual exclusion）使用 6
4.3 共享記憶體通信範例 6
5. 檔案系統結構(File-System Structure) 6
6. 檔案系統的分層架構 7
6.1 I/O Control（最底層） 7
6.2 Basic File System（基本檔案系統） 7
6.3 File-Organization Module（檔案組織模組） 7
6.4 Logical File System（邏輯檔案系統） 7
7. File-System Operations（檔案系統操作） 8
7.1 建立與操作檔案流程 8
8. 目錄實作(Directory Implementation) 9
8.1 Linear List 9
8.2 Hash Table 10
8.3 碰撞處理（Collision Handling） 10
9. File Allocation（檔案配置） 10
9.1 Contiguous Allocation（連續配置） 10
9.2 Linked Allocation（鏈結配置） 11
9.3 Indexed Allocation（索引配置） 12
10. Free-Space Management（空間管理） 13
10.1 方式：Bit Vector（位元向量） 13
10.2 方式：Linked List（鏈結串列） 13
10.3 方式：Grouping（分組） 14

10.4 方式：Counting（計數法）.....14
11. 檔案系統內部結構(File-System Internals).....14
11.1 儲存裝置與檔案系統的關係.....14
11.2 File-System Mounting（檔案系統掛載）.....15
11.3 檔案共享(File Sharing).....15
12. Virtual File Systems（虛擬檔案系統）.....16
13. Remote File Systems（遠端檔案系統）.....17
13.1 Client–Server 模型.....17
13.2 認證與安全性問題.....17
14. Consistency Semantics（一致性語意）.....17
14.1 UNIX Semantics.....17
14.2 Session Semantics（如 AFS）.....17
14.3 Immutable-Shared-Files Semantics.....18

1. File Concept

File（檔案）定義為使用者與程式儲存資訊的邏輯單位，是記錄在次儲存裝置上的一組相關資訊的集合。作業系統將實體的儲存裝置抽象化為統一的「邏輯檔案單位」，以便管理與存取。

抽象化的目的：

- 讓使用者不用關心實際的裝置細節，例如磁軌、磁區、快取等。
- 抽象成一個「檔案」，提供一致的使用介面（例如 open、read、write）。

Table 1：File Operations（檔案操作）

操作	說明
Create	建立檔案並分配空間與目錄項目
Open	打開檔案，建立檔案控制項（file handle）
Write	將資料寫入開啟的檔案，更新寫入指標
Read	從檔案讀資料，更新讀取指標
Reposition	改變目前檔案指標位置（seek）
Delete	移除檔案內容與目錄項目
Truncate	清除檔案內容，但保留檔案屬性

1.1 開啟與關閉檔案：open / close

開啟後，系統將檔案資訊記錄在 open-file table（開啟檔案表）中，而關閉檔案會從 open-file table 中移除其項目。也因此，在資料結構上：

- 每個進程有一個 per-process open-file table
- 系統層級有一個 system-wide open-file table

1.2 檔案鎖定(File Locking)

目的：當多個進程存取同一檔案時，防止資料衝突或損壞。

其鎖定類型：

- Shared：多個進程可讀
- Exclusive：只有一個進程可讀寫

其鎖定方式：

- Mandatory（強制鎖定）：系統強制執行鎖定，其他程式不能存取。
- Advisory（建議性鎖定）：其他程式需「自願遵守」鎖定機制（如 UNIX）。

1.3 存取方法（Access Methods）

當資料儲存在檔案中後，必須能夠被有效地讀取到記憶體中。作業系統定義了多種檔案存取方式，以下是三種主要類型：

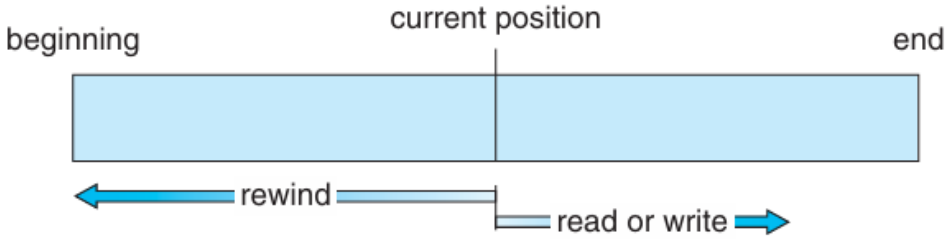
1.4 順序存取（Sequential Access）

最簡單也最常見的存取方式。OS 會按照檔案內容的順序來讀寫，一次處理一筆資料（record 或區塊）。

不能直接跳到特定位置，只能依序往前處理。

```
read_next()    // 讀取下一筆資料，自動更新檔案指標
write_next()   // 寫入資料至檔案尾端，自動移動指標
reset()        // 重設指標到開頭
```

Figure 1：Sequential-access file



1.5 直接存取（Direct Access / Random Access）

資料被視為「編號的區塊（block）」或「固定長度的紀錄（record）」。可以直接跳至特定區塊進行讀寫，順序不重要。

常見於硬碟系統（磁碟能隨機存取）。

```
read(14)       // 讀取第 14 個區塊
write(53)      // 寫入第 53 個區塊
position_file(n)   // 將指標移動到第 n 區塊，再 read_next()
```

Figure 2：Simulation of sequential access on a direct-access file.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

1.6 索引存取方式（Index Access and Beyond）

在直接存取法的基礎上，加入索引（Index）以加速查找。

假設有一個商品售價表檔案，每筆紀錄包含：

- 10 位數的 UPC（商品條碼）
- 6 位數的價格
- 每筆資料大小：16 bytes
- 每個區塊大小：1024 bytes（可存 64 筆資料）
- 若有 120,000 筆資料 ⇒ 需約 2,000 個區塊

解法：

- 建立「主索引表」：每個索引項目記錄一個區塊的第一個 UPC。
- 主索引表大小：10 × 2000 = 20,000 bytes，可載入記憶體。
- 查詢過程：使用二分搜尋法在主索引中查找，再存取目標區塊。

2. 檔案系統保護機制（Protection）

保護檔案的兩大面向：

- Reliability（可靠性）：防止資料因硬體損壞、錯誤操作等而遺失。
- Protection（保護性）：防止未授權的存取或惡意使用資料。

2.1 存取控制（Access Control）

方法：Access Control List（ACL），每個檔案或目錄都有一張 Access Control List，每筆記錄指定：

- 使用者或群組
- 可執行的操作類型（read/write/execute...）

然而，問題是：

- 使用者多時，ACL 太長不好管理
- 對目錄資料結構的空間配置造成挑戰（需變長）

作業系統：File System

簡化方法：UNIX-style 三類使用者模型：

- Owner：檔案建立者
- Group：同群組使用者
- Others：其他使用者（陌生人）

其中，權限三位元組：

- R：讀取
- W：寫入
- X：執行

UNIX 每個檔案會有 9 個權限位：

-rw-rw-r-- book.tex

- 第一格：regular file
- 第二格：owner：rw
- 第三格：others：r

3. Memory-Mapped Files（記憶體映射檔案）

記憶體映射檔案是一種將檔案內容對映（mapping）到虛擬記憶體空間的技術，使應用程式可像存取記憶體一樣直接存取檔案。

Table 2：對比傳統 I/O 模式

傳統檔案存取	**記憶體映射模式**
使用 read()、write()	使用 mmap() 將檔案對映到記憶體
每次 I/O 都會觸發系統呼叫與磁碟存取	存取檔案如同操作記憶體（低延遲）

運作流程：

1. 程式呼叫 mmap() → 建立記憶體與磁碟檔案的對應關係
2. 初次存取時會觸發 Page Fault
3. 系統自動從磁碟載入頁面到實體記憶體
4. 後續存取就像在使用一般記憶體（array、指標操作）

寫入注意：

- 非同步寫入：修改不會立即同步到磁碟
- 通常在關閉檔案或釋放頁面時才寫回

優點	缺點／風險
高效：省略 system call overhead	寫入非同步：資料可能未即時寫回磁碟
程式碼簡潔：直接當作 array 操作	資料一致性需同步控制（多進程共享時）
支援共享記憶體（實現 IPC）	若不正確同步，可能發生 race condition
支援部分區段對映、分頁載入	不適合過大檔案（記憶體限制）

4. 多進程共享記憶體

4.1 多個程式如何共享檔案？

- 多個 process 可以對映（mmap）相同檔案的相同部分
- 共享的是同一個實體記憶體頁面
- 若搭配 Copy-On-Write (COW)，可：
  - 開始時共享
  - 修改時複製（只影響自己）

4.2 搭配互斥機制（mutual exclusion）使用

例如

- 信號量（semaphore）
- mutex
- condition variable

4.3 共享記憶體通信範例

- IPC（Inter-Process Communication）
- Producer/Consumer 模式
- 多進程共享 cache/buffer

5. 檔案系統結構(File-System Structure)

主要使用的儲存設備：

- 傳統磁碟（HDD）
- 非揮發性記憶體裝置（NVM）：如 SSD、Flash

其儲存設備特性：

- 可重寫（Rewrite in place）：可隨時覆寫磁碟上任意區塊
- 隨機存取（Random Access）：可直接定位任一區塊，支援高效率存取

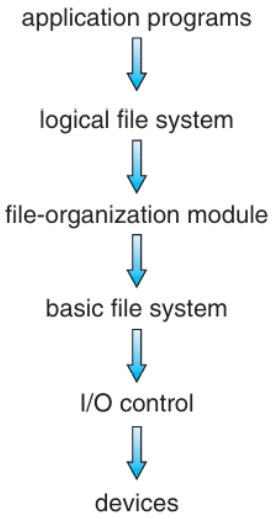
其中，I/O 操作是以 Block 為單位進。

檔案系統的設計挑戰，可以涉及兩個主要問題：

使用者介面層級（邏輯設計）	● 檔案名稱、屬性、權限 ● 檔案操作（如 open、read、write） ● 目錄結構與存取方式
實作層級（物理儲存對映）	● 如何將檔案映射到磁碟區塊 ● 區塊分配、空間管理、快取策略等

6. 檔案系統的分層架構

Figure 3：檔案系統的分層架構



6.1 I/O Control（最底層）

包含裝置驅動程式（Device Driver）與中斷處理程式（Interrupt Handler）。其功能：

- 將高階命令（如讀取區塊）轉換為低階硬體操作
- 控制器與 I/O 裝置的橋樑
- 撰寫特殊位元到控制器暫存區（memory-mapped I/O）

6.2 Basic File System（基本檔案系統）

負責發出「區塊 I/O 命令」，例如 read block 17、write block 30。涉及：

- I/O 排程（I/O Scheduling）
- 快取與緩衝區（buffer cache）管理

6.3 File-Organization Module（檔案組織模組）

檔案與區塊之間的邏輯對應（ex: 檔案第 3 個邏輯區塊 → 磁碟第 47 區塊）

空間管理：追蹤哪些區塊是空的、可配置，包括

- 檔案區塊號編號：Block 0, 1, ..., N
- 使用空間配置策略（如 Free List、Bitmap）

6.4 Logical File System（邏輯檔案系統）

管理所有的檔案中繼資料（Metadata），Metadata 包含：

- 檔案名稱、屬性、存取權限
- 檔案位置資訊（透過 FCB: File Control Block）

7. File-System Operations（檔案系統操作）

作業系統提供一系列操作來存取檔案內容，如 open()、close()、read()、write() 等。這些操作背後涉及多種儲存結構（on-storage structures）與記憶體內部結構（in-memory structures）。

Table 3：儲存結構（On-storage Structures）

結構名稱	說明	在 UNIX 的名稱	在 NTFS 的名稱
Boot Control Block	開機資訊區，含是否可從此磁區開機	boot block	partition boot sector
Volume Control Block	整體磁區資訊，如總區塊數、區塊大小、剩餘區塊	superblock	master file table（MFT）
Directory Structure	目錄結構，用於管理檔案位置、名稱與關聯	inode 對應表	master file table（MFT）
File Control Block	FCB，紀錄單一檔案的屬性、權限、位置等資訊	inode	MFT 中的一筆資料列

Table 4：記憶體結構（In-memory Structures）（這些結構會在系統掛載（mount）時載入，卸載（dismount）時清除）

結構名稱	說明
Mount Table	每個掛載磁區的資訊
Directory Cache	快取最近使用過的目錄資訊
System-Wide Open File Table	所有開啟中的檔案資訊（每個開啟檔案會有一份）
Per-Process Open File Table	每個行程自身開啟的檔案，會指向系統表
Buffers / Caches	暫存區塊資料，加速 I/O 操作

7.1 建立與操作檔案流程

1. 建立檔案（Create）

   呼叫邏輯檔案系統 → 配置 FCB → 更新目錄資料結構 → 寫入磁碟。

2. 開啟檔案（Open）

   執行 open("file.txt")：搜尋系統級 open-file table，看是否已開啟

   - 是：該程序加入指標（per-process open-file table
   - 否：搜尋目錄 → 載入 FCB → 新增 entry 到系統級與程序級 table

   回傳一個「指標」作為 file descriptor（UNIX）或 file handle（Windows）。

3. 讀寫檔案（Read / Write）

   read(fd, buf, n)：

   → 根據檔案指標，查找目前位移

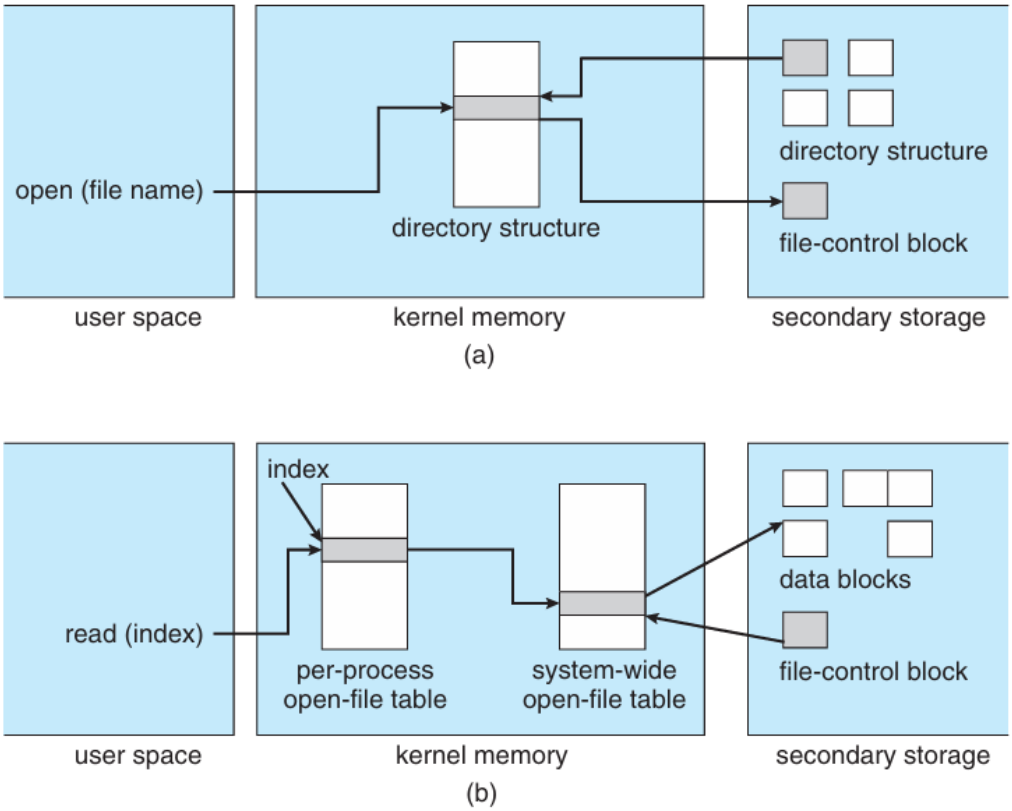
   → 從對應區塊讀取資料，回傳 buffer

   - FCB 中的 data blocks pointer 決定實際資料在哪裡
   - 資料會暫存於 buffer，降低磁碟 I/O 負擔

4. 關閉檔案（Close）

- 從程序級表格移除 entry。
- 系統級表格的 open count 減 1，若為 0，更新 metadata 回磁碟。

Figure 4： In-memory file-system structures. (a) File open. (b) File read



8. 目錄實作(Directory Implementation)

目錄是作業系統中管理檔案名稱與檔案資料區塊位置的結構。良好的目錄設計會影響以下幾點：

- 檔案搜尋效率（如開啟檔案時）
- 檔案新增與刪除的處理成本
- 使用者體驗（例如列出目錄內容時是否快速）

8.1 Linear List

最簡單的目錄資料結構是一個線性串列，每筆資料包含：

- 檔案名稱
- 指向該檔案資料區塊的指標（如 inode 或 block pointer）

Table 5：操作說明

操作	說明
Create	搜尋有無重複名稱 → 加入到串列尾端
Delete	搜尋目標檔案 → 釋放資源 → 處理空位（方法如下）

Search	線性搜尋（時間複雜度：O(n)）

刪除後如何處理空位

- 標記未使用（如使用空白名稱、無效 inode、used/unused 位元）
- 建立空位清單
- 將最後一筆資料搬移至刪除的位置（避免資料碎裂）

缺點

- 搜尋效率低（對大型目錄非常不利）
- 若需保持排序，新增與刪除變得困難（需大量搬移資料）

8.2 Hash Table

使用線性串列 + 雜湊表的結構，其中：

- 雜湊表根據檔名產生 index
- index 對應至線性串列中的目錄項目

Table 6：操作說明

操作	**時間複雜度**
Search	O(1)（理想狀況）
Insert	O(1)（附加於鏈結串列）
Delete	O(1) 或 O(n)（取決於碰撞處理方式）

8.3 碰撞處理（Collision Handling）

線性探查法（linear probing）	將新項目放在下一個空位（容易形成「聚集效應」）
鏈結法（chaining）	每個 bucket 是一個 linked list，遇到碰撞就附加在其後

9. File Allocation（檔案配置）

檔案配置（File Allocation）的目的是將檔案的資料區塊配置到儲存裝置（如磁碟）上。因為檔案可以動態成長，也可能被刪除或擴展，因此有效的配置策略會直接影響效能與空間利用率。

9.1 Contiguous Allocation（連續配置）

為檔案分配一連串連續的磁碟區塊

資料結構：檔案控制區塊（FCB）紀錄起始區塊與長度。

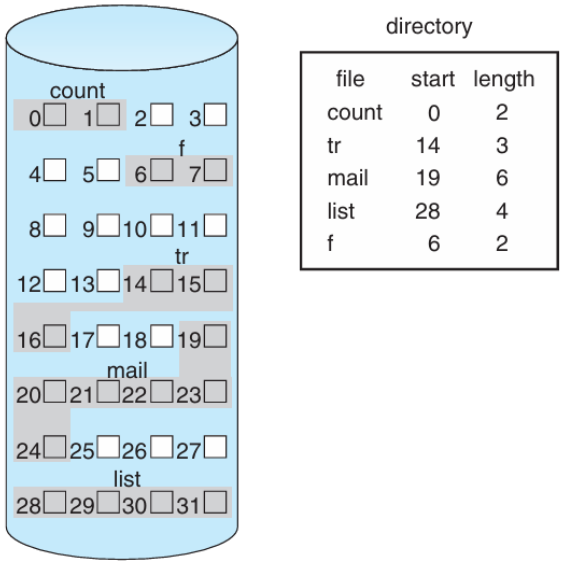
問題：

- 外部碎裂（external fragmentation）
- 若使用者低估檔案長度，需搬移檔案重新配置

File A: 起始區塊 = 10，長度 = 4

則該檔案佔據區塊 10, 11, 12, 13

Figure 5：Contiguous allocation of disk space

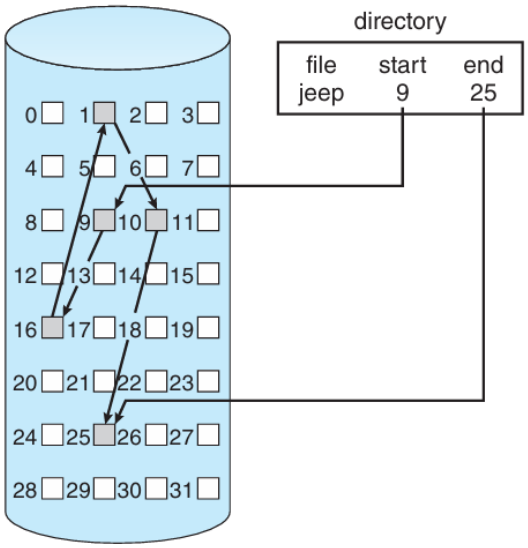


9.2 Linked Allocation（鏈結配置）

每個檔案由一串磁碟區塊組成，每個區塊都包含指向下一個區塊的指標（類似 Linked List）。FCB 只需要儲存 第一個區塊號碼

- 每個區塊的最後一部分是「下一個區塊」的位址
- 優點：
  - 不需預估檔案長度
- 不會產生外部碎裂
- 缺點：
  - 只支援 sequential access
  - 對於每次存取都需要 追蹤整條鏈，較慢
  - 若任一指標損壞，整個檔案會遺失

Figure 6：Linked allocation of disk space

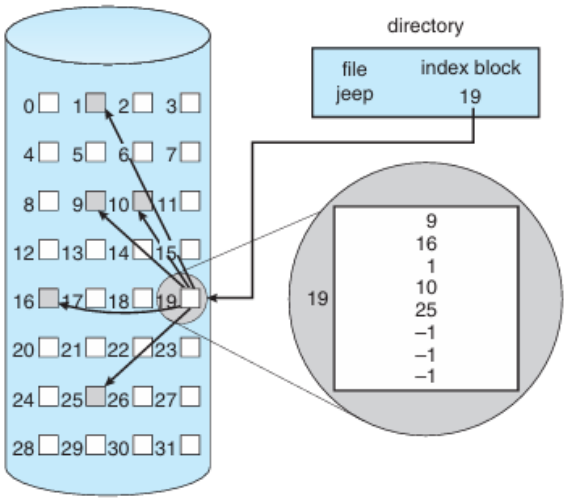


9.3 Indexed Allocation（索引配置）

使用一個獨立的「索引區塊」來儲存檔案所有資料區塊的位址。

- 類似陣列的方式管理檔案的所有區塊
- FCB 包含指向此「索引區塊」的指標
- 優點：
  - 支援 random access
  - 不會產生碎裂
  - 易於新增資料區塊
- 缺點：
  - 小檔案的索引區塊開銷偏高
  - 索引區塊大小限制了檔案最大容量（除非使用多層索引）

Figure 7： Indexed allocation of disk space.



10. Free-Space Management（空間管理）

儲存空間有限，因此當檔案被刪除或資料更新後，系統必須能夠重複利用這些空間。

此機制透過「free-space list（空間管理表）」來記錄哪些區塊（blocks）是「尚未被使用的」，以便供後續檔案配置使用。

10.1 方式：Bit Vector（位元向量）

使用一串 位元（bit） 來記錄每個磁碟區塊的使用狀態：

- 1 → 區塊是空的（free）
- 0 → 區塊是已使用（allocated）

假設區塊 2–5、8–13、17–18、25–27 是空的，其餘是已使用，則 bitmap 如下：

```
Index:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
Value:  0 0 1 1 1 1 0 0 1 1  1  1  1  1  0  0  0  1  1  0 ...
```

優點

- 簡單直觀
- 使用硬體提供的 位元操作指令（如 bsf, btr）可快速找出第一個可用區塊
- 易於尋找連續的空區塊（e.g. 尋找 n 個連續 1）

缺點

- 若整個位圖太大，放不進主記憶體，搜尋效率會下降
- 寫入與同步到磁碟需額外時間

10.2 方式：Linked List（鏈結串列）

將所有 free blocks 以鏈結方式串起來，每個區塊存有下一個 free block 的位址

```
Free List Head → Block 2 → Block 3 → Block 4 → ... → Block 27 → NULL
```

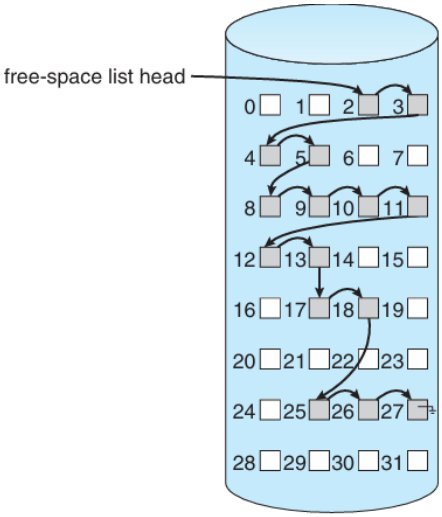
優點

- 不需要大型 bitmap，僅需追蹤第一個 free block
- 資料寫入簡單，只需修改指標

缺點

- 若要搜尋第 n 個可用區塊，需反覆讀取磁碟（I/O 成本高）
- 不適合隨機或大規模查找

Figure 8：Linked free-space list on disk.



10.3 方式：Grouping（分組）

類似 linked list，但每個 free block 裡面不只存一個位址，而是存 n 個 free block 的位址，其中最後一個位址指向下一組區塊。

優點

- 一次可以取得大量 free block 的位址
- 減少 I/O 次數，適合批次分配

缺點

- 還是需要讀磁碟才能取得區塊列表
- 若 n 選得不好，仍可能效能受限

10.4 方式：Counting（計數法）

針對「連續」的 free blocks，以 (開始位址, 連續個數) 方式記錄。

(100, 4) → 表示區塊 100～103 是空的

(110, 2) → 表示區塊 110～111 是空的

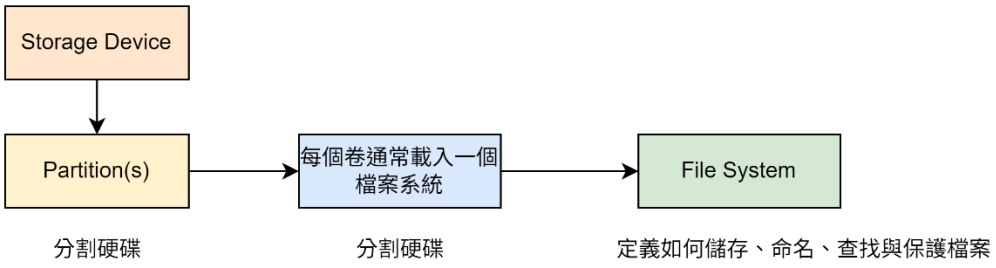
11. 檔案系統內部結構(File-System Internals)

電腦系統的主要功能之一是儲存與管理大量檔案。檔案不是儲存在記憶體中，而是儲存在非揮發性儲存裝置（如 HDD、SSD、光碟、NVM）中。現代系統中往往有數百萬個檔案，因此需要一個結構化方式來管理這些檔案：檔案系統。

11.1 儲存裝置與檔案系統的關係

儲存裝置 → 分割區（partitions）→ 卷（volumes）→ 檔案系統（file systems）

Figure 9：儲存裝置與檔案系統的關係



檔案通常會組織成群組，並透過目錄（directory）來管理。每個目錄是一個可包含檔案與子目錄的特殊檔案。這樣的層級架構方便搜尋、分類與權限控制。

11.2 File-System Mounting（檔案系統掛載）

將一個檔案系統的內容整合進現有的目錄結構中，稱為掛載（mounting）。只有掛載過的檔案系統，系統與使用者才能透過檔名來存取其內容。

Table 7：掛載程序（Mount Procedure）

步驟	**說明**
提供裝置名稱與掛載點	使用者或系統提供如 /dev/sda2 和目錄 /home
（可選）指定檔案系統類型	某些系統自動偵測；某些則需明確指定（如 ext4, FAT32）
驗證檔案系統格式	作業系統會讀取該儲存裝置的檔案系統結構是否正確
建立掛載記錄	作業系統在目錄結構中標記該掛載點對應新的檔案系統

11.3 檔案共享(File Sharing)

目的是讓多位使用者可協作處理任務、共享資源，提升效率。然而，共享檔案會帶來挑戰：

- 權限控制（誰可以讀寫？）
- 檔案同步（多人同時修改怎麼辦？）
- 跨系統共享（本地 vs 遠端）

Table 8：多使用者的問題（Multiple Users）

問題	說明
存取控制與保護（Access Control & Protection）	檔案不能預設開放給所有人，需設定誰可看、誰可寫
檔案屬性管理增加	相較單一使用者系統，需額外管理「檔案擁有者」與「群組」資訊
檔案識別與比對機制	系統需根據使用者 ID (UID) 與群組 ID (GID) 決定權限

Table 9：UNIX 系統中常見的三層權限模型（UID 與 GID 是寫在檔案的 Metadata 中，作業系統會比對使用者的身分來決定能否存取）

類別	**權限**	**控制方式**
擁有者（Owner）	可修改檔案屬性與權限	使用者本人

群組（Group）	擁有部分讀寫權限	同群組成員
其他人（Others）	最少權限	非上述對象

12. Virtual File Systems（虛擬檔案系統）

現代系統需支援多種檔案系統類型（如 ext4、NTFS、NFS），還可能包含網路檔案系統，傳統作法為每種檔案系統寫一份讀寫函式，不利維護與擴充。

VFS 是一種中介層（Abstraction Layer），解耦「檔案操作」與「實際檔案系統實作」。

Figure 10：Schematic view of a virtual file system

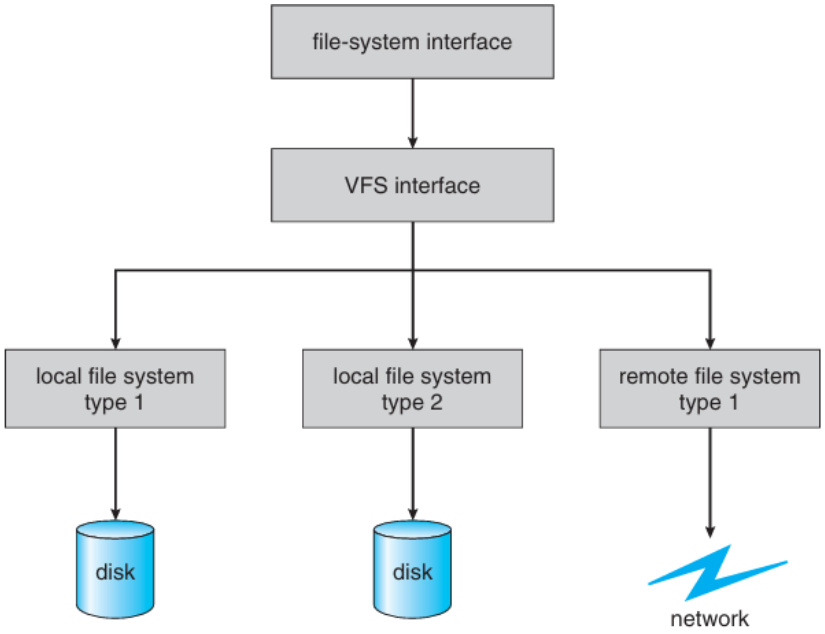


Table 10：VFS 核心功能

功能	說明
分離泛用操作與檔案系統實作	同樣的 read() 可用於本地或網路檔案
對檔案建立 vnode（虛擬節點）	vnode 提供唯一識別碼，跨主機識別檔案
支援跨檔案系統存取	像是在本地存取一樣操作 NFS、tmpfs 等

Linux VFS 中的四大物件如下表所示，每個物件內含 函式表（function pointer table），指向該檔案系統的實作函式。VFS 呼叫對應函式，例如 file->read()，不需關心底層檔案類型。

Table 11：Linux VFS 中的四大物件

物件	**說明**
inode	單一檔案的資訊（檔名、權限、大小）
file	開啟檔案後的操作控制結構（如檔案描述元）

superblock	整個檔案系統的統一描述（如掛載點、磁區資訊）
dentry	目錄項目，描述檔案名稱與 inode 的對應關係

13. Remote File Systems（遠端檔案系統）

早期的 Remote File Systems 是透過 ftp 傳送檔案，其特性是需要手動上傳下載。而中期開始使用 DFS（Distributed File System），他是把遠端資料夾掛載進本地。而現代主要是採用 Web / Cloud Storage 的方式，透過 HTTP 瀏覽與下載。

13.1 Client–Server 模型

- 伺服器（Server）：擁有實體檔案，宣告哪些資料夾可分享
- 客戶端（Client）：將遠端檔案系統掛載進自己系統中使用

存取流程：

1. Client 發出 open 請求，附上使用者 ID
2. Server 檢查權限並授權
3. 若允許，回傳檔案句柄（handle）
4. Client 可進行 read/write 等操作
5. 完成後 close

13.2 認證與安全性問題

- ID 對不上的問題（client UID ≠ server UID）會導致錯誤授權
- 假冒 IP 或 UID 會造成未授權存取
- 常見認證方式：
  - 不安全：IP 判別、帳號密碼明碼傳送
  - 安全：加密金鑰、Kerberos、LDAP 認證

14. Consistency Semantics（一致性語意）

當多個使用者共享同一檔案時，一致性語意（Consistency Semantics）定義了這些使用者何時能看到彼此的更動，與更動是如何同步的。這個概念類似於第 6 章的同步（synchronization）問題，但考慮到磁碟與網路傳輸的延遲，檔案系統通常不實作完全同步，而是依照不同需求設計一致性模型。常見的幾種語意模型如下：

14.1 UNIX Semantics

- 只要一位使用者寫入檔案，其他使用者立即可見該變動
- 多使用者共享一個 file pointer（檔案位置指標）
- 適合小型協作，但容易造成競爭與延遲

14.2 Session Semantics（如 AFS）

- 使用者在 open() ~ close() 間操作

- 檔案變更在 close() 前不會對他人可見
- 他人需重新 open() 才會看到新內容
- 適合分散式系統，減少同步負擔

14.3 Immutable-Shared-Files Semantics

- 檔案一旦共享即「不可變更」
- 內容固定、名稱唯一不可重複
- 適合公開讀取資料，如教學材料或公版數據