行程

Process 行程

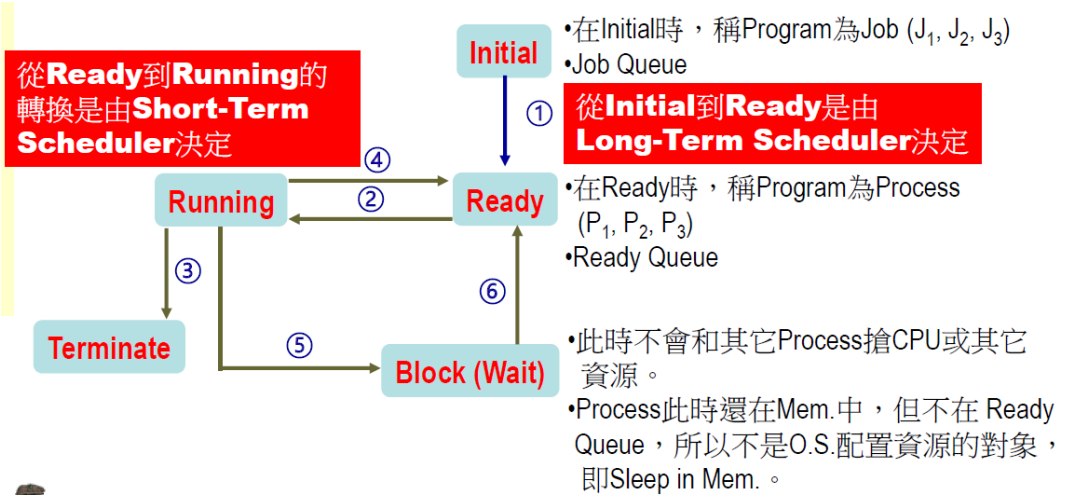
- **正在執行中的程式(A program in execution)**
- 包含
  - **Code Section** 程式碼、程式區間 } **存放於記憶體**
  - **Data Section** 資料區間 } **存放於記憶體**
  - **Program Counter** 程式計數器
    - 下一個要執行的指令所在位址
  - **CPU Register**
    - 通用暫存器、基底(限制)暫存器
  - **Stack**
    - 多個 **Process 互相呼叫**、**遞迴工作**，用以**存放返回位址**
- Process 是 OS 分配資源的對象
- 程式未執行時，只是存放在硬碟中的**檔案**

Process	Program
主動(Active Entity)	被動(Passive Entity)
執行中的程式 (有 Program Counter)	儲存在次儲存體中的檔案

Process State Transition Diagram 行程狀態圖

- Process 在執行時會改變其狀態

- **Process STD** 用以描述 **Process** 由開始到結束的生命週期



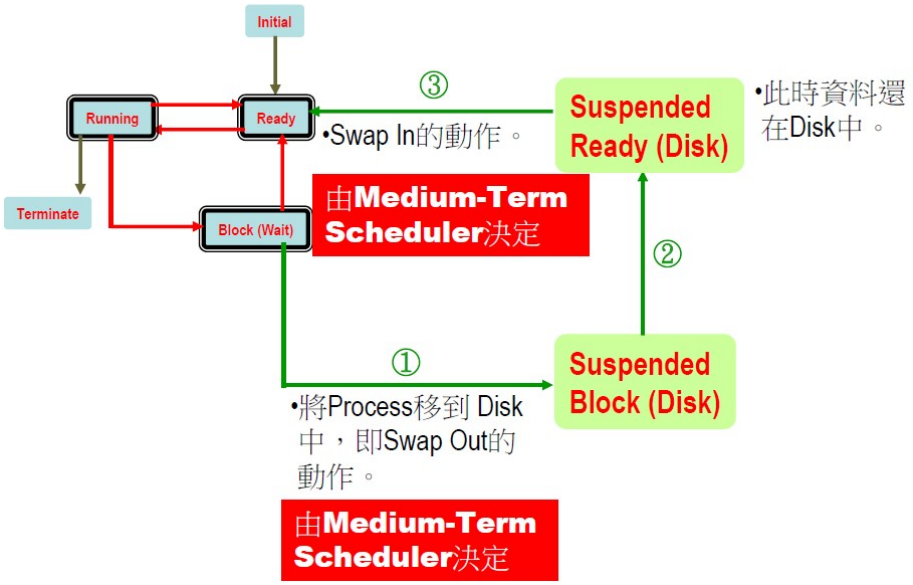
狀態圖行為

1. 引入(或產生)一個新的 **Program** 到電腦去執行
2. 從記憶體中挑選一個 **Process** 到 **CPU** 去執行
3. Process **完成工作**後正常結束；Process **發生不正常結束**就中止
   - 運算除 0
   - 溢位
4. 發生**短暫中止**(被迫中止)，會直接回到 Ready 狀態
   - 被高優先權 Process 插隊
   - 中斷發生
   - 超過 CPU Time Quantum
5. 發生**較長時間中止**，會將 Process Block
   - 等待 I/O Complete
   - 等待 Resource Available
   - 不會和其他 Process 搶 CPU 資源
   - Process 還在記憶體中，但不在 eady Queue

- 此 STD 是針對 CPU 資源，且資料都在記憶體中

Process Scheduling Queue

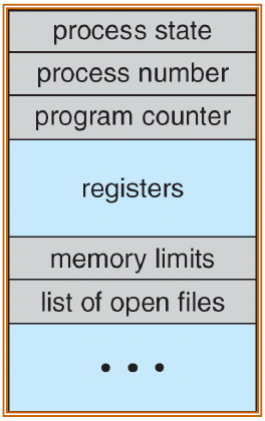
- **Job Queue**
  - 在次儲存體中，等待進入主記憶體的 Program 的集合
- **Ready Queue**
  - 在主記憶體中，就緒並等待執行的 Process 的集合
  - 此佇列一般都是用 Linked List 方式儲存
- **Device Queue**
  - 等待 I/O 裝置的 Process 的集合
  - 每個裝置都有各自的 Device Queue，記錄不同 Process 的請求



1. 當 Process **待在主記憶體的時間太長**(被 Block 太久)，或有其他高優先權的 Process 來**搶主記憶體的資源**
2. 所有等待的 Long-Time Event 發生，或是花費長時間的事情做完了
   - e.g. Long-Time I/O Complete
3. 將 Process 從 Disk 引入主記憶體中的 Ready Queue

- 此 STD 是針對主記憶體，且資料都在硬碟中
- **一個 Process 的執行時間**是**CPU 執行時間**和**I/O 等待時間**組成
- 幾乎每個 Process 都會在兩種工作狀態切換
  - **CPU Burst (CPU Bound Job)**
  - **I/O Burst (I/O Bound Job)**
  - **Process 一開始一定是 CPU Burst**，也就是開始交由 CPU 處理該 Process
  - 接著是 CPU Burst, I/O Burst 兩個狀態下切換
  - 最後 **CPU 必須呼叫一個終止執行的 System Call 結束行程 (CPU Burst)**

Process Control Blcok (PCB) 行程控制表



- OS 為了執行 Process Management，建立一個集合**儲存每一個 Process 的所有相關資訊**
- 每個 Process 皆有自己的 PCB
- PCB 包含
  - **Process ID**

    - **處理行程狀態**位於 Process STD 的哪一個狀態
    - **程式計數器**紀錄 Process 下一個要執行的指令位址
    - **CPU 暫存器**
    - **CPU 排班資訊** e.g. 優先權值
    - **記憶體管理資訊** Base/Limit Register 內容、Page Table 的相關資訊
    - **帳號資訊**用掉多少 CPU 時間、CPU 使用的最大時間量
    - **I/O 狀態資訊**尚未完成的 I/O Request、I/O Queue 中排隊的 Process 編號
- **PCB 存在 Monitor Area 中**

排班程式

Long-Term Scheduler

- **目的：從 Job Queue(e.g. Disk) 挑選合適的 Job 載入記憶體內準備執行**
- **特徵**
    - 執行頻率最低
    - 可調控 **Multiprogramming Degree**
    - 可調整 **CPU Bound 和 I/O Bound 的混合比例**
    - 適用於 **Batch System**
    - 不適用於 Time-Sharing 和 Real-Time System

Short-Term Scheduler

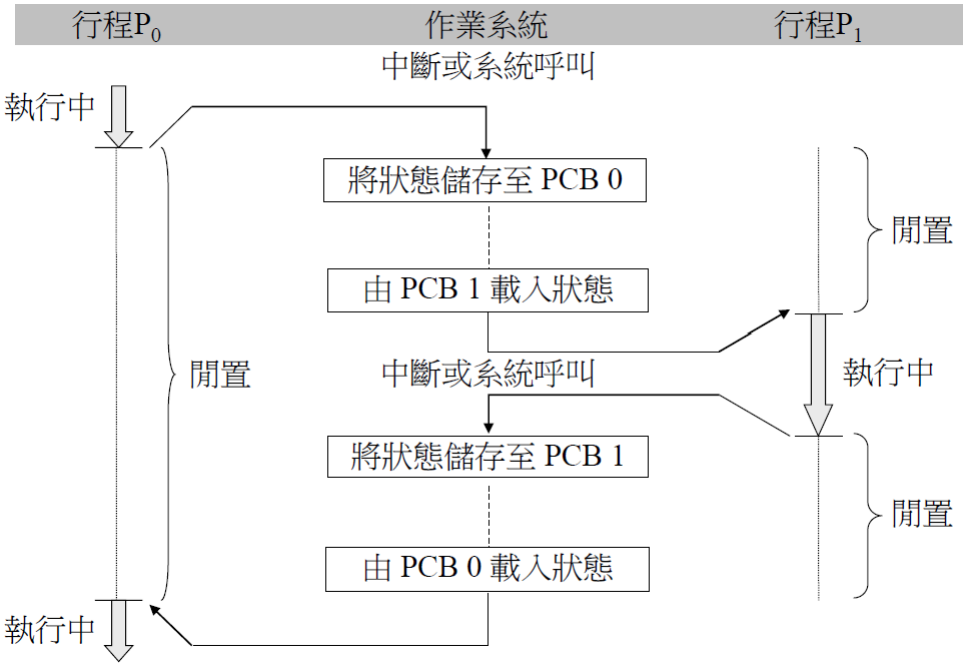
- **目的：**從 **Ready Queue**(主記憶體) 挑選一個已經 Ready 且適當的 Process，讓他獲得 CPU 控制權來執行
- 只負責挑工作，CPU 控制權授與由 Dispatcher 執行
- **CPU Scheduler** 或 **Process Scheduler**
- **特徵**
    - 執行頻率高
        - CPU 可能隨時會 Idle，Short-Term Scheduler 就要挑下一個工作
    - 各種系統均需要
    - 不能到次儲存體抓資料
        - 無法調整 Multiprogramming Degree
        - 無法調整 CPU Bound 和 I/O Bound 的混合比例

Medium-Term Scheduler

- **目的：**當記憶體空間不足，又有其他 Process 要進入記憶體
    - 挑選 Process 條件
        - 在記憶體的時間超時 Storage-Time Slice Expires
        - 低優先權 Lower Priority Process
    - 將這些 Process **Swap Out 到硬碟中**

    - 等記憶體空間足夠，在將這些 Process **Swap In 回記憶體中**繼續執行
- **特徵**
    - 執行頻率介於 Long-Term 和 Short-Term 之間
    - 用於 Time-Sharing System (Real Time, Batch 不用)
    - 可調控 Multiprogramming Degree
        - 除了靠 Long-Term Scheduler 之外，可以將之前 Swap Out 的行程在 Swap In 到記憶體
    - 可調整 CPU Bound 和 I/O Bound 的混合比例

Context Switching 內容轉換



- **定義：** 當 CPU 的使用權從一個行程切換給另一個行程
    - 必須**儲存舊行程的相關資訊**(**儲存在主記憶體**)，並且把**新行程的相關資訊載入**系統中(**載入到暫存器**)
- Context Switching 所花費的時間對系統而言是**額外浪費**
    - 這個過程是**不具有生產力的工作**
    - Context Switching 過多，會造成系統效能瓶頸
- Context Switching 速度取決於**硬體**
    - 記憶體速度
    - Register 數量
    - 特殊機器指令

如何降低 Context Switching 的負擔

方法一 提供多套 Register Sets

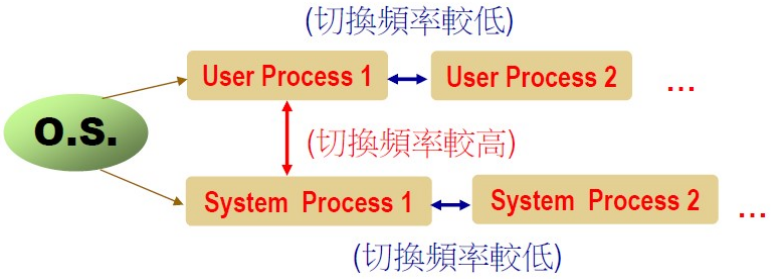
- 當 Register 夠多，**每個行程都有自己的 Register Sets**
    - Context Switching 時 OS **只要切換 Register Sets 的指標**到新的行程

- **不會用到記憶體進行存取動作**
  - 舊行程的 PDB 不用 Swap Out 到記憶體，也不用從記憶體 Swap In 新行程的 PDB
- 優點：速度快
- 缺點：不適用於 Register 數量少的系統

方法二 改用 Thread 替代 Process

- **Thread:** Light-Weighted Process 輕量級行程
- 每個行程都有各自的**私有資訊 (PDB)**，會占用 Register
- Thread 之間可以**共享記憶體空間**，私有資訊不多
  - Code Section, Data Section, Open File...etc
  - Context Switching 不需大量記憶體存取

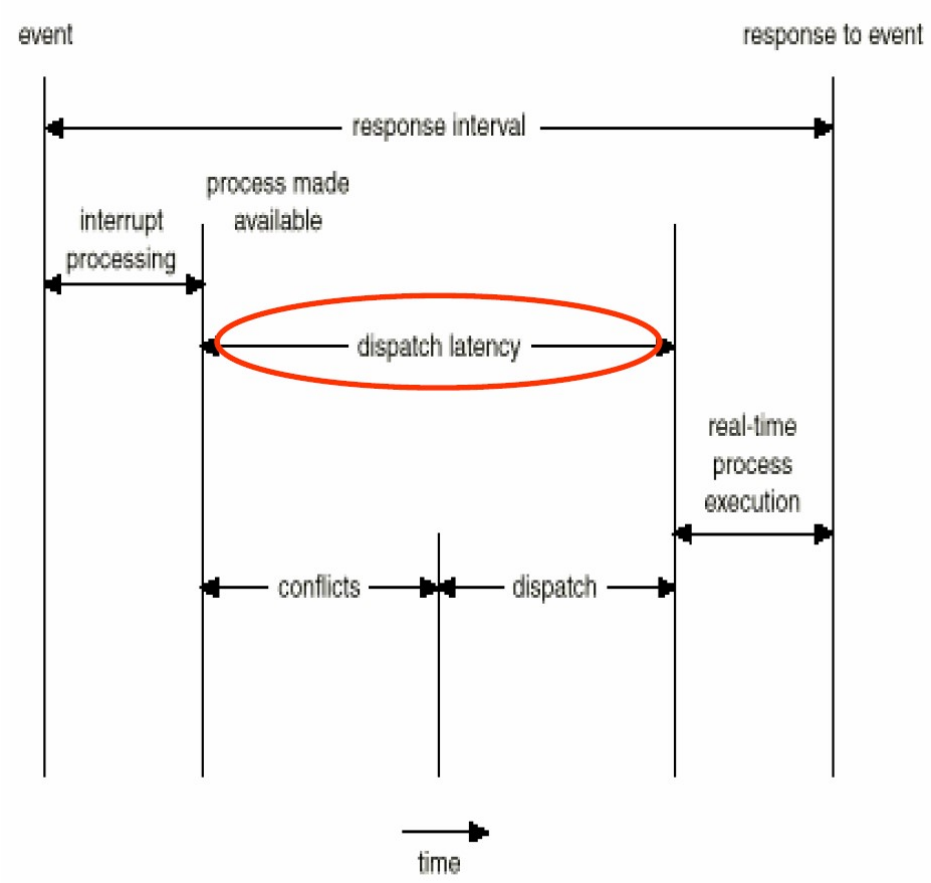
方法三 Register 有限



- 判斷哪一種類的行程**切換較頻繁**
  - User Process 之間切換頻率低
  - System Process 之間切換頻率低
  - User Process 和 System Process 之間的切換頻率高
- **System Process 和 User Process 都有自己的 Register Set**
  - User Process 和 System Process 之間的 Context Switching，只要改變 Register Set 指標即可

Dispatcher

- **負責將 CPU 控制權交給經由 Short-Term Scheduler 挑選出來的行程**
- **工作**
  - **Context Switching**
  - **Change to user mode from monitor mode**
  - **控制權轉移**跳到 User Process 之適當起始位址以便執行
- **Dispatch Latency: 停止一個行程，並開始另一個行程所耗用的時間**
  - 分派潛伏期或**分派延遲**
  - 越短越好，新的行程**開始執行的時間**得以提早



CPU Scheduler

- 一旦 CPU Idle，OS 必須從**主記憶體的 Ready Queue** 挑選行程來執行
- **Short-Term Scheduler** 從記憶體挑選行程，又稱為 **CPU Scheduler**
- 讓系統隨時都有一個行程在執行，提高 CPU 使用率

Scheduling Criteria 排班準則

- **CPU Utilization** 使用率
  - (CPU Use Time) / (CPU Use Time + CPU Idle Time)
    - **CPU Use Time:** CPU 花在行程執行的時間
    - **CPU Idle Time**
      - CPU 閒置
      - CPU 花在非行程的執行時間 e.g. Context Switching
- **Throughput** 產能
  - 單位時間內能完成的工作或行程數
- **Waiting Time** 等待時間
  - **行程在 Ready Queue 等待獲取 CPU 的時間總和**
  - **一個行程真正受到排班法則影響的 Criterion**
- **Turnaround Time** 完成時間 回復時間
  - Task 或行程從進入系統到完成工作的時間
- **Response Time** 反應時間
  - User 下命令到系統產生第一個回應的時間

- 通常在 User Interactive, Time Sharing System 中被要求

排班目標

- **CPU Utilization** ⇑
- **Throughput** ⇑
- **Waiting Time** ⇓
- **Turnaround Time** ⇓
- **Response Time** ⇓
- Resource Utilization⇑
- Fair 行程在 Ready Queue 排隊時不會被插隊
- No Starvation

Preemptive & Non-Preemptive

- **行程在 CPU 執行時，可不可以被趕走**

Non-Preemptive 不可搶先排班

- 當行程取得 CPU，除非這個行程**自願釋放 CPU**，其他行程才有機會取得 CPU
  - **結束工作**
  - **Wait for I/O complete**
- 無法強迫執行中的行程放棄 CPU
- Process STD 中，從 Run State 到 Wait / Terminate 皆是 Non-Preemptive

Preemptive 可搶先排班

- 當一個行程取得 CPU，有可能**被迫放棄 CPU**，將 CPU 交給其他行程執行
  - 高優先權行程進入系統
  - 中斷發生
  - CPU time slice expire(超時)
- Process STD 中，從 Run State 到 Ready 皆是 Preemptive

Preemptive	**Non-Preemptive**
排班效益佳 (Avg. Waiting, Avg. Turnaround Time 較低)	Avg. Waiting Time 較高
Context Switching 頻繁	Context Switching 較少
完成時間不可預期	完成時間可預期
適用於 Real Time 或 Time Sharing System	適用於 Batch System
平均等待時間短	平均等待時間長
不會發生護位效應	會有護衛效應 Convoy Effect

飢餓現象

- **某些行程長期無法取得足夠的 CPU 服務來完成工作，造成自身無窮停滯 (Infinite Blocking)**

- 常發生在不公平環境，如果加上 **Preemptive** 更容易發生
  - CPU Scheduler 公平：對每個行程配置平均
  - CPU Scheduler 不公平：對每個行程配置不平均
- 解決方法
  - 採用 Fair 的 Scheduling Algorithm
  - Aging Techique

Scheduling Algorithm 排班演算法

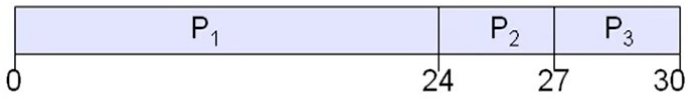
First Come First Served Scheduling

- Arrival Time 越早的行程，越先取得 CPU 控制權
- **特質**
  - **易於製作**
  - **排班效益最差**
    - Avg. Waiting Time, Avg. Turnaround Time 最長
  - **會產生 Convoy Effect**
    - 很多行程**要等一個需要很長 CPU Time 的行程**，造承平居等待時間大幅增加
  - **公平**
  - **無飢餓**
  - **Non-Preemptive**

例題 1

Process	**CPU Burst Time**
P1	24
P2	3
P3	3

- Process 的 Arrival Time 皆為 0
- Process 的到達順序為 P1, P2, P3
- 採用 FCFS Scheduling Algorithm
- 求 Avg. Waiting & Avg. Turnaround Time
- **Solution**

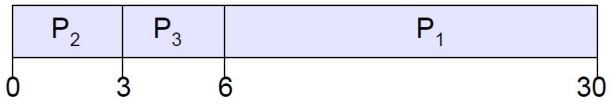


  - 每個行程的 Waiting Time:
    - P1 = 0
    - P2 = 24
    - P3 = 27
  - Average Waiting Time = ((0 - 0) + (24 - 0) + (27 - 0)) / 3 = 17
    - (取得 CPU 的時間 - 到達時間)

- Average Turnaround Time = ((24 - 0) + (27 - 0) + (30 - 0)) / 3 = 27
  - (完成時間 - 到達時間)

例題2

- 假設行程的到達順序為 P2, P3, P1
- **Solution**



  - 每個行程的 Waiting Time:
    - P1 = 0
    - P2 = 3
    - P3 = 6
  - Average Waiting Time = ((0 - 0) + (3 - 0) + (6 - 0)) / 3 = 3
    - (取得 CPU 的時間 - 到達時間)
  - Average Turnaround Time = ((3 - 0) + (6 - 0) + (30 - 0)) / 3 = 13
    - (完成時間 - 到達時間)

Shortest Job First Scheduling

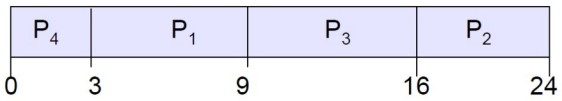
- **定義: CPU Burst Time 越小，越優先取得 CPU 控制權**
- **特質**
  - **排班效益最佳**
    - Average Turnaround Time & Average Waiting Time 最小
    - 不能保證 Response Time，**CPU Bound Job 會被 SJF 放到最後**
  - **不會有 Convoy Effect**
  - **不公平**
  - **可能有飢餓**
  - **Non-Preemptive**
- **不適用 Short-Term Scheduler**
  - 行程的 CPU Burst Time 是預估的
  - Short-Term Scheduler **執行頻率頻繁**，很難在很短的時間間隔內求算每個行程的 CPU Burst Time(沒時間算)
- **Long-Term Scheduler 可採用**
  - **執行頻率較低**，有時間求算 CPU Burst Time 的預估值

例題3

Process	CPU Burst Time
p1	6
P2	8
P3	7

Process	CPU Burst Time
P4	3

- Process 的 Arrival Time 皆為 0
- Process 的到達順序為 P1, P2, P3, P4
- 求 Avg. Waiting & Avg. Turnaround Time
- **Solution**



  - Average Waiting Time = ((0 - 0) + (3 - 0) + (9 - 0) + (16 - 0)) / 4 = 7
  - Average Turnaround Time = ((3 - 0) + (9 - 0) + (16 - 0) + (24 - 0)) / 4 = 13

CPU Burst Time 的預估公式

- 下一個 CPU Burst Time 的預估值 = 前幾次 CPU Burst Time 的指數平均值

  $\tau_{n+1}=\alpha\times t_n+(1-\alpha)\times\tau_n$
- τ_n: 上一次預估的 CPU Burst Time
- t_n: 上一次實際的 CPU Burst Time
- τ_{n+1}: 此次預估的 CPU Burst Time
- α: 加權機率
  - $0\le\alpha\le1$ (Commonly $\alpha=1/2$)
  - 上次預估值很準，α下降
  - 上次預估值不準，α上升

Shortest Remaining Time First Scheduler

- **Preemptive SJF Scheduling**
- **Remaining Time 越小，越先取得 CPU 控制權**
  - 新到達的行程 CPU Burst Time 比執行中的行程的 Remaining Time 小，則執行中的行程被迫放棄 CPU，讓新行程插隊執行
- 特質
  - Average Waiting Time 小於 SJF Scheduling
  - Preemptive
  - Context Switching 負擔較 SJF 大
  - 不公平
  - 可能有飢餓

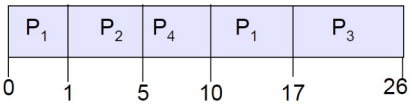
例題4

- 求SRJF Avg. Waiting Time

- 求 SJF Avg. Waiting Time

Process	Arrival Time	Burst Time
P1	0.0	8
P2	1.0	4
P3	2.0	9
P4	3.0	5

- **Solution**



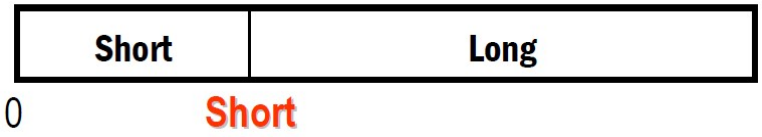
  - SRJF Avg. Waiting Time = ((0 - 0) + (10 - 1) + (1 - 1) + (5 - 3) + (17 - 2)) / 4 = 6.5
  - SJF Avg. Waiting Time = ((0 - 0) + (8 - 1) + (12 - 3) + (17 - 2)) / 4 = 7.75

如何證明SJF 是最理想的

- **有最低的 Avg. Waiting Time**

  原先的 Gantt Chart --> **Avg. Waiting Time = Long / 2**

  SJF 排班的 Gantt Chart --> **Avg. Waiting Time = Short / 2**



- Long-Time Job 所增加的 Waiting Time 遠小於 Short-Time Job，所以若將每一個都依此前移，必使 Avg. Waiting Time 最小

Priority Scheduling

- 優先權高的行程先取得 CPU 控制權
- **特質**
  - **不公平**
  - **可能有飢餓**
  - **Preemptive & Non-Preemptive**
    - **Preemptive**
      - 行程到達 Ready Queue 後，會和正在執行的行程比較優先權值，較高的取得 CPU 控制權
    - **Non-Preemptive**
      - 若新的優先權值較高，不會搶走 CPU 控制權，會放在 Ready Queue 前端

優先權定義

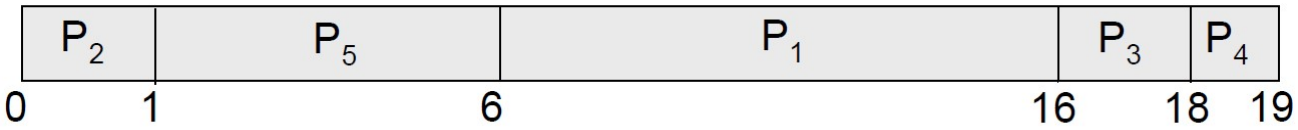
- **內部 & 外部**
  - **內部：** 針對每個行程對 Resource 的需求考量，通常是 OS 掌控
    - **Arrival Time** 越小優先權越高 --> 退化成 FIFO
    - **CPU Burst Time** 越小優先權越高 --> 退化成 SJF
  - **外部：** 針對**政策面**考量，通常是**人**來掌控
    - 行程的重要性
    - 支付的費用
- **Static & Dynamic**
  - **Static:** 行程的優先權設定後，**不能再更改**
    - Soft Real Time System
  - **Dynamic:** 行程的優先權設定後，**可依需求更改**

例題5

- 行程的 Avg. Arrival Time = 0
- 行程的到達順序 P_1 P_2 P_3 P_4 P_5
- 求 Avg. Waiting Time & Avg. Turnaround Time

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- **Solution**



  - Avg. Waiting Time = (1 + 6 + 16 + 18) / 5 = 8.2
  - Avg. Turnaround Time = (1 + 6 + 16 + 18 + 19) / 5 = 12

Round Robin (RR) Scheduling 依序循環排班

- 規定一個 **CPU Time Slice**
  - 行程若未能在 CPU Time Slice(Time Quantum)，會**被迫放棄 CPU** 到 Ready Queue，等待下一輪再使用 CPU
  - 通常用於 Time-Sharing System，**OS 可以掌控 Response Time**
- 需要硬體支援 **Timer**
  - 當行程取得 CPU 後，Timer 的初值會設成 Quantum
  - 隨著行程執行，Timer 遞減
  - 直到 Timer 為 0，OS 會發出 **Time Out 中斷**，強迫行程放棄 CPU

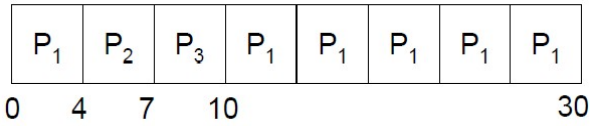
- **特質**
  - 公平
  - 沒有飢餓
  - Preemptive
  - 適用於 Time Sharing System
  - 排班效益取決於 CPU Slice Time
    - q 極大 --> 退化成 FIFO
    - q 極小 --> **Context Switch** 太頻繁，**產能低**
    - 通常 **80%的工作**可以在 Time Slice 內完成，效果效果較好

例題6

- 所有 Processes 的 Arrival Time = 0
- Process 到達的順序 P1, P2, P3
- 採用 RR Scheduling Algorithm, Time Quantum = 4
- 求 Avg. Waiting Time

Process	Burst Time
P1	24
P2	3
P3	3

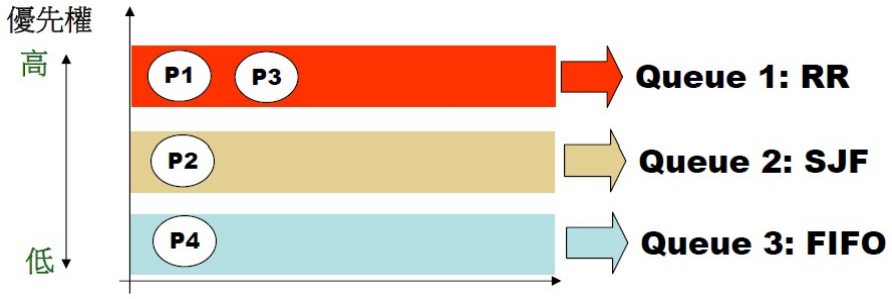
- **Solution**



  - Avg. Waiting Time = ((4 - 0) + (7 - 0) + (10 - 4)) / 3 = 5.67

Multilevel Queue 多層佇列

- 根據**行程不同特性**，將主記憶體單一的 Ready Queue 分成**不同優先等級的 Ready Queue**



- 每個 Queue 都有自己的 Scheduling Algorithm
- Queue 之間採用 **Preemptive Priority** 排班
- 不允許行程進行 Queue 之間的移動
- **特質**
  - 排班設計/調整的彈性高
    - 可參數化 Queue 數目、Queue 演算法、Queue 之間的演算法、行程進入 Queue 的標準

    - 不公平
    - 會有飢餓
    - Preemptive

Multilevel Feedback Queue 多層回饋佇列

- 與 Multilevel Queue 類似，差別在於**允許行程在各佇列之間移動**，避免**飢餓**
- **做法**
    - 採用類似 Aging 技術，每隔一段時間就將行程提升到上一層 Queue
    - 採用**降級**，當上層 Queue 中的行程取得 CPU 後，若未能在 Quantum 內完成工作，則此行程再放棄 CPU 後，會被放到較下層的 Queue 中
- **特質**
    - 排班設計/調整的彈性高
    - 不公平
    - 無飢餓
    - Preemptive

排班演算法統整

- **Preemptive**
    - SRTF
    - Preemptive Priority
    - RR
    - Multilevel Queue
    - Multilevel Feedback Queue
- **Non-Preemptive**
    - FCFS
    - SJF
    - Non-Preemptive
- **No Starvation**
    - FIFO
    - RR
    - Multilevel Feedbacck Queue
- **Fair**
    - FIFO
    - RR