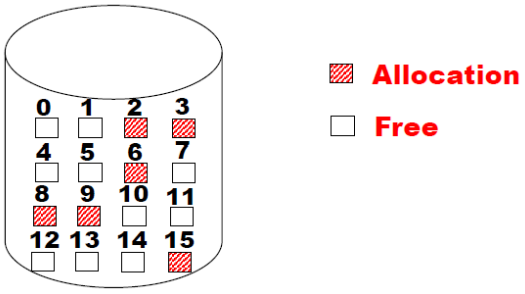
磁碟管理

Free Space Management

- 基本概念: Disk Allocation/Free Space 單位是 Block 為主

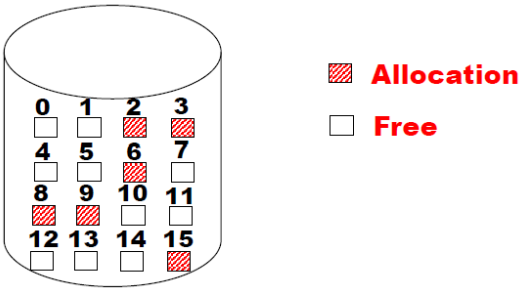


- 方法
  - Bit Vector(Bit Map)
  - Link List
  - Combination

- Counting

Bit Vector

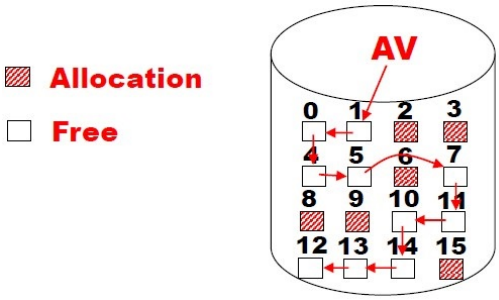
- 定義：用一組 Bit 表示 Block 是否被配置，一個 Bit 對應一個 Block
  - Bit = 0 表示 Free
  - Bit = 1 表示 Allocate
  - 下圖 Bit Vector = 0011001011000001



- Bit Vector 必須先載入記憶體，OS 才知道那些 Block 可以使用
- 優點
  - 簡單
  - 容易找到連續可用空間 (找到連續"0")
- 缺點
  - Bit Vector 佔用記憶體間，不適用大型 Disk
  - ex. Block > 107 --> Bit Vector 會非常大，**記憶體可能容納不下**

Link List

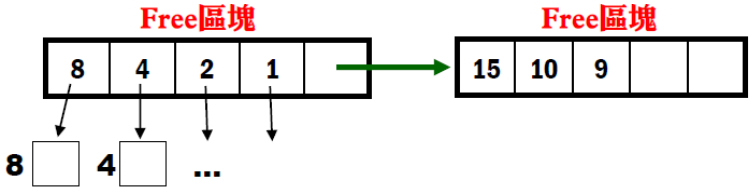
- 利用 Link List 串接 Free Block，形成 AV List



- 優點：加入/刪除 Block 容易
- 缺點：效率不佳，搜尋可用區塊時需從頭檢視串列，I/O Time 過長 (檢視一個 Block 的時間複雜度 O(N))

Combination

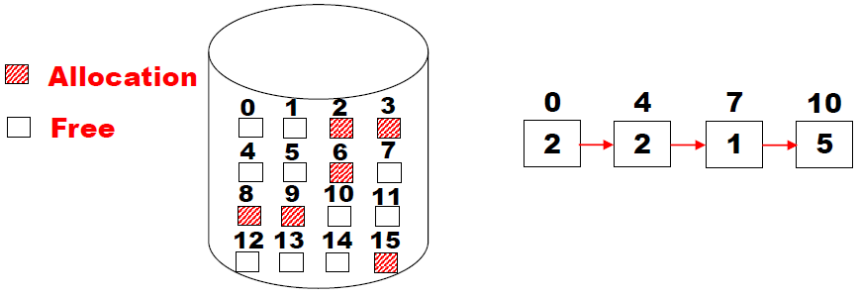
- 用某個可用區塊記錄其他可用區塊的編號，如果此區塊空間不足，再用 Link List 方式串接
  - AV List



- 優點:
  - 加入/刪除 Block 容易
  - 可迅速取達大量可用區塊
- 缺點
  - 效率不佳， I/O Time 過長

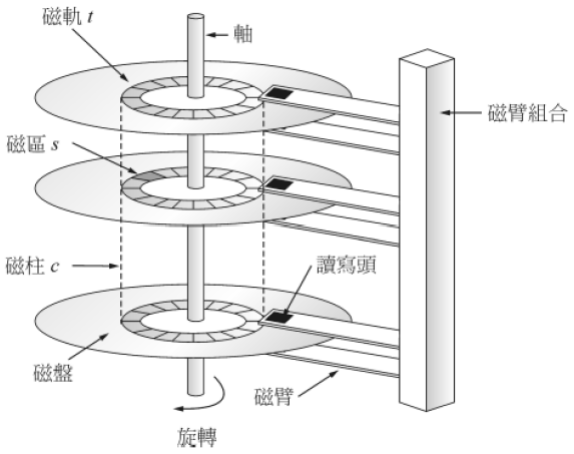
Counting

- 適用於連續區塊較多的情況
  - 在一個可用區塊中，記錄其後的連續可用區塊數目(含本身)
  - AV List



- 如果連續區塊數目夠多，Link List 的長度可大幅縮短

Disk 結構



- Track(磁軌)
- Sector(磁區)
- Cylindar(磁柱)
  - 不同面的相同 Tracks 行程的集合
- Read/Write Head(讀寫頭)
- Disk Capacity
  - 若硬碟有 10 Disks，正反兩面都可以儲存資料，但最上一面和最下一面不存資料，每一個表有 1024 Tracks，每一個 Track 有 256 Sector，一個可以儲存 512 bytes，求硬碟容量?
  - Ans:
    - (10 * 2 - 2) * 1024 * 256 * 512 bytes

      = 18 * 210 * 28 * 29* bytes

= 18 * 27 MB

Disk Access Time

- 由下列 3 個時間加總
  - Seek Time
    - 將讀寫頭移到指定 Track 上方所花的時間
  - Latency Time
    - 將要存取的 Sector 轉到讀寫頭下方所花的時間
  - Transfer Time
    - 資料在 Disk 和 Memory 之間傳輸的時間
- Seek Time 最耗時
  - 機械動作
  - Disk Scheduling Algorithm
  - Track Service Request Arrival Order 服務請求的到達順序
- Buffer
  - 同時有多個服務請求，換先存放在 Buffer
  - 由 Disk Scheduling Algorithm 來決定服務順序

Disk Scheduling Algorithm

- FCFS
- SSTF
- SCAN
- C-SCAN
- LOOK --> Elevator
- C-LOOK
- 說明範例：目前讀寫頭位於 53 軌，磁碟緩衝區內的磁軌讀寫需求：
  98,183,37,122,14,124,65,67

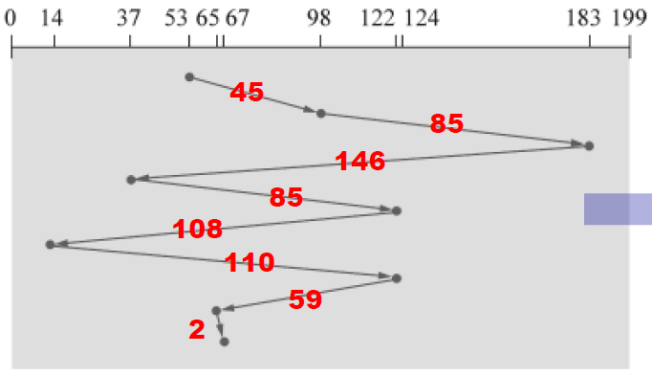
FCFS (First Come First Serve)

- 先到達的 Track Request 優先服務
- 範例結果：

  (98-53)+(183-98)+(183-37)+(122-37)+(122-14)+(124-14)+(124-65)+(67-65)

  = 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2

  = 640

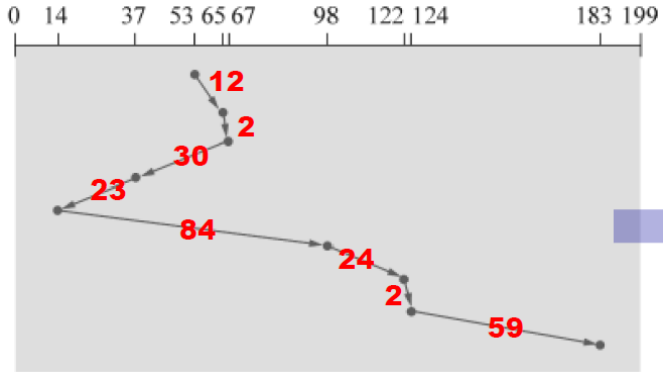


- 特點
  - 簡單
  - 公平，不會有飢餓
  - 效果不佳，`Average Seek Time` 長

SSTF (Short Seek Time First)

- 距離目前讀寫頭最近的 `Track` 優先服務
- 範例結果

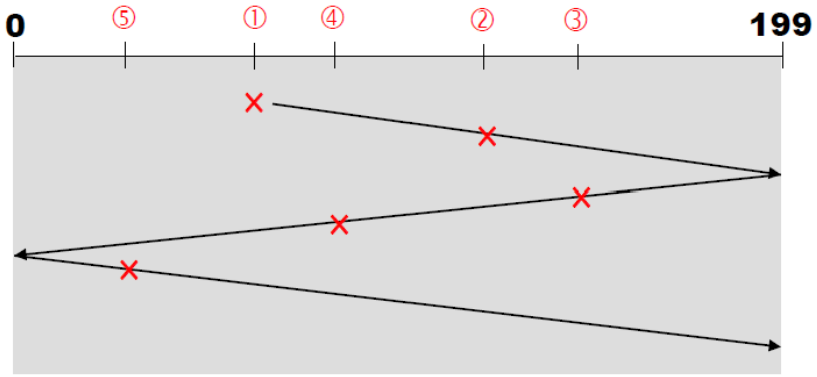
  `12 + 2 + 30 + 23 + 84 + 24 + 2 + 59 = 236`



- 特點
  - 並非是最佳，無法預測未來的服務需求
  - 不公平，可能會有飢餓 (如果後續進來的請求在讀寫頭附近，距離遠的 `Track` 會飢餓)

SCAN

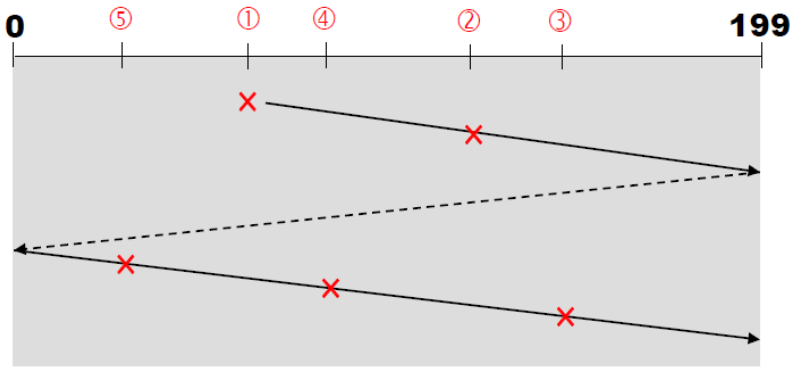
- 讀寫頭不斷來回掃描，有 `Track` 請求就服務
  - 提供**雙向服務**
  - 遇到 `Track` 起頭和尾端才折返
- 範例結果



- 特點
  - **不必要的磁軌移動**，因為要碰到起頭或是尾端才折返
  - 相對不公平，但是不會發生飢餓

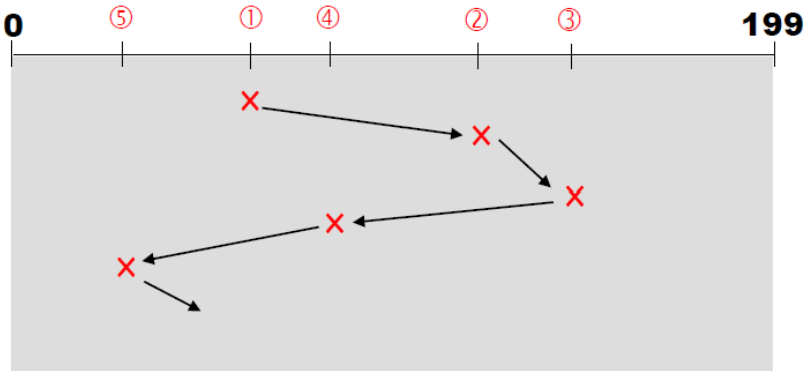
C-SCAN

- 讀寫頭不斷來回掃描，有 `Track` 請求就服務
  - 提供**單向服務**
  - 遇到 `Track` 起頭和尾端才折返



LOOK

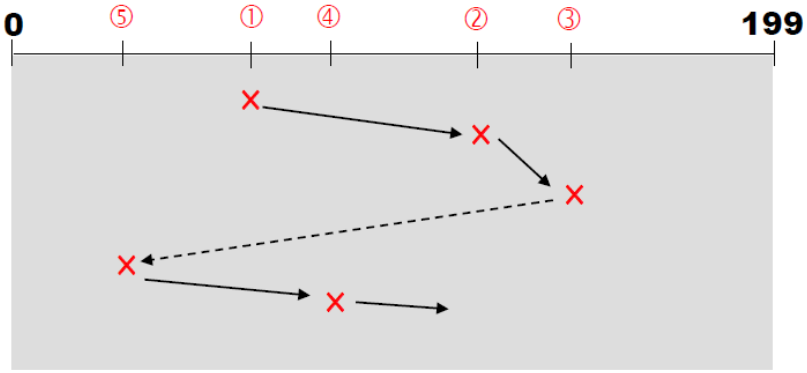
- 又稱為 `Elevator` 電梯演算法
- 讀寫頭不斷來回掃描，有 `Track` 請求就服務
  - 提供**雙向服務**
  - 處理完某個方向的最後一個 `Track` 服務立刻折返，不需要到 `Track` 起頭或尾端才折返



C-LOOK

- 讀寫頭不斷來回掃描，有 `Track` 請求就服務

- 提供**單向服務**
- 處理完某個方向的最後一個 Track 服務立刻折返，不需要到 Track 起頭或尾端才折返



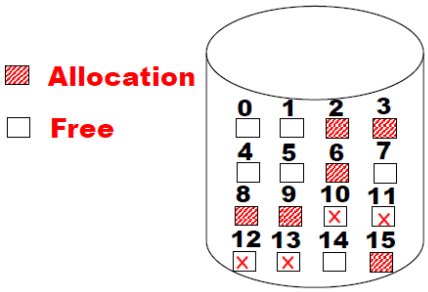
Allocation Methods 檔案配置方式

- Contiguous Allocation
- Linked Allocation
- File Allocation Table(FAT)
- Index Allocation

Contiguous Allocation

- 如果 File 的大小為 n Blocks，OS 必須在 Disk 上找到>=n 個連續 Free Blocks 才能配置
- **Physical Directory** 的記錄方式
  - File Name, Size, Starting Block#
  - **Physical Directory:** OS 要調用資料時查詢 Block 資訊
  - **Logical Directory:** User 看到的目錄，e.g. dir 或是檔案總館中的資訊
  - Logical Directory 轉 Physical Directory 由 OS 負責
- 範例: 若 File 1 要求 4 Blocks 空間
  - **File Name Size Starting Block#**

    File 1 4 10



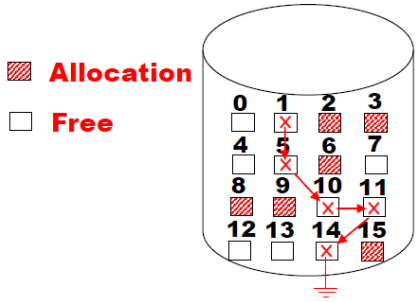
- 優點
  - 可支援 Sequential Access 和 Random Access
  - Seek Time 較短，File 所佔的連續區塊東常會在同一個 Track 或鄰近的 Track
    - Block 位置集中在同一個 Track 或相鄰的 Track
- 缺點

- 容易有**外部碎裂問題**
  - 如果 File 需要 7 Blocks，Disk 有 10 Free Blocks，但是不連續無法配置給 File
- 檔案大小無法任意增刪
- 建檔時須事先宣告大小

Linked Allocation

- 如果 File 的大小為 n Blocks，OS 必須在 Disk 上找到>=n 個 Free Blocks 才能配置 (不一定要連續)，各區塊用 Link 來串連
- Physical Directory 的記錄方式
  - **File Name, Starting Block#, End Block#**
  - **每一個 Block 必須提供幾個 byte 存放連結**
- 範例: 若 File 1 要求 4 Blocks 空間
  - **File Name Starting Block# Starting Block#**

    File 2 1 14



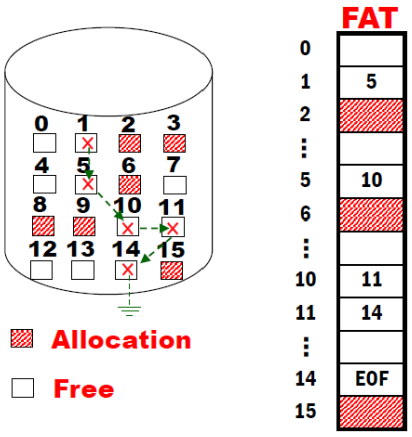
- 優點
  - 沒有外部碎裂問題
  - File 可以任意增刪空間
  - 建檔時不需要事先宣告大小
- 缺點
  - 不支援 Random Access，只支援 Sequential Access
  - Seek Time 比 Contiguous 配置方式長
    - Block 可能散落在不同 Track 上
  - Reliablility 差
    - 若 Link 斷裂 Data Block Lost

FAT

- Link Allocation 的變形，將所有配之區塊之間的 Link 集 Free Block 資訊記錄在 FAT
  - 若 Disk 有 n Blocks，FAT 會有 n 個項目 (所有 Block 都需要記錄)
  - FAT 資訊
    - **空白:** Free Block
    - **Block#:** 有 Link 指向此 Block，表示使用中
    - **EOF:** End of File

- MS-DOS
- Physical Directory的記錄方式
  - **File Name, FAT Index**
  - **FAT Index = 1表示由第1個Block開始**
- 範例：若File 1要求4 Blocks空間
  - **File Name FAT Index**

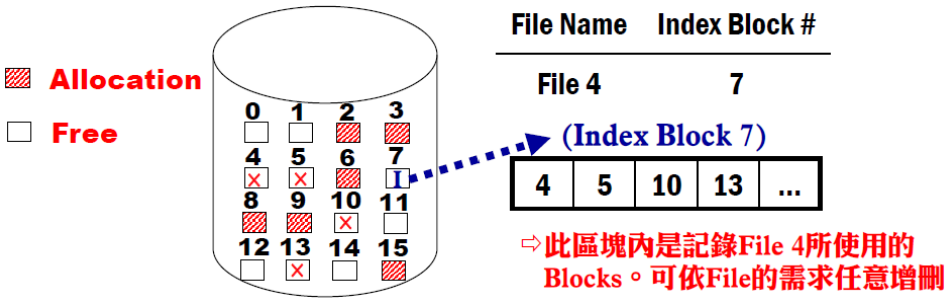
    File 3 1



- Block可以**全都存放Data**，不需要存放Link(和Link Allocation的差異)
- 優缺點同Link Allocation

Index Allocation

- 每個File都會多配置一些Index Blocks，內含Data Block編號，採非連續性配置
- Physical Directory的記錄方式
  - **File Name, Index Block#**
- 範例：若File 1要求4 Blocks空間
  - 會配置5 Blocks(1 Index Block + 4 Free Blocks)
  - 假設1 Index Block可以存10 Block#



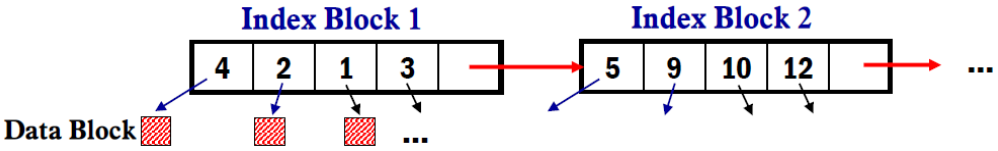
- 優點
  - 沒有外部碎裂問題
  - 支援Random Access和Sequential Access
  - File可任意增刪空間
  - 建檔時無需事先宣告大小
- 缺點

- Index Blocks 佔用額外空間
- Index Block 大小設計問題
  - 太小，不夠存放一個 File 所有的 Block 編號
  - 太大，浪費空間

解決 Index Block 不夠大的方法

方法一

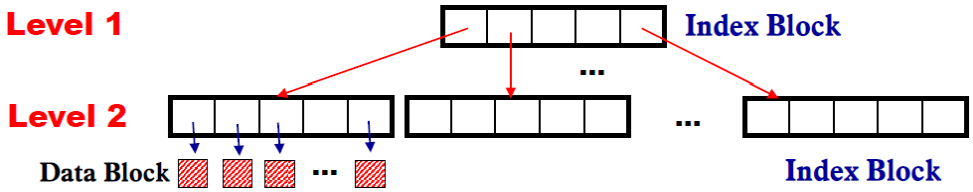
- 利用多個 Index Block 儲存，用 Link 方式串連



- 適用於小檔案
- 如果 Link 斷裂會造成 Data Lost

方法二

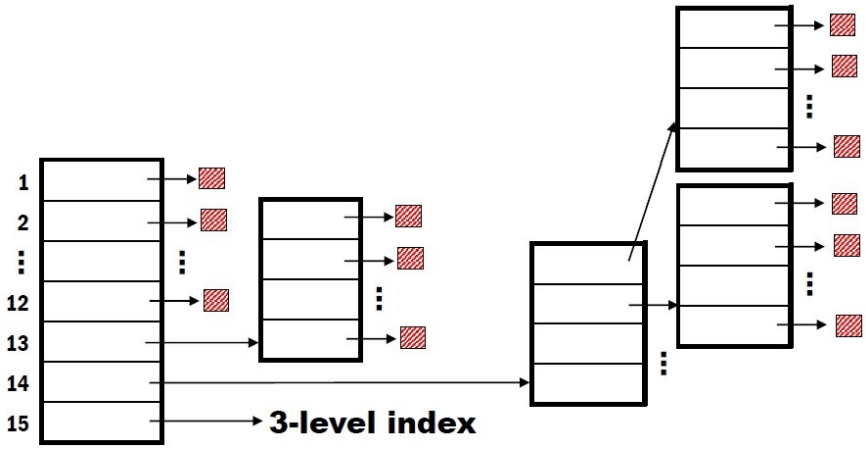
- Multilevel Index 多層索引
- 範例: 2 level index



- 適用於大型檔案，小型檔案不適用
  - 一旦系統的索引決定用 2 level，不論檔案大小都使用 2 level
  - Index 的大小可能大於小型檔案

方法三

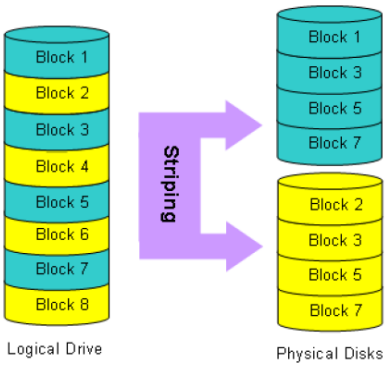
- 混用法，Unix - i-Node Structure
- 有 15 Pointer
  - Pointer No. 1 - 12: 儲存 Data Block#(Block 需求<=12)
  - Pointer No. 13: 指向 Single-Level Index --> 中型檔案(12 + n Blocks)
  - Pointer No. 14: 指向 Two-Level Index --> 大型檔案(12 + n + n2)
  - Pointer No. 15: 指向 Three-Level In --> 超大型檔案(12 + n + n2 + n3)



磁碟陣列 RAID

- 動機
  - Disk **效率差**，機械式運作
  - **安全性、可靠性**
- **多顆磁碟組成一個陣列，以 Striping (切割) 方式同時對不同磁碟做讀寫**
  - 將 1 byte 資料切割成 8 bit 分散儲存在 8 個磁碟，如果一次存取 1 byte 則可獲得 8 倍傳輸速度
  - 平行 I/O 提升效率
  - 安全性、可靠性提升
  - OS 負責 Striping 和分散存取，User 只能看到大且速度快的**邏輯磁碟空間**
- RAID Level: 技術不同的 RAID 代號
  - RAID 0, RAID 1, RAID 0+1, RAID 2, RAID 3, RAID 4, RAID 5, RAID 6...
  - Level 高低不代表效能或可靠度

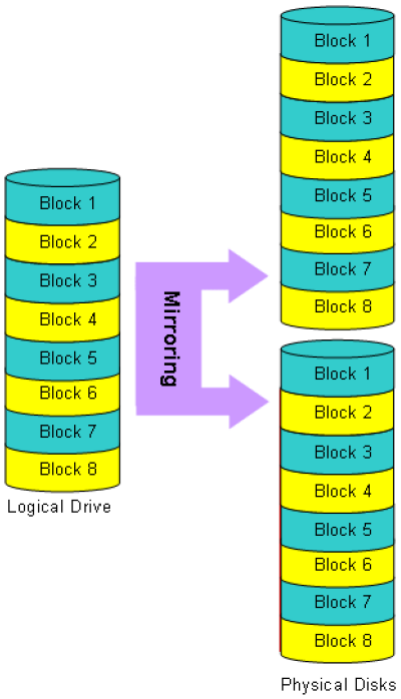
RAID 0



- Striping RAID
- 最少需要 2 顆硬碟
- 將資料切割存放到多顆硬碟，不會重複存放
- 效能導向，可平行讀寫，適用於高效能需求的系統
  - 理論上效能=[單一顆硬碟效能] * [硬碟數]
  - 實際上效能受到匯流排 I/O 和其他硬體影響，效能隨之遞減 (硬體溝通也有成本)
  - 所以兩顆硬碟的效能提升最明顯

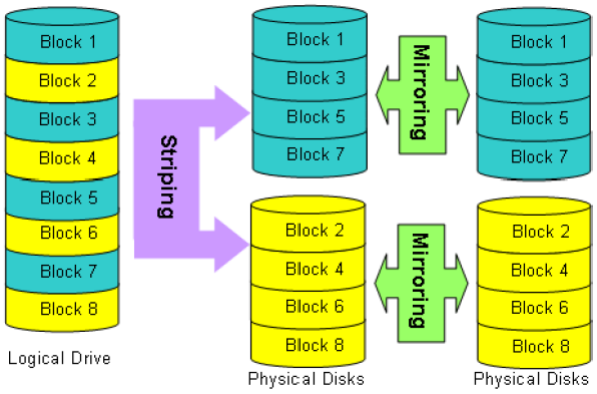
- 缺點：沒有容錯能力，只要有一顆硬碟故障，會導致所有資料損毀，無法挽回

RAID 1



- Mirror RAID
- 最少需要 2 顆硬碟
- **同時寫入兩顆硬碟，內部資料完全相同**
- 其中一顆硬碟作為備份，可靠性高
- Read 效能佳，Write 效能差
- 優點：可靠度高，容易實作
- 缺點：效率較差，成本高

RAID 0+1 (RAID 01) Striping + Mirror



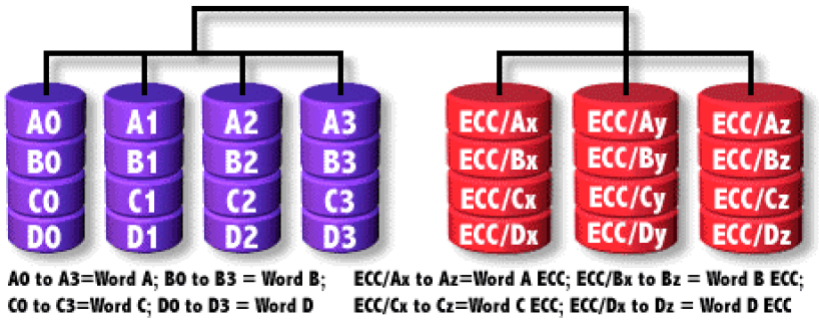
- 綜合 RAID 0 和 RAID 1 兩種模式
- 至少需要 4 顆硬碟，且硬碟數必須是偶數
- 先由 2 顆硬碟進行 RAID 0 的 Striping，再由另外 2 顆硬碟進行 RAID 1 的 Mirror
- 優點：效率高，可靠度高
- 缺點：成本高

RAID 1+0 (RAID 10) Mirror + Striping

- 綜合 RAID 0 和 RAID 1 兩種模式
- 至少需要 4 顆硬碟，且硬碟數必須是偶數
- 先由 2 顆硬碟進行 RAID 1 的 Mirror，再由另外 2 顆硬碟進行 RAID 0 的 Striping
- 優點: 可靠度比 RAID 高(先進行 Mirror)，效率較差
- 缺點: 成本高

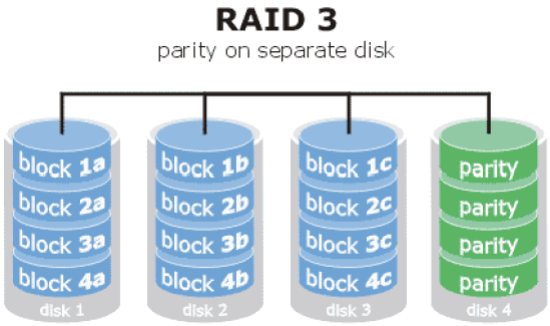
RAID 2

- RAID 0 的改良版，以 **Hamming Code** 的方式進行資料編碼
- 不同位元儲存在不同的硬碟
  - 4 bits 的資料需要 7 顆硬碟儲存，4 顆硬碟存資料，3 顆硬碟存 Hamming Code
- 加入錯誤修正碼 Error Correction Code(ECC)
  - 具錯誤檢查和修正能力，當一顆硬碟損壞，可由其他顆硬碟的內容來更正



RAID 3 Paraller with Parity 平行同位元檢查

- 資料切割以 **bit** 為單位
- **同位元檢查**
- 所有的同位元資料都存放在同一顆硬碟上(**Parity Disk**)
- 問題: Parity Disk 會成為效能瓶頸
  - 每次修改資料，可能只更動其中一顆硬碟，但是 Parity Disk 每次都必須更動



- 舉例:

```
   D1 D2 D3 D4 Parity
---------------------
a: 1  0  1  1  1
b: 1  1  0  0  0
c: 0  0  1  0  1
```

0. 假設 Disk 2 損毀，更正過程如下

```
   D1 D2 D3 D4 Parity
---------------------
a: 1  x  1  1  1
b: 1  x  0  0  0
c: 0  x  1  0  1
```

1. 利用 Parity Check Bit 更正錯誤訊息的內容 (必須滿足偶同位)

```
   D1 D2 D3 D4 Parity
---------------------
a: 1  x  1  1  1  -> Error Data = 0
b: 1  x  0  0  0  -> Error Data = 1
c: 0  x  1  0  1  -> Error Data = 0
```

2. 再找一顆正常的硬碟存入更正的內容

```
   D1 D5 D3 D4 Parity
---------------------
a: 1  0  1  1  1
b: 1  1  0  0  0
c: 0  0  1  0  1
```

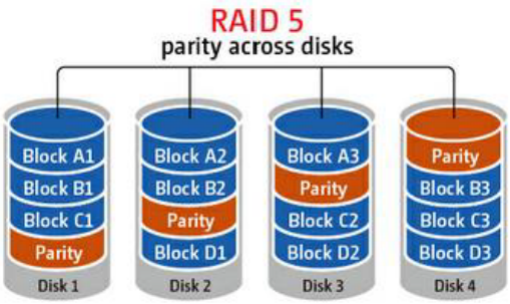
如果兩顆硬碟同時損毀，則無法用 Parity Check 更正錯誤內容

RAID 4 Parallel with Parity 平行同位元檢查

- 跟 RAID 3 相似，以 **Block** 為單位切割資料
- 效率較 RAID 3 高

RAID 5

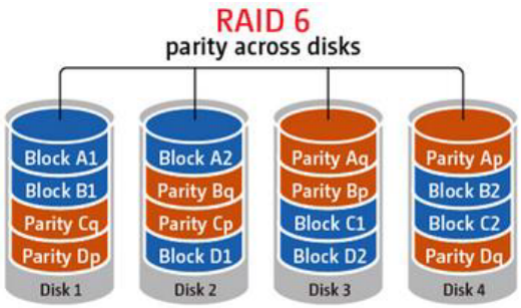
- 跟 RAID 4 相似，以 Block 為單位切割資料
- 把同位元**分散儲存於陣列中的每顆硬碟**



RAID 6

- 儲存 2 份同位元資料，即使 2 顆硬碟損換仍然能進行資料回復
- Write 效能較 RAID 5 差，資料寫入的同位元校正機制必須執行 2 次
  - **計算同位元值**

    - **驗證資料正確性**
- 資料重建時，回復時間比 RAID 5 多一倍



固態硬碟

- SSD 與 HDD 比較

SSD		HDD	
優點	電子式資料讀寫 抗震性佳	機械式資料讀寫 抗震性差	缺點
	散熱需求低???	散熱需求高???	
	Random Access	Read/Write Header	
缺點	成本高	成本低	優點
	容量小	容量大	
	顆粒損毀，資料無法救回	磁面損毀，資料有機會救回	

錯誤碼偵測與更正

- Parity Check 具偵測錯誤能力
- Hamming Code 具偵測錯誤能力和更正錯誤能力

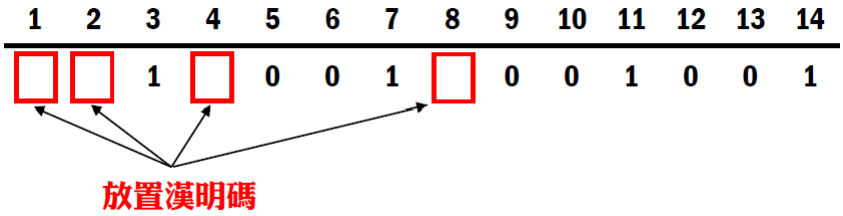
Parity Check

- 要傳輸的訊息加上 Parity Check Bit 所形成的傳輸碼
- Even Parity: 位元=1 的個數必須是偶數才算正確
- Odd Parity: 位元=1 的個數必須是奇數才算正確
- 範例
    - 採用 Even Parity，求下列傳輸訊息需補上的 Parity Check Bit
        - 訊息 1101000 --> Parity Check Bit = 1
        - 訊息 0110011 --> Parity Check Bit = 0
    - 下列傳輸碼符合 Odd Parity
        - 1111111 --> v
        - 1011100 --> x
        - 10000100 --> x
- 特點
    - 簡單，易於操作
    - 用於 ASCII、RAM 資料傳輸檢視、RAID(RAID 3, RAID 4, RAID 5)

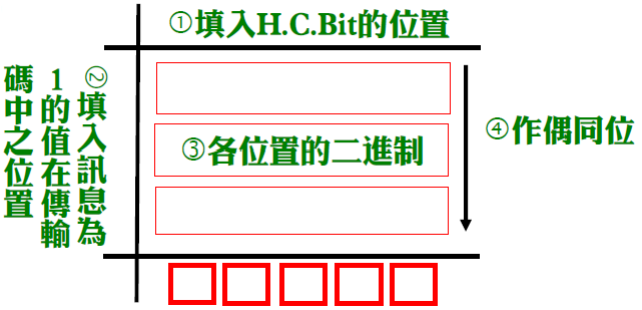
- 若有偶數個 bit 同時出錯，則收方無法偵測出錯誤，會誤判傳輸資料為正確
- 不具錯誤更正能力

Hamming Code

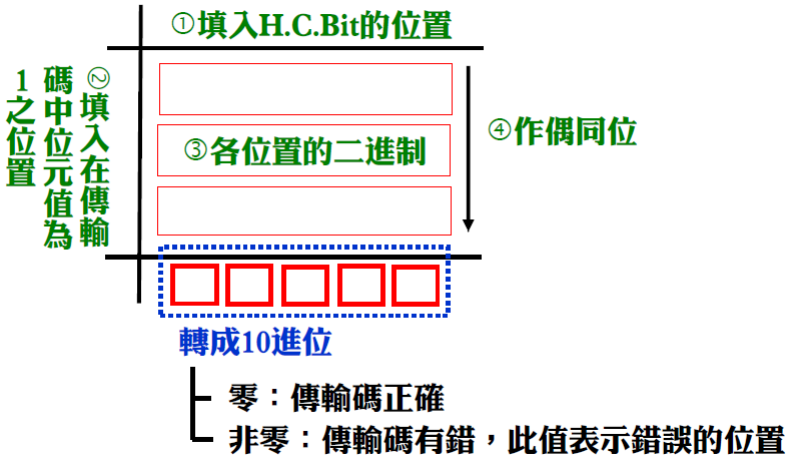
- 傳輸碼 = 傳輸訊息 + Hamming Code Bits(H.C. Bit)
  - 規定 H.C. Bit 一定要出現在 2 的冪次方位置(20, 21, 22, 23...)
  - 舉例: 1001001001
    - 傳輸碼:



- H.C. Bit 數目
  - 公式: 2k >= n + k + 1
  - n: 訊息長度
  - k: H.C. Bit 數，取最小的 k
  - 舉例: n = 10 --> 2k >= 10 + k + 1 --> 最小的 k = 4
- 如何決定 H.C. Bit 的值?



- 如何判斷傳輸碼是否有錯誤，哪個 Bit 出錯?



- 範例:

```
          1   2   3   4   5   6   7   8   9   10
-------------------------------------
```

Message 1 0 0 1 0 0 1 0 0 1

H.C. Bit 數目利用公式計算: n = 10 --> 2k >= 10 + k + 1 --> k = 4

| 8 4 2 1 H.C. Bit 的位置

---

3 | 0 0 1 1

7 | 0 1 1 1

11 | 1 0 1 1

14 | 1 1 1 0

| 0 0 0 1

--> 傳輸碼:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Message	1	0	1	0	0	0	1	0	0	0	1	0	0	1

假設收到的 H.C. Bit 傳輸碼如下，判斷並更正錯誤

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Message	1	0	1	0	0	0	1	0	**1**	0	1	0	0	1

| 8 4 2 1 H.C. Bit 的位置

---

1 | 0 0 0 1

3 | 0 0 1 1

7 | 0 1 1 1

9 | 1 0 0 1

11 | 1 0 1 1

14 | 1 1 1 0

| 1 0 0 1 --> 轉換成十進位=9(如果傳輸碼正確為 0)--> 錯誤位置在第 9 個位元

正確的 H.C. Bit 傳輸碼: 10100010001001

正確訊息: 1001001001 (移除 H.C. Bit)