作業系統：File System

**目錄**

[作業系統：File System 1](#_Toc204023883)

[1. File Concept 1](#_Toc204023884)

[1.1 開啟與關閉檔案：open / close 2](#_Toc204023885)

[1.2 檔案鎖定(File Locking) 2](#_Toc204023886)

[1.3 存取方法（Access Methods） 2](#_Toc204023887)

[1.4 順序存取（Sequential Access） 2](#_Toc204023888)

[1.5 直接存取（Direct Access / Random Access） 2](#_Toc204023889)

[1.6 索引存取方式（Index Access and Beyond） 3](#_Toc204023890)

[2. 檔案系統保護機制（Protection） 3](#_Toc204023891)

[2.1 存取控制（Access Control） 4](#_Toc204023892)

[3. Memory-Mapped Files（記憶體映射檔案） 4](#_Toc204023893)

[4. 多進程共享記憶體 5](#_Toc204023894)

[4.1 多個程式如何共享檔案？ 5](#_Toc204023895)

[4.2 搭配互斥機制（mutual exclusion）使用 5](#_Toc204023896)

[4.3 共享記憶體通信範例 5](#_Toc204023897)

# 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 |

## 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

## 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）。

## 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：

## 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）。

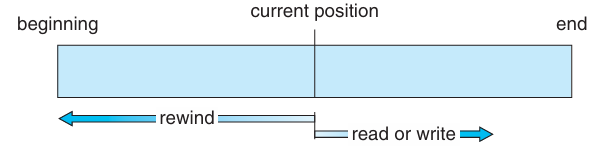
不能直接跳到特定位置，只能依序往前處理。

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



## 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。

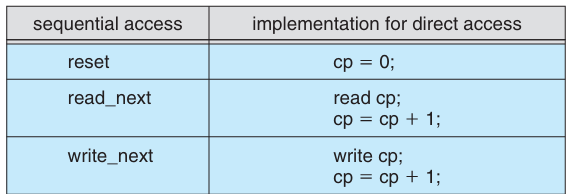
常見於硬碟系統（磁碟能隨機存取）。

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.



## 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。

# 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。

## 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）

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

# 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） |

# 多進程共享記憶體

## 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）

## 搭配互斥機制（mutual exclusion）使用

例如

* 信號量（semaphore）
* mutex
* condition variable

## 共享記憶體通信範例

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

# 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

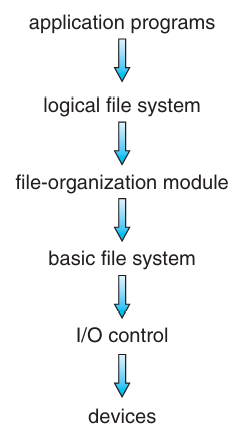
其中，I/O 操作是以 Block 為單位進。

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 |

# 檔案系統的分層架構

Figure 3：檔案系統的分層架構



## 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）

## 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

## 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）

## 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）

# 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 |

## 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。

1. 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）。

1. 讀寫檔案（Read / Write）

read(fd, buf, n)：

→ 根據檔案指標，查找目前位移

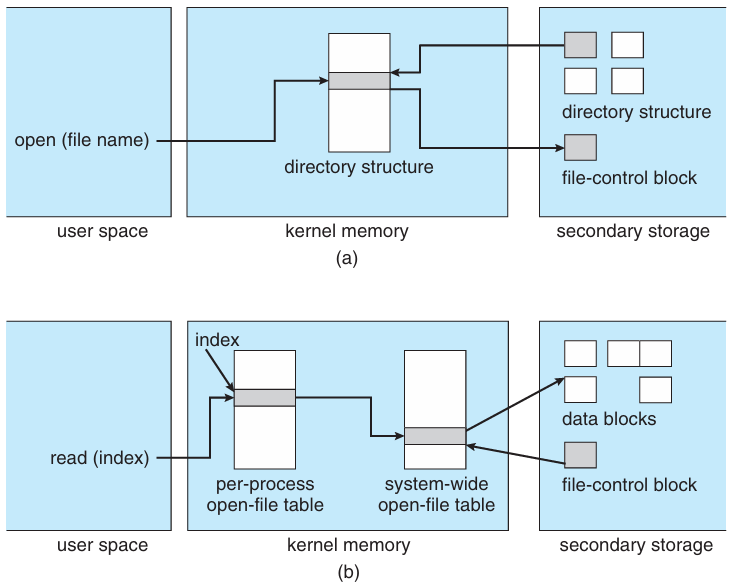
→ 從對應區塊讀取資料，回傳 buffer

* 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

1. 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



# 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）

## 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）

## 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） |

## 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 |

# 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。

## 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。

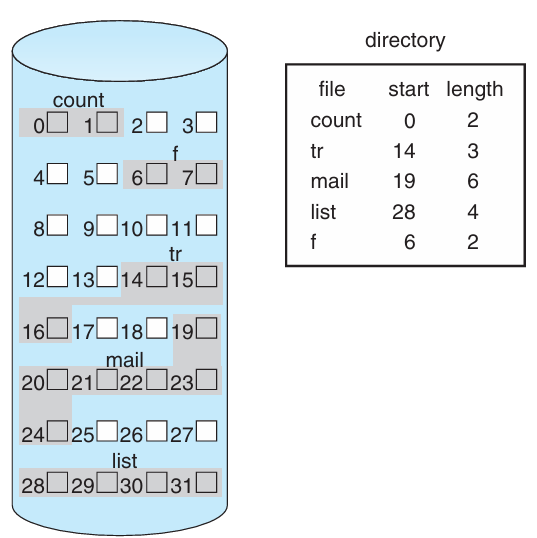
問題：

* 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

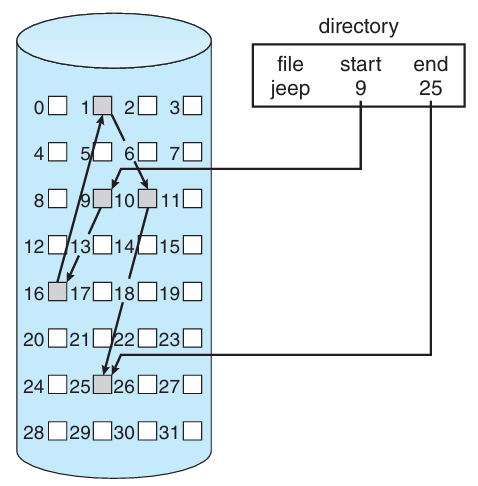


## Linked Allocation（鏈結配置）

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

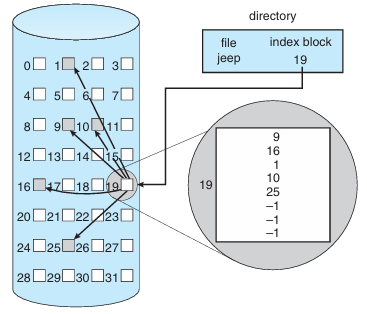


## Indexed Allocation（索引配置）

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.



# 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。

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

Figure 8：Linked free-space list on disk.

