作業系統：CPU Scheduling

目錄

1. Basic Concepts

在單核心系統中，一次只能執行一個 process，其他 process 必須等待。為什麼需要 CPU 排程（Scheduling）？

- 讓 CPU 不要閒著：當某個程式因 I/O 卡住，就切去跑別的程式。
- 多個程式同時保留在記憶體中，讓 CPU 每次都有事情做 → 提升效能。

1.1 CPU – I/O Burst Cycle（爆發週期）

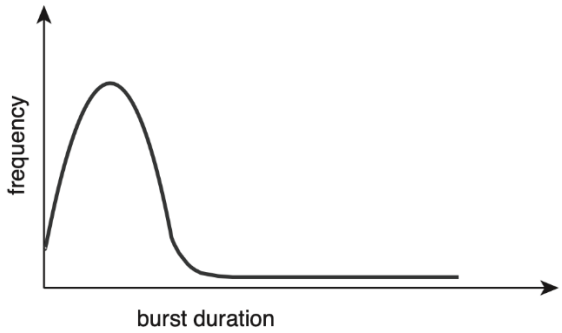
每個 process 執行的模式：CPU Burst → I/O Burst → CPU Burst → I/O Burst → … 最後結束。其中：

- CPU Burst： 使用 CPU 計算的時間
- I/O Burst： 等待輸入輸出（如：讀檔案、網路請求）的時間

針對於 CPU Burst 這裡面可以細分：

- 多數是短時間（exponential 分布），少數是長時間
- I/O-bound 程式：很多小 CPU burst → 等 I/O
- CPU-bound 程式：少量但長 CPU burst → 不太等 I/O

Figure 1：Histogram of CPU-burst durations



2. Preemptive vs Nonpreemptive Scheduling（可搶佔與不可搶佔）

- Nonpreemptive（不可搶佔）： process 一旦拿到 CPU，除非自願放棄或終止，否則不會被中斷。
- Preemptive（可搶佔）： 現代 OS 幾乎都支援這種排程（像 Linux、Windows）

有四種情況會觸發排程決策：

1. Process 從 running → waiting（例如：要 I/O）=> 非搶佔式（Nonpreemptive）
2. Process 從 running → ready（例如：被中斷）=> 搶佔式（Preemptive）
3. Process 從 waiting → ready（I/O 完成）=> 搶佔式（Preemptive）
4. Process 結束=> 非搶佔式（Nonpreemptive）

可搶佔排程可能引發 race condition（資料競爭），需要額外保護機制（像鎖或關中斷）

3. Dispatcher（派遣器）

主要功能：當 scheduler 選好了要執行的 process，dispatcher 負責交接 CPU 給它。

1. 儲存舊 process 的 context 到 PCB => 還原新 process 的 context
2. 切到使用者模式
3. 跳至新程式該跑的程式碼位置

Dispatch Latency（派遣延遲）：指一次 context switch 所花的時間

4. Scheduling Criteria（排程準則）

準則 01	CPU Utilization（CPU 利用率）	概念：CPU 的忙碌程度，越忙表示使用越有效率。 查詢方式：使用 top 指令（Linux/macOS）
準則 02	Throughput（吞吐量）	概念：每單位時間完成幾個 process。 越多代表系統越有效率。
準則 03	Turnaround Time（周轉時間）	概念：從程式提交到完成的整體時間 包含：等待時間（在 ready queue）、CPU 執行時間、I/O 處理時間 Turnaround Time ＝ Completion Time - Arrival Time
準則 04	Waiting Time（等待時間）	概念：只算在 ready queue 裡乾等的時間 不包含：執行 CPU 的時間、I/O 的時間 是 CPU 排程演算法的主要影響範圍
準則 05	Response Time（回應時間）	概念：從提出請求到第一次反應出現的時間 跟 Turnaround 不同，不是整個完成，只是開始回應（按下 Enter → 等看到第一行輸出）

5. Scheduling Algorithms

CPU 排程的目的是：從 Ready Queue 中選擇一個 Process 執行。以下演算法皆以「單核心系統」為基礎說明。

5.1 First-Come, First-Served（FCFS）

最簡單的排程方式，誰先來誰先跑（FIFO），屬於非搶佔式（nonpreemptive）。缺點是會出現 Convoy Effect（車隊效應）長 process 擋住後面所有 process，I/O 效能浪費

範例：假設我們有 3 個 process

Process	Burst Time（執行時間）
P1	24 ms
P2	3 ms
P3	3 ms

如果以原始順序(P1, P2, P3)去跑的話，平均等待時間為(0+24+27)/3 = 17ms，而如果以短程先跑(P2, P3, P1)去跑的話，平均等待時間為(0+3+6)/3 = 3ms，明顯縮短平均等待時間！

Figure 2：原始順序(P1, P2, P3)

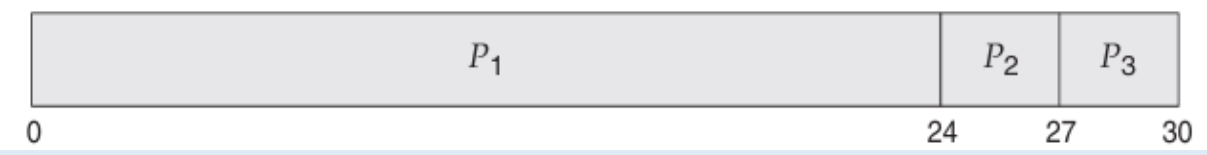
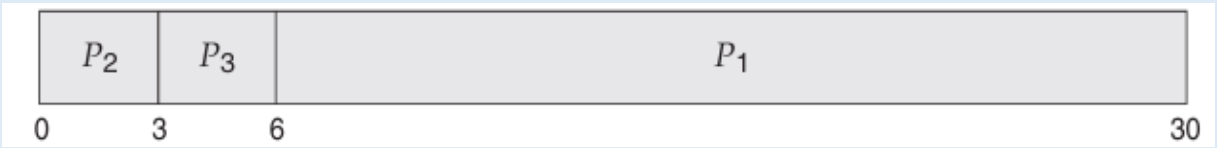


Figure 3：短程先跑(P2, P3, P1)



5.2 Shortest-Job-First（SJF）

執行下一個 CPU burst 最短的 process，可為非搶佔式（non-preemptive） 與 搶佔式（preemptive，又稱 SRTF）。優點是最小平均等待時間（理論上最佳），而缺點是很難準確預測 CPU burst 時間。

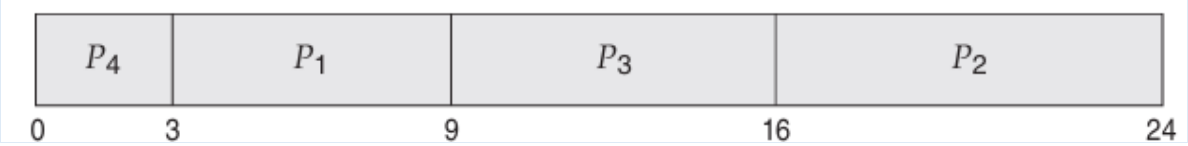
非搶佔 SJF 範例：

假設四個 process 都同時到達，Burst Time 如下

Process	Burst Time（執行時間）
P1	6 ms
P2	8 ms
P3	7 ms
P4	3 ms

其等待時間為(0+3+9+16)/4 = 7ms

Figure 4：非搶佔 SJF 排成順序



搶佔式 SJF（SRTF）範例

假設四個 process 都同時到達，Burst Time 如下

Process	Arrival Time	Burst Time（執行時間）
P1	0	8 ms
P2	1	4 ms
P3	2	9 ms
P4	3	5 ms

Table 1：完整計算等待時間

Process	結束時間	開始執行	Arrival	Waiting Time
P1	17	1+10	0	(1-0) + (10-1) = 10
P2	5	1	1	0
P3	26	17	2	17 - 2 = 15
P4	10	5	3	5 - 3 = 2

其平均等待時間為(10+0+15+2)/4 = 6.75ms

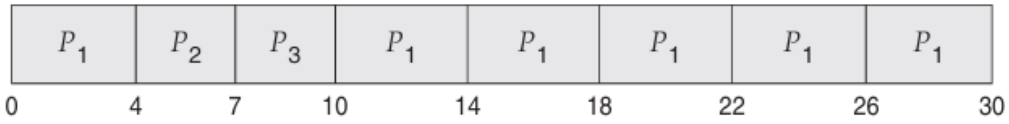
5.3 Round-Robin（RR）

RR 為 FCFS 加上 搶佔（Preemption），每個 process 分得一段固定的時間 → Time Quantum

範例：假設我們有 3 個 process，其 time quantum = 4

Process	Burst Time（執行時間）
P1	24 ms
P2	3 ms
P3	3 ms

Figure 5：RR 排成



Time Quantum 設定關鍵：

- 太大 → 變成 FCFS
- 太小 → 過多 context switch，浪費效能

5.4 Priority Scheduling

每個 process 有一個「優先順序」，數字越小，優先度越高。可為 preemptive 或 nonpreemptive。其缺點是可能造成 Starvation（飢餓）：低優先程式永遠等不到。

解法：Aging（等待越久，優先度逐漸提高）

範例：假設我們有 5 個 process

Process	Burst	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

執行順序：P2 → P5 → P1 → P3 → P4，平均等待時間 = 8.2ms

5.5 Multilevel Queue Scheduling

根據「process 類型」分成多個隊列，例如：

- Real-time processes
- System processes
- Interactive processes
- Batch processes

其特徵是：

1. 每個 queue 有獨立的排程策略（如前面提到的 FCFS 或 RR）
2. Queue 之間的排程通常採用 Fixed-Priority Preemptive Scheduling
3. 低優先 queue 會被高優先 queue 抢走 CPU

此外，可以設計每個 queue 分配不同 CPU 時間比例（Time Slicing）。

Figure 6：Separate queues for each priority

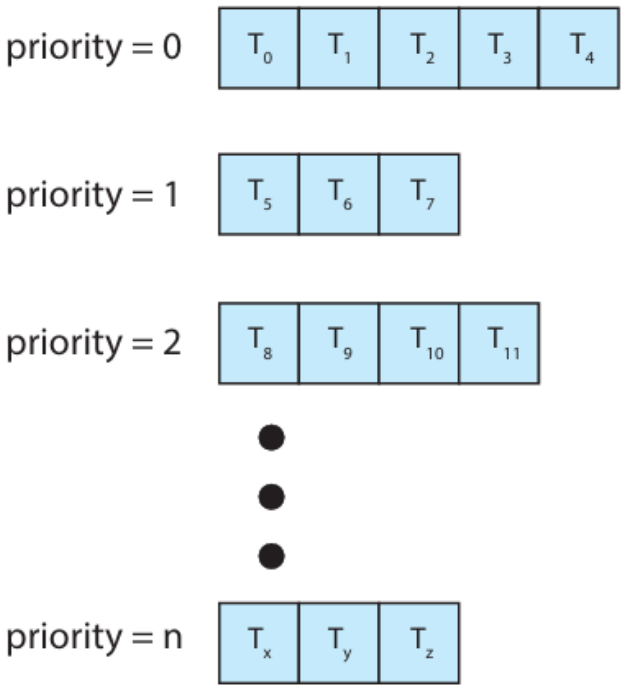
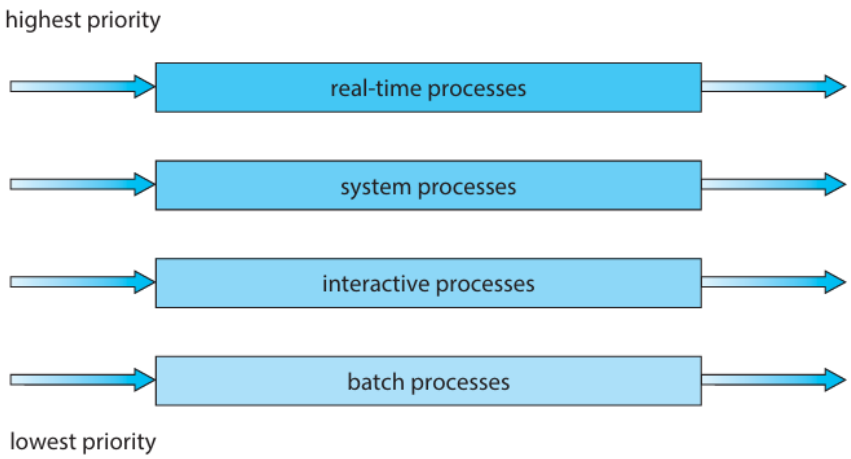


Figure 7：Multilevel queue scheduling



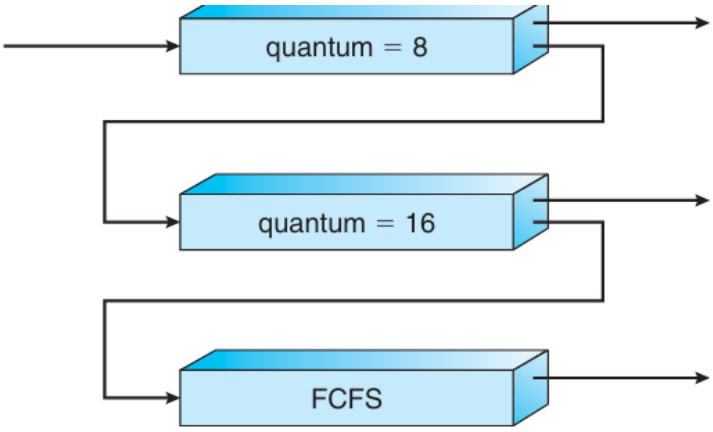
5.6 Multilevel Feedback Queue Scheduling

與 Multilevel Queue 最大不同 → 允許 process 在 queue 之間移動。移動規則根據 CPU burst 長度或等待時間（防

止飢餓）。優點是綜合各家排程策略，適應性強。缺點是複雜度高，參數多。常見設計：三層 queue：

- Queue 0（優先）: RR，quantum=8
- Queue 1：RR，quantum=16
- Queue 2：FCFS

Figure 8：Multilevel feedback queues



6. Thread Scheduling

在現代作業系統中，CPU 排程不再只針對「行程（process）」，而是排程「核心層級的執行緒（kernel-level thread）」。意味：作業系統會直接對「核心執行緒」進行排程。而「使用者執行緒」必須透過與其對應的核心執行緒才能真正執行在 CPU 上。

- User-level Thread（使用者層級執行緒）：由使用者程式的 thread library（例如 Pthreads）管理。作業系統「看不到」它們。
- Kernel-level Thread（核心層級執行緒）：作業系統可以排程和管理。真正執行在 CPU 上的是它。

LWP（Lightweight Process）：一種核心支援的結構，用來「橋接」使用者執行緒與核心執行緒。可以把它當作「使用者 thread 與 kernel thread 的連接點」。

7. 排程範圍（Contention Scope）

定義：一條執行緒到底是在和誰搶 CPU？

		情境
PCS（行程層級競爭）	同一個 Process 裡的 Thread 搶	部門內升遷：你跟你自己部門（process）裡厂的同事競爭升職。升職名額只有一個（代表 CPU），誰升上去是部門內部自己排的，跟其他部門無關。
SCS（系統層級競爭）	整個系統所有 Thread 都搶	公司整體升遷：你這次不是跟自己部門的人搶，而是全公司的人都來搶這個升職名額（CPU）。

8. Multicore Processors（多核心處理器）

記憶體速度跟不上 CPU，便會產生問題。

8.1 Memory Stall

當 CPU 等待資料從記憶體傳回來，而 無法繼續執行 的這段時間，就叫做 Memory Stall。就像是你煮飯很快，但食材還沒送來，你只能乾等。

解決方式：Multithreaded Core（多執行緒核心）

一個核心內放 2 條以上硬體執行緒（hardware thread），當 Thread 1 等資料（stall），切去跑 Thread 2。在 Intel 上，這叫 Hyper-threading（超執行緒）

Figure 9：Multithreaded multicore system

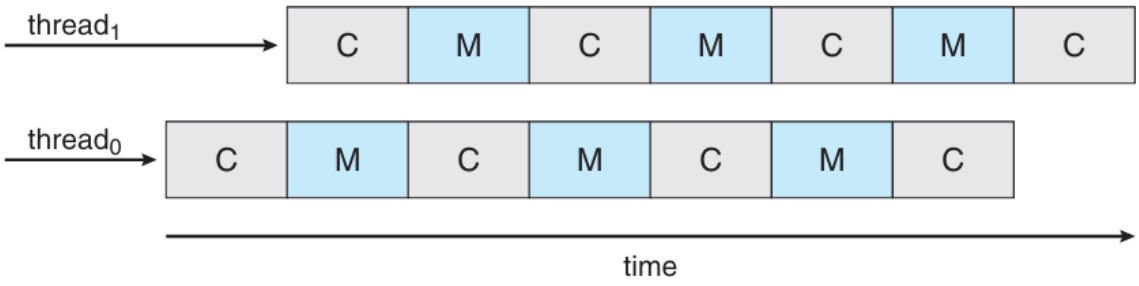
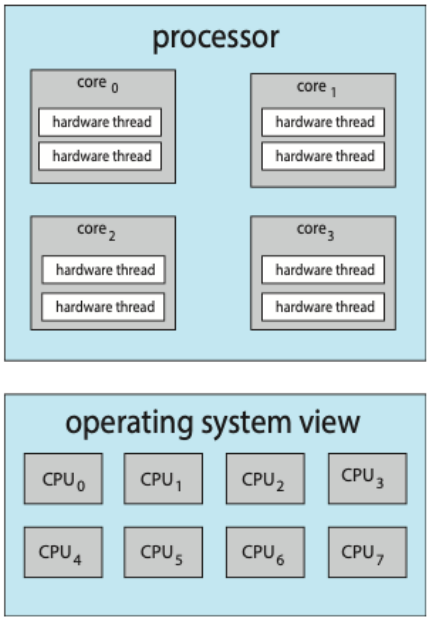


Figure 10：Chip multithreading

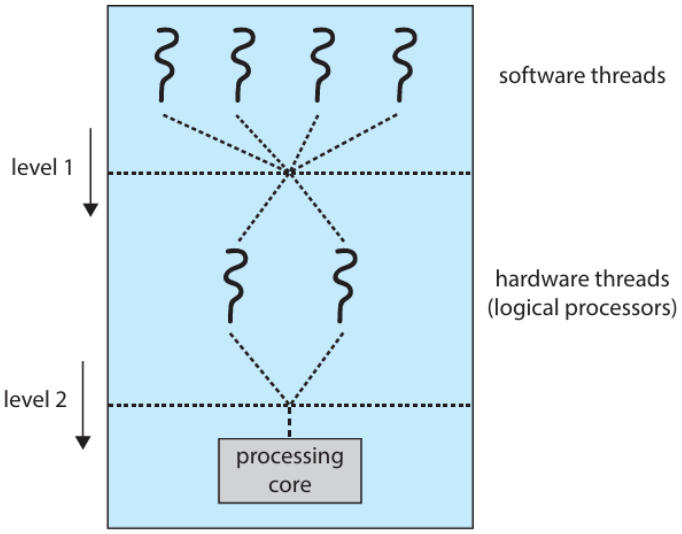


8.2 兩層排程（Two-Level Scheduling）

Level 1：OS 決定哪條「軟體執行緒」要跑在哪個邏輯 CPU（hardware thread）。Level 2：每個核心內部決定執行哪一條硬體執行緒。

有些處理器如 Intel Itanium，會根據 urgency（緊急值）來選擇哪一條 thread 跑。

Figure 11：Two levels of scheduling



8.3 Load Balancing（負載平衡）

為什麼需要？→ 防止一顆核心超忙、其他核心閒著。兩種負載平衡方式：

- Push migration：主動定時檢查，把任務從忙的核心移到閒的核心
- Pull migration：閒的核心主動拉，閒核心找工作來做

8.4 Processor Affinity（處理器傾向性）

如果一條 thread 一直在同一核心跑，那核心的 cache 就會有它的資料 → 速度快！

Affinity 的兩種形式：

- Soft affinity：OS 嘗試維持執行緒在同一核心，但不保證
- Hard affinity：可以用系統呼叫指定「只能跑在某幾顆 CPU」上

9. NUMA 架構下的問題

在傳統的對稱多處理（SMP）系統中，所有的處理器共用一個主記憶體，從任一個 CPU 存取記憶體的速度基本上是相同的。但這種設計在系統變大（例如有很多個核心）時會變得沒效率，因為所有的 CPU 都要搶同一個記憶體資源。因此，現代高效能的多處理器系統使用了一種架構叫做 NUMA。NUMA = Non-Uniform Memory Access（非一致性記憶體存取）。系統中每一顆 CPU（或 CPU 群組）會擁有自己的本地記憶體。也就是說，CPU0 擁有記憶體 A，CPU1 擁有記憶體 B，等等。而記憶體存取有快慢差別：

- 如果 CPU0 存取自己的本地記憶體 A → 很快
- 如果 CPU0 存取 CPU1 的記憶體 B → 比較慢（要透過系統匯流排）

這就是「非一致性」的由來：不同位置的記憶體有不同的存取速度。

當系統執行程式時，作業系統會將 thread 排程到某顆 CPU 上執行。這個 thread 用到的資料也會被配置在與該 CPU 靠近的本地記憶體。但如果為了「負載平衡」，作業系統把 thread 從 CPU0 移到 CPU1 執行：

- 該 thread 的資料還留在 CPU0 的記憶體中
- CPU1 要使用這些資料時，必須遠端存取 → 存取速度變慢 → 效能下降

解法：

1. 在排程 thread 時，盡可能讓它留在原來的 CPU 上（保持資料的「區域性」）
2. 在配置記憶體時，把資料放在靠近 thread 執行所在的 CPU 的記憶體中
3. 如果真的要移動 thread，也要考慮把資料一起搬過去，雖然成本高

10. Real-Time CPU Scheduling

real-time 作業系統（RTOS）需要能夠即時處理重要任務。依照嚴格程度可以分為兩類：

- soft real-time system：不保證準時執行，只保證 real-time 任務的優先權比其他任務高。
- hard real-time system：絕對要求在 deadline 前完成工作，逾時即算失敗。

11. 事件延遲（latency）與即時性問題

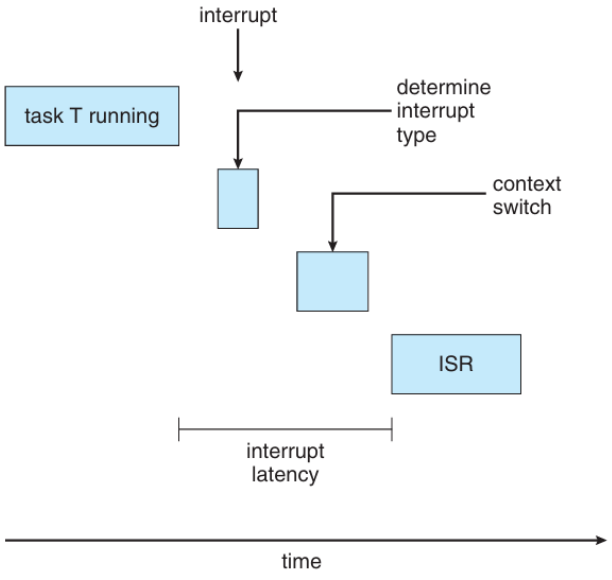
事件延遲（event latency）：從事件發生到系統開始回應這段時間。real-time 系統通常是事件驅動的。當事件發生後，系統必須儘快回應。舉例而言：

- 車輛防鎖死煞車系統（ABS）的容忍延遲是 3 到 5 毫秒。
- 飛機雷達控制器可容忍幾秒鐘。

11.1 interrupt latency

中斷發生到開始執行 ISR 的時間，包含完成目前指令、識別中斷類型、儲存上下文、執行中斷服務程式（ISR）。

Figure 12：Interrupt latency



11.2 dispatch latency

OS 從停止一個 process，到啟動另一個 process 所需的時間。

Figure 13：Dispatch latency