

作業系統：Threads & Concurrency

目錄

作業系統：Threads & Concurrency	1
1. Background	1
1.1 單一執行緒 vs 多執行緒.....	1
2. 多核心程式設計（Multicore Programming）	2
3. Amdahl's Law（安達爾定律）	3
4. 多執行緒模型（Multithreading Models）	4
4.1 Many-to-One 模型（多對一）	4
4.2 One-to-One 模型（一對一）	4
4.3 Many-to-Many 模型（多對多）	5
5. 隱式多執行緒(Implicit Threading)	5
5.1 Thread Pool（執行緒池）	5
5.2 Fork-Join 模型（分叉-合併）	6
5.3 OpenMP	6
5.4 Grand Central Dispatch (GCD)	6
5.5 Intel TBB（Thread Building Blocks）	7

1. Background

執行緒（thread）是 CPU 利用的最基本單位，裡面包含：

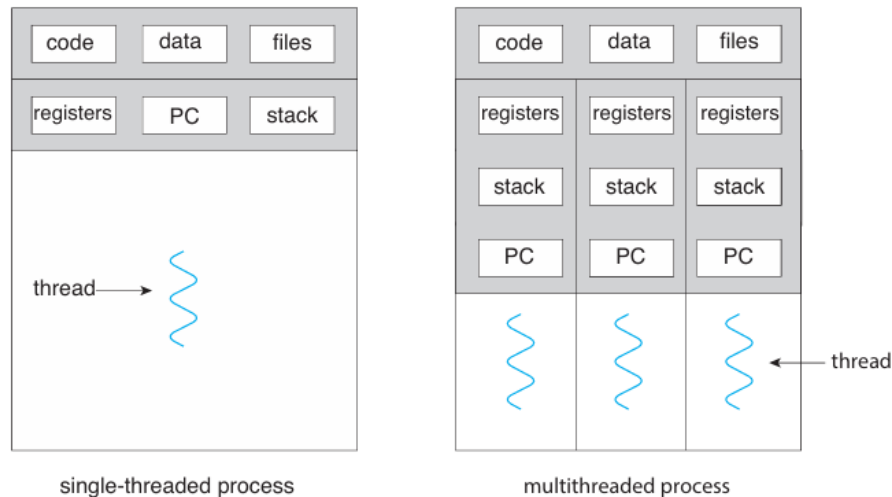
- Thread ID：唯一識別編號
- Program Counter：程式計數器，記錄下一個執行的指令
- Register Set：暫存器組（CPU 狀態）
- Stack：堆疊（函式呼叫、區域變數）

想像一個跑操場的情境：

- Process（行程）就像是一場比賽：一個比賽就佔用整個操場（CPU）。
- Thread（執行緒）就是比賽中的選手：一場比賽可以只有一位選手，也可以有很多選手同時跑不同的賽道。

1.1 單一執行緒 vs 多執行緒

Figure 1：Single-threaded (左)與 Multithreaded (右)



Multithreaded 的好處如下：

- 反應速度 (Responsiveness)：使用者操作不中斷，程式更即時互動，例如按下按鈕還能繼續使用介面
- 資源共享 (Resource Sharing)：同一個 process 中的 threads 天生就能共享資料、變數與資源
- 經濟效益 (Economy)：thread 比 process 更輕便，建立與切換成本更低
- 可擴展性 (Scalability)：在多核心系統上，執行緒可平行運作，效能提升更明顯

2. 多核心程式設計 (Multicore Programming)

為什麼現在的電腦動不動就說「8 核心」、「16 核心」？這些核心，說白了就是同時幫你做事的小幫手們。如果你只派給他們一個任務，就太可惜了！

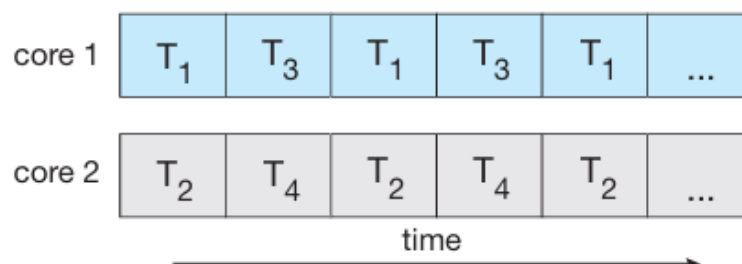
我們先搞懂兩個重要名詞：

- Concurrency (併發)：多個任務能交錯執行，看起來同時在跑（單核心也可以達成）
- Parallelism (平行)：多個任務真的同時在跑（多核心才能實現）

Figure 2：Concurrent execution on a single-core system.



Figure 3：Parallel execution on a multicore system.



其中 Multicore Programming 的挑戰：

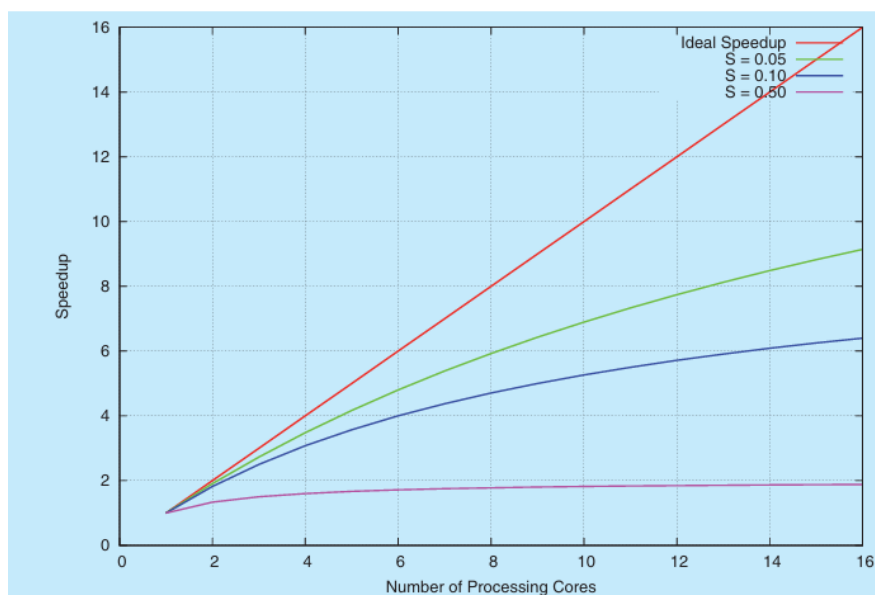
挑戰項目	說明
任務分解 (Identifying Tasks)	把程式拆成多個小任務 (thread)，而且這些任務要能獨立執行才行。
工作平衡 (Balance)	每個 thread 工作量要平均，不能一人爆肝、別人耍廢。否則會出現「其中一核心超忙，其他核心閒到發呆」的情況。
資料切分 (Data Splitting)	不只任務要分，資料也要分！讓每個核心各處理自己的那一塊，減少互搶資源。
資料相依 (Data Dependency)	如果 thread A 需要 thread B 的結果，那 A 就得等 B，這會讓平行變「卡頓」。這種「你等我我等你」的情況，要靠同步機制來處理。
測試與除錯 (Testing & Debugging)	多執行緒的錯誤難抓！bug 可能只在某個時間點、某個執行順序才出現，叫做「難以重現的地雷」。測試與除錯會比單執行緒程式更複雜。

3. Amdahl's Law (安達爾定律)

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

其中 S 為不能平行的部分 (serial)，N 為核心數量

Figure 4：AMDAHL'SLAW



假設一個程式：

- 75% 可平行 (0.75)

- 25% 必須串行執行 (0.25)

如果用：

- 2 顆核心 → 最快提升 ≈ 1.6 倍
- 4 顆核心 → 最快提升 ≈ 2.28 倍
- ∞ 顆核心 → 最大速度 $= 1 / 0.25 = 4$ 倍

重點：即使有 100 顆核心，若程式裡面有 25% 不能平行，最大速度也只能提升 4 倍。

4. 多執行緒模型 (Multithreading Models)

現代程式為了提升效率，常常會用「執行緒」來分工處理任務。但有個問題來了——使用者程式的使用者執行緒 (User Thread) 怎麼跟作業系統的核心執行緒 (Kernel Thread) 搭配呢？

- 使用者執行緒 (User Thread)：由應用程式與執行緒函式庫控制，在使用者空間中管理
- 核心執行緒 (Kernel Thread)：由作業系統內核直接支援與管理

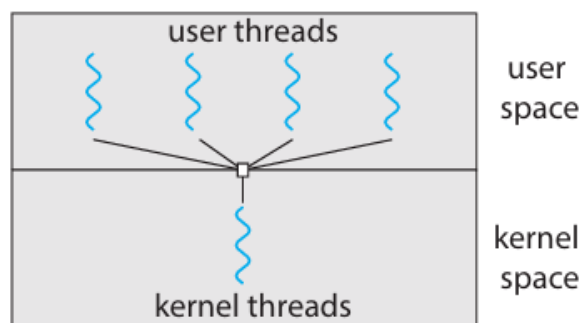
4.1 Many-to-One 模型 (多對一)

多個使用者執行緒 → 對應到 一個核心執行緒

- 優點：快速建立、上下文切換成本低
- 缺點：無法發揮多核心效能

因此，幾乎已被淘汰，因為無法發揮多核心的平行能力

Figure 5：Many-to-One 模型

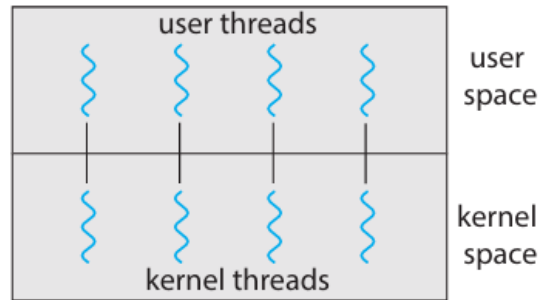


4.2 One-to-One 模型 (一對一)

每個使用者執行緒 → 對應到一個 核心執行緒，真正支援平行執行 (可利用多核心)。

- 阻塞 (Blocking) 不會影響其他執行緒
- 缺點：建立大量執行緒 = 建立大量核心執行緒 → 系統負擔大

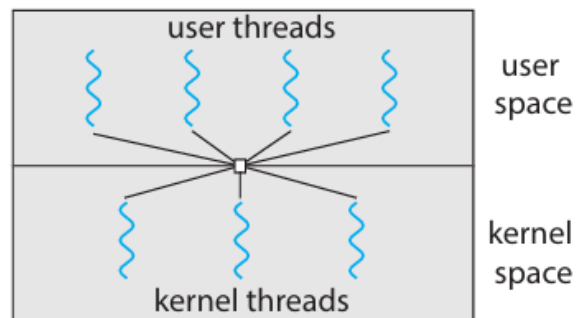
Figure 6：One-to-One 模型



4.3 Many-to-Many 模型（多對多）

多個使用者執行緒 ↔ 映射到一群（較少或等量）核心執行緒。可以建立「很多使用者執行緒」，核心只處理部分執行緒（核心與效能平衡）。支援平行處理，不會因為一個 thread 阻塞整個 process

Figure 7：Many-to-Many 模型



5. 隱式多執行緒(Implicit Threading)

在傳統的多執行緒程式設計中，開發者必須明確建立每個執行緒（thread creation）、指定工作分配（task assignment）、處理同步與互斥問題（如 mutex、semaphore）、這就像身兼老闆與搬運工，既要規劃又要親自操作，非常繁瑣且容易出錯。

而現代設計「隱式多執行緒」的概念，則將這些細節交給系統或執行環境代勞。開發者只需要定義工作項目（tasks），系統會自動判斷如何分配、何時執行，工作可能使用執行緒池、排程器或其他機制處理。這讓設計變得更像是「我提出工作需求，系統負責高效執行」，好比有個智慧工頭自動安排人力，開發者則可專注於邏輯設計與功能實現。

四種隱式多執行緒：

5.1 Thread Pool（執行緒池）

1. 預先建立一群執行緒（例如 10 個）
2. 當有任務來時 → 從池中取出 thread 來執行
3. 執行完 → 回收進池

優點：

- 效率高：省去頻繁建立與銷毀 thread 的開銷
- 控制數量：可限制 thread 數，避免耗盡資源
- 彈性排程：支援延遲執行、週期執行等策略

5.2 Fork-Join 模型（分叉-合併）

適用於「分而治之」的演算法（Divide and Conquer）

- 主 thread 把工作切割成多個子任務 → fork
- 等待子任務完成 → join → 整合結果

Figure 8 : Fork-Join in java

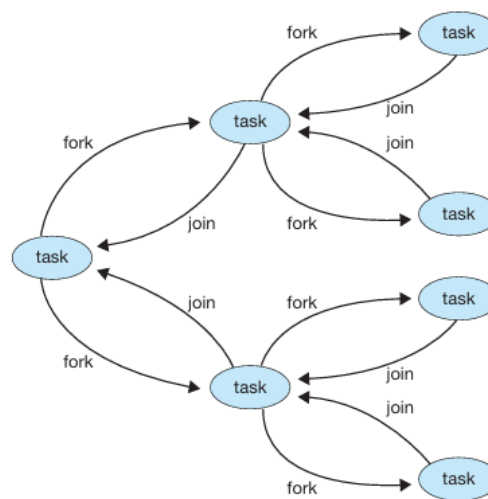


Figure 4.17 Fork-join in Java.

5.3 OpenMP

只要在 for 迴圈上面加一句「魔法註解 `#pragma omp parallel for`」，編譯器就會幫你自動把迴圈切開，讓不同核心處理不同段落。適用於 C/C++/Fortran。

```
// 假設你有 4 位工人要搬 100 箱貨物
// 這句話的意思就是：「請幫我自動分配這 100 箱工作，分給不同的工人（CPU 核心）一起做！」

#pragma omp parallel for
for (int i = 0; i < 100; i++) {
    c[i] = a[i] + b[i]; // 每一箱做一次加法
}
```

5.4 Grand Central Dispatch (GCD)

GCD 是 Apple 發明的一種「任務派送中心」，你只要把任務丟到「佇列」(queue)，它會幫你找空閒的核心來執行。Apple 專用。其中 GCD 的兩種 queue：

作業系統：Threads & Concurrency

- Serial Queue：任務一個一個來，順序執行（像排隊）
- Concurrent Queue：任務可以同時跑（像排一排人然後一起搬貨）

```
// 請幫我找一個空的核心，幫我執行這段任務，等一下可以印出 'This is a concurrent task.'  
let queue = DispatchQueue.global(qos: .userInitiated)  
queue.async {  
    print("This is a concurrent task.")  
}
```

5.5 Intel TBB (Thread Building Blocks)

它是 C++ 的一個函式庫，幫你自動做平行處理。你只要告訴我你有一堆工作，我來幫你分配給 CPU 跑，跑得快還自動幫你調整。

```
parallel_for(size_t(0), n, [=](size_t i) {  
    apply(v[i]);  
});
```