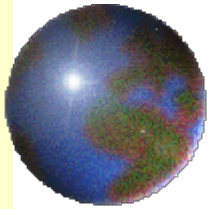
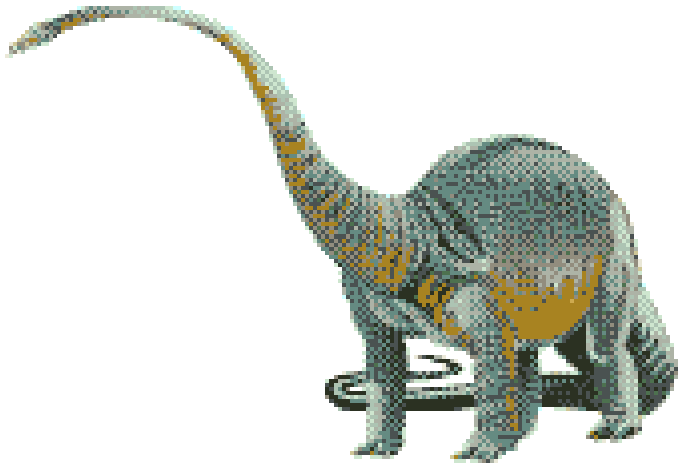


作業系統(Operating Systems)

Course 4: Process (行程)



授課教師:陳士杰

國立聯合大學　資訊管理學系

■本章重點

- Process定義、與Program的不同
- PCB內容
- 行程狀態圖 (Process STD)
- Scheduler種類
  - Long Term, Short Term, Medium Term
- 內容轉換 (Context Switching)
- 分派程式 (Dispatcher)
- Scheduling效能評估之5個Criteria (觀點)
- Scheduling Algo. (7種)及其相關計算
  - Preemption, Non-Preemption
  - Starvation, Aging
  - Convoy Effect (護衛效應)

Process(行程)

- **Def: 正在執行中的程式 (A program in execution)。**
- 一個Process主要包含有：
  - **Code Section (程式碼、程式區間)**
  - **Data Section (資料區間)**

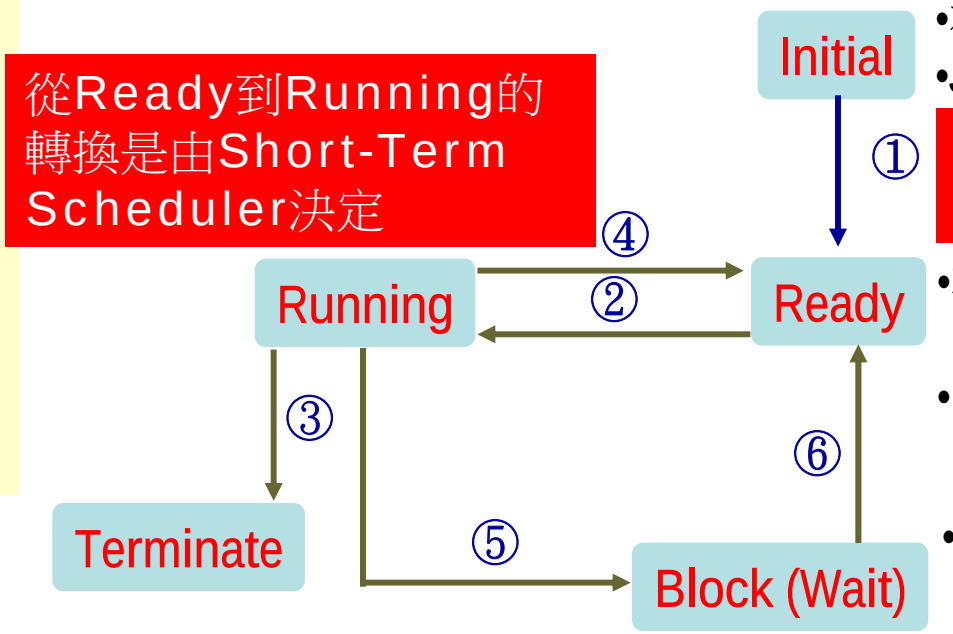
  } 存放於Memory Space
  - **Program Counter (程式計數器)**
  - **CPU Register**
    - 如: 通用暫存器、基底(限制)暫存器...。
  - **Stack**
    - ∵多個Process之間會**相互Call來Call去**及**從事遞迴工作**，用以**存放返回位址**。
- **Process為OS分配資源的對象單位**
- 程式未執行時，只是一個存放在電腦硬碟中的**檔案 (File)**。

Process與Program的不同

Process	Program
主動 (Active Entity)	被動 (Passive Entity)
執行中的程式 (帶有Program Counter，以指出下一個指令所在)	儲存於次儲存體 (e.g., Disk)中的檔案

■行程狀態圖 (Process State Transition Diagram)

- **Process 在執行時會改變其狀態。而Process STD則是用以描述Process由開始到結束的生命週期 (Life-Cycle)，而一個Process在此週期中會經歷數種狀態。**



從Ready到Running的轉換是由Short-Term Scheduler決定

從Initial到Ready是由Long-Term Scheduler決定

- 在Initial時，稱Program為Job (J_1, J_2, J_3)
- Job Queue

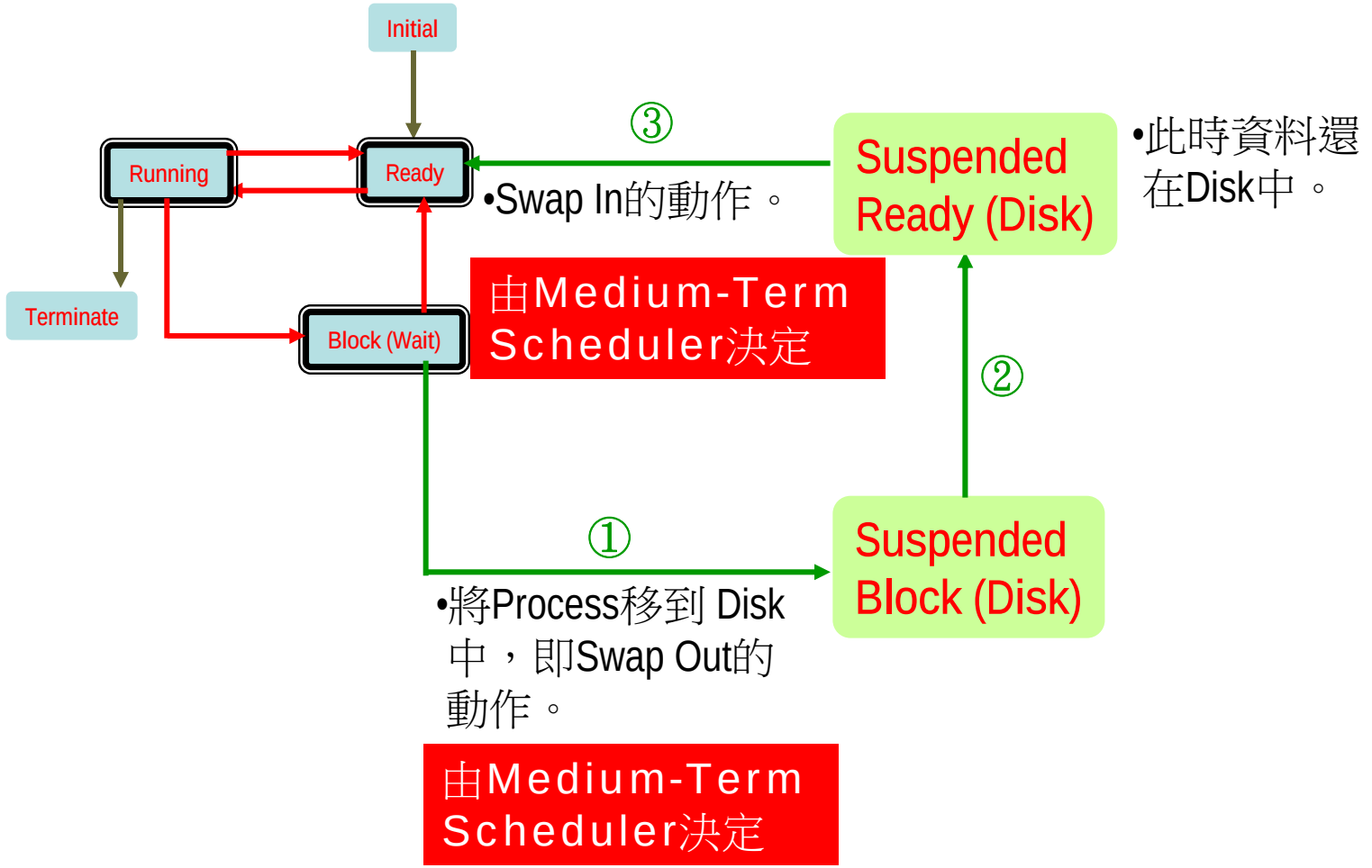
- 在Ready時，稱Program為Process (P_1, P_2, P_3)
- Ready Queue

- 此時不會和其它Process搶CPU或其它資源。
- Process此時還在Mem.中，但不在 Ready Queue，所以不是O.S.配置資源的對象，即Sleep in Mem.。

- 觸發此狀態圖每一個狀態轉換之行為如下:
  - ① 要引入(或產生)一個新的Program到電腦去執行
  - ② 要從記憶體中挑選出一個Process到CPU去執行
  - ③ 一個Process做完其工作時，就正常結束;或當一個Process發生不正常結果時，就中止。
    - 除零
    - 溢位
  - ④ 發生短暫中止時，會直接回到Ready狀態
    - 被高優先權Process插隊
    - 中斷發生
    - CPU Time Quantum 超過
  - ⑤ 發生較長時間中止時，會將Process Block住
    - Wait for I/O complete
    - Wait for resource available
- 此STD是針對CPU這項資源，且Data都在Mem.中。

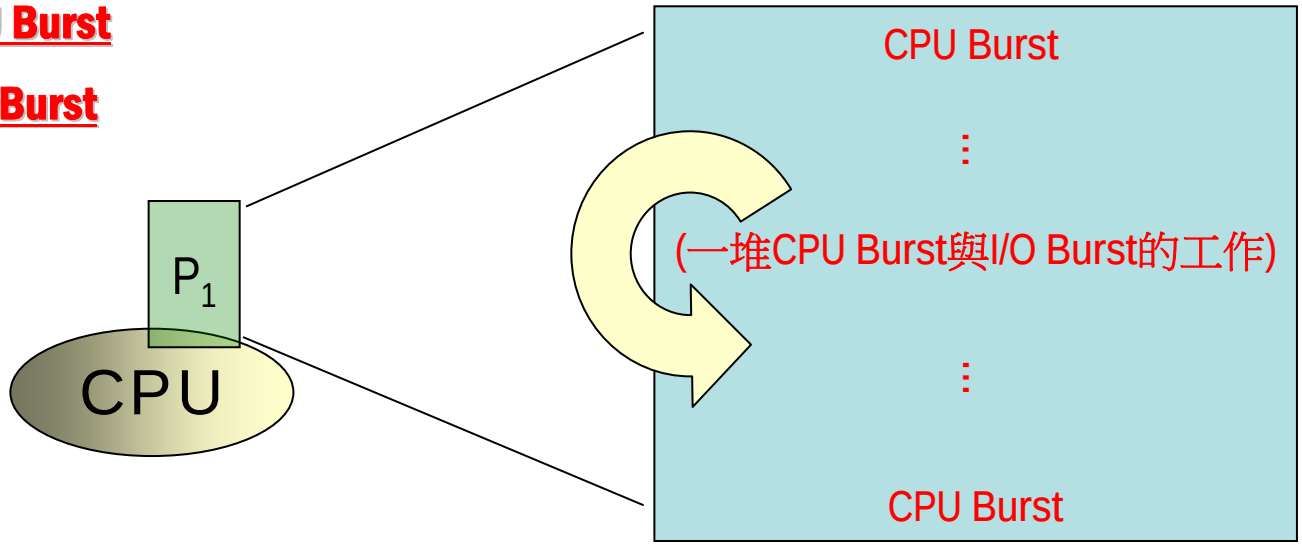
Process Scheduling Queues

- 在Process STD中的Queue：
  - **Job queue**
    - 由一群位於次儲存體 (如：硬碟) 中，等待進入主記憶體之Programs (或稱Jobs) 所形成的集合。
  - **Ready queue**
    - 由一群位於記憶體中，就緒並等待執行的Processes 所形成的集合。
    - 此佇列一般都是用鏈結串列 (Link List) 的方式儲存。
  - **Device queue**
    - 由一群正在等待I/O裝置的Processes所形成的集合。
    - 每個裝置本身都有其Device Queue，以記錄不同Process的請求。
- Processes 一生中可能會在多個不同的Queue中來回遊走。


Initial
Running
Ready
Terminate
Block (Wait)
③
•Swap In的動作。
Suspended Ready (Disk)
•此時資料還在Disk中。
由Medium-Term Scheduler決定
②
①
Suspended Block (Disk)
•將Process移到 Disk中，即Swap Out的動作。
由Medium-Term Scheduler決定

- 觸發此狀態圖每一個狀態轉換之行為如下：
  ① 當Process待在Mem.的時間太長(被Block太久)，或有其它高優先權的Process來搶Mem.這項資源。
  ② 所等待的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中引入Mem.中的Ready Queue。
- 此STD是針對Mem.這項資源，且Data都在Disk中。

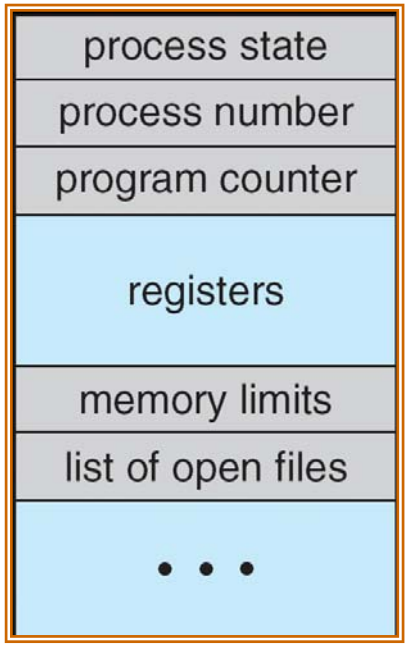
- 一個Process的執行時間是一連串CPU執行時間和I/O等待時間組成。
  - 因為一個Process除了交由CPU執行外，也可能需要由I/O Device進行資料的傳送。
- 幾乎每個在電腦系統中執行的Process都會在兩種工作狀態間切換：
  - CPU Burst
  - I/O Burst



  - Process一開始都是 CPU Burst，也就是交由CPU去處理該Process；接著是I/O Burst，亦即做I/O資料的傳送。在正常的狀況下，Process就在這兩個狀態間一直循環，最後一個 CPU分割會呼叫一個終止行程執行的系統呼叫(system call)來作為行程的結束。

■行程控制表 (Process Control Block; PCB)

- **Def: O.S.** 為執行**Process Management**，所以**將每個Process的所有相關資訊聚集在一起**，建立一個**集合 (Block)**，稱之為**PCB**。(每個**Process**皆有自己的**PCB**)
- **PCB**包含以下資訊：
  - **Process ID**
  - **處理行程狀態: Process**位於**Process STD**的哪一個狀態
  - **程式計數器:** 指明該**Process**下一個要執行的指令位址
  - **CPU暫存器:** 因電腦架構而異 **(e.g.,** 通用暫存器、**PSW...**等**)**
  - **CPU排班資訊 (e.g., Process**優先權值**...**等**)**
  - **記憶體管理資訊: Base/Limit Register**的內容、**Page Table**的相關資訊
  - **帳號資訊:** 用掉多少**CPU**的時間、使用**CPU**的最大時間量**...**
  - **I/O狀態資訊:** 尚未完成的**I/O Request**還有哪些、還在**I/O Queue**中排隊之**Process**的編號**...**



- PCB存在於User Area或Monitor Area?
  - OS為了管理Process方便，會存一份PCB在OS所在之Monitor Area中。

排班程式 (Scheduler) 的種類

- 可分為下列三種:
  - Long-Term Scheduler (或稱 Job Scheduler)
  - Short-Term Scheduler (或稱 Process (CPU) Scheduler)
  - Medium-Term Scheduler

Long-Term Scheduler

- 目的:
  - 從Job Queue中挑選合適的Jobs，並將之載入到Memory內準備執行。
  - 又稱Job Scheduler
- 特徵:
  - 執行頻率最低
  - 通常適用於Batch System，但不適用於Time-Sharing及Real-Time System。
  - 可調控 Multiprogramming Degree (# of processes in memory)，視CPU或Mem.的使用率高低而定。
  - 可調合CPU-Bound與I/O Bound之混合比例 (∵可視資源負荷來決定載入Job與否)。

Short-Term Scheduler

- 目的:
  - 從**Ready Queue**挑選一個已**Ready**且適合的**Process**，使之獲得**CPU**的控制權來執行。
  - 又稱**CPU Scheduler** 或 **Process Scheduler**。
- 特徵:
  - 執行頻率最高

    ∵每個**Process**執行時狀況很多，如：不同情況的**中斷**...等。
  - 各種系統均需要 **(Batch System, Time-Sharing, Real-Time System)**

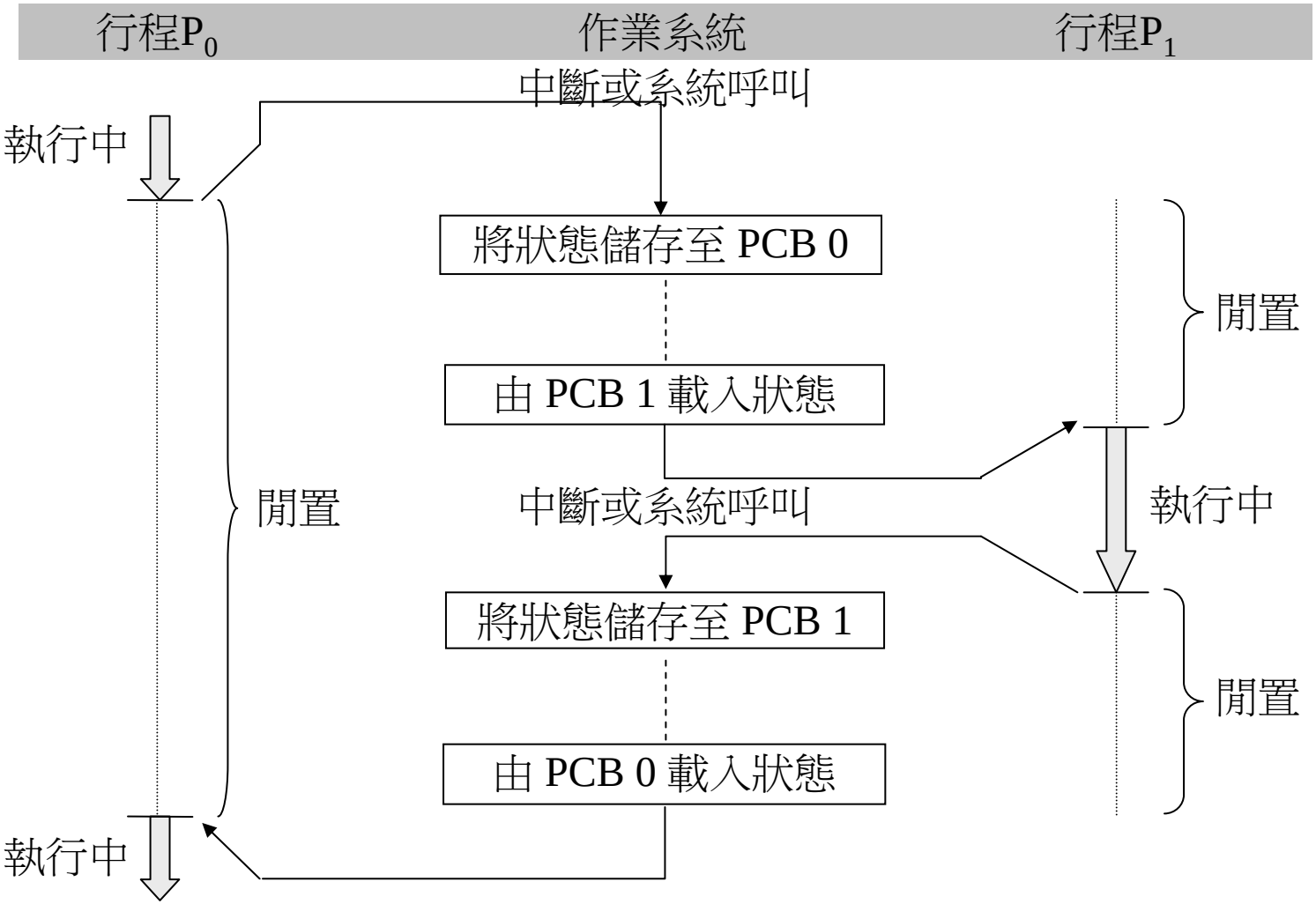
    ∵每種系統都有**CPU**嘛。
  - 無法調整**Multiprogramming Degree** 及 **I/O Bound** 與**CPU Bound Job**之混合比例

Medium-Term Scheduler

- 目的:
  - 通常用在Time Sharing System。當Memory Space不足，且又有其它Process欲進入Mem.執行，此時該Scheduler必須挑選某些Process (e.g., Storage-Time Slice Expires, Lower Priority Process)，將其Swap Out到Disk中，以空出Memory Space，待Mem.有足夠空間時，再將其Swap In回Mem.中繼續執行。
- 特徵:
  - 執行頻率介於Long-Term與Short-Term之間
  - 用於 Time-Sharing System (Real Time, Batch不用)
  - 可調控Multiprogramming Degree
  - 可調合I/O Bound與CPU Bound的比例 (當Long-Term Scheduler有誤判之時!!)

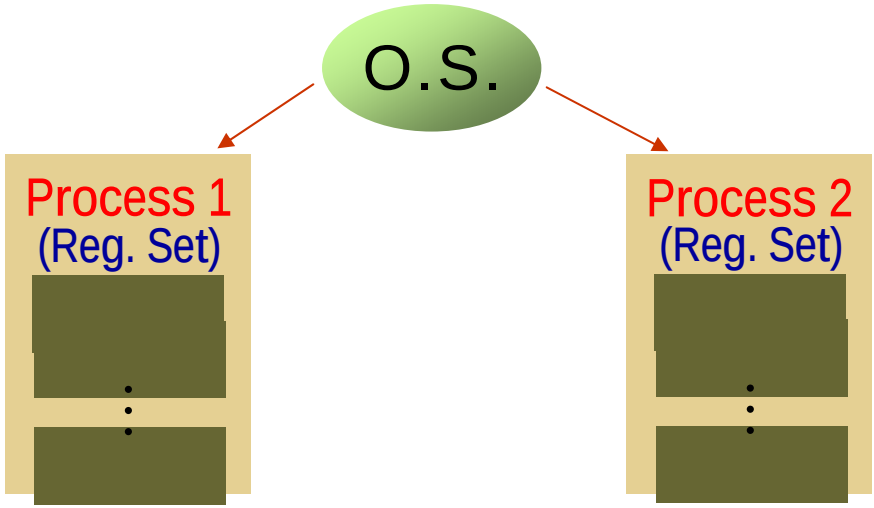
Context Switching (內容轉換)

- **Def:** 當CPU的使用權由一個行程切換給另一個行程時，必須將舊Process的相關資訊 (e.g., PCB內容) 儲存起來，並且把新Process之相關資訊載入到系統中，這個工作就稱做內容轉換(Context switch)。
- Context Switching所花費的時間對系統而言是額外的浪費。
  - 因為在這個過程中，系統所做的事是不具有生產力的工作。
  - 所以，如果 Context Switching 的次數過多，它將會成為系統效能的瓶頸。
- 而 Context Switching 的速度取決於硬體支援的程度。
  - 如：Memory 速度, Register數量, 特殊機器指令存在與否。


行程P_0
作業系統
行程P_1
中斷或系統呼叫
執行中
將狀態儲存至 PCB 0
閒置
由 PCB 1 載入狀態
中斷或系統呼叫
執行中
閒置
將狀態儲存至 PCB 1
閒置
由 PCB 0 載入狀態
執行中

如何降低Context Switching的負擔?

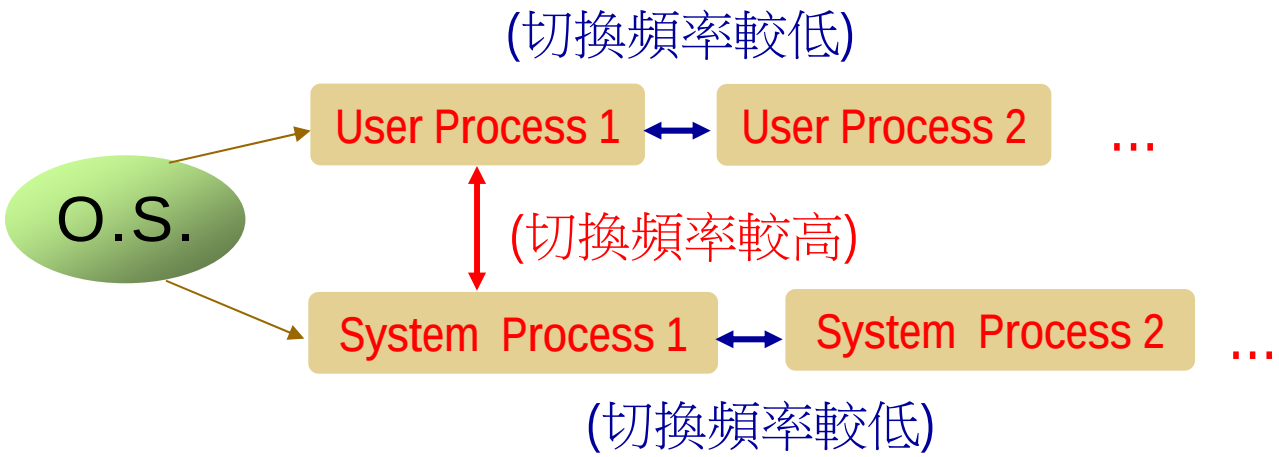
- **(方法一) 提供多套Register Sets**
  - 若Register數量夠多，則**每個Process皆可有自已的Register Set**，所以當需要做Context Switching時：
    - O.S.只要**切換Register Set**的指標到新Process即可。



    - 舊Process的PDB不用Swap Out到Mem.，也不用從Mem.中Swap In新Process的PDB。所以**不會用到Memory進行存取動作**。
    - 優點：**速度快**；缺點：不適用於**Register數量少的系統**。

- **(方法二) 改用Thread代替Process**
  - 使用**Thread (Light-weighted Process)**以降低**Context Switching**負擔。
    - 每個**Process**都有其**私有的資訊(PDB)**，這些私有資訊會佔用**Register**。
    - 然而**Threads**之間彼此可以**共享Memory Space** (如：**Code Section, Data Section, Open File...)**，私有的資訊不多。所以從事**Context Switching**時**不須大量的Memory Access**，可降低**Context Switch**負擔。

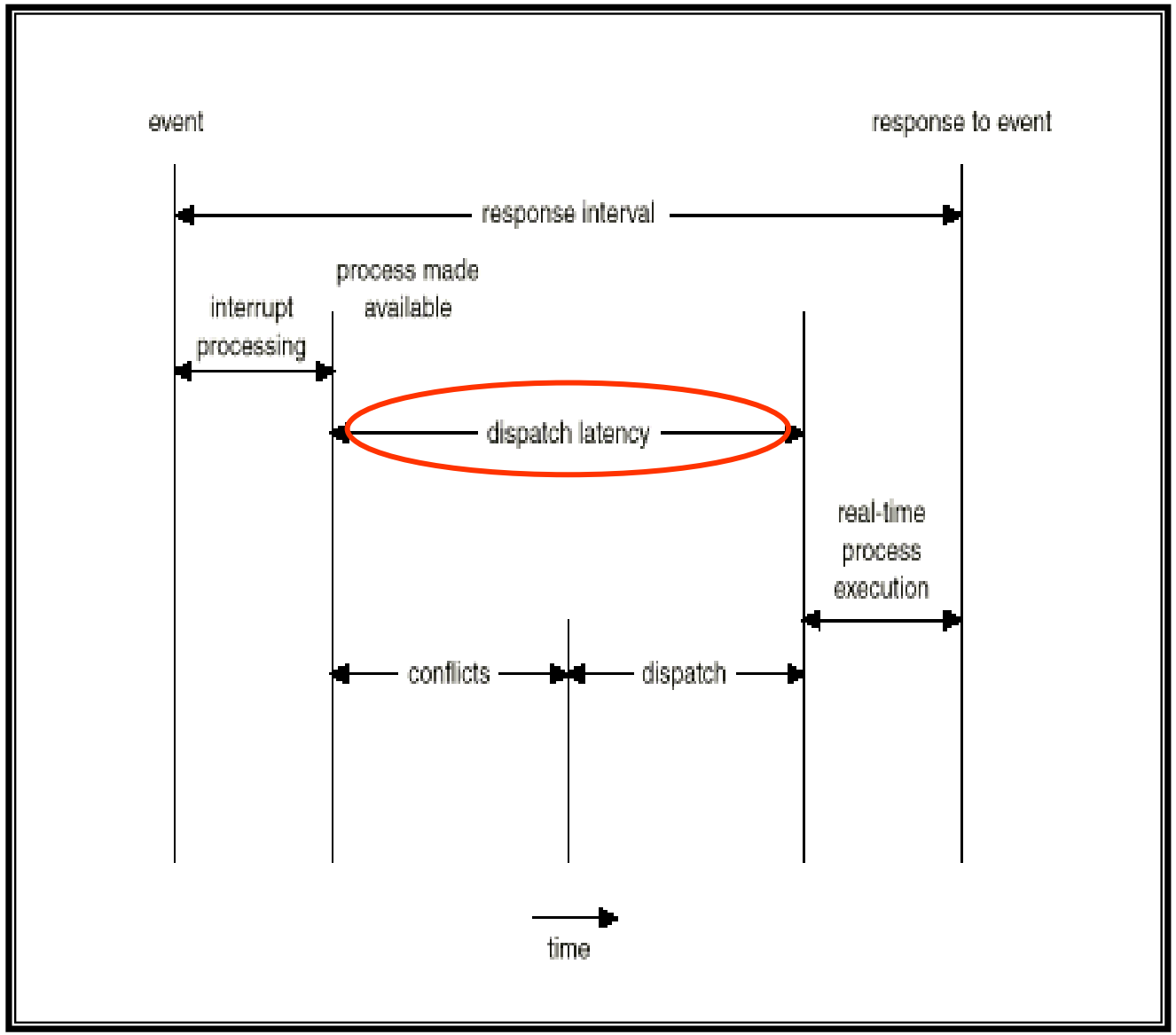
- **(方法三) Register有限時**
  - 當Register有限時，視哪一種類的Process**切換較頻繁**。
  - **System Processes與User Processes都有自已的Register Set**。∴當User Process與System Process之間的Context Switching時，O.S.只要改變Register Set的指標即可。



  - Register不多，但也不會太少!!在乎於調適其用途。

■分派程式 (Dispatcher)

- **Def:** 負責將**CPU控制權**交給經由**Short-Term Scheduler所挑選出的Process**之功能模組。
- 主要的工作有三:
  - **Context Switching**
  - **Change to user mode from monitor mode.**
  - 跳到**User Process**之適當起始位置 **(Starting Address)** 以便執行 **(**為"控制權轉移"的動作**)**
- **Dispatcher**用來**停止一個Process，並開始另一個Process所耗用的時間**，就是**Dispatch Latency** **(**即:上述工作的時間總合，又稱為分派潛伏期或分派延遲**)**。
  - **Dispatch Latency Time愈短愈好**，因為**Dispatch Latency Time**愈短，可使得新**Process** "**可以開始執行的時間**" 得以提早。


event
response to event
response interval
process made available
interrupt processing
dispatch latency
real-time process execution
conflicts
dispatch
time

CPU Scheduler (CPU排班程式)

- 由行程狀態圖可以看出，一旦CPU閒置，O.S.必須從位於主記憶體的Ready Queue之中選出其中一個Process來執行。
- 從主記憶體挑選Process的工作，是由Short-Term scheduler (或CPU Scheduler) 來執行。
  - O.S.利用CPU Scheduler，從存放於記憶體中的數個準備執行之Processes 中挑出一個，透過Dispatcher將CPU配置給它。
- 因此，CPU排班最主要的目的就是讓電腦系統隨時都能夠保有一個Process在系統內執行，藉以提高CPU的利用率。

Scheduling Criteria (衡量排班效能的準則)

- **CPU Utilization (CPU使用率)**
  - **Def:** (CPU Use Time)/(CPU Use Time + CPU Idle Time)
    - CPU Use Time : CPU花在Process執行的時間
    - CPU Idle Time: CPU花在非執行工作本身的時間
- **Throughput (產能)**
  - **Def:** 單位時間所能完成的Job (或Process) 數
- **Waiting Time (等待時間)**
  - Process待在Ready Queue等待獲取CPU的時間總和
  - 一個Process真正受到排班法則影響的Criterion
- **Turnaround Time (完成時間、回復時間)**
  - **Def:** 自一個Task (或Process)進入系統，到其完成工作這段時間
- **Response Time (反應時間)**
  - 自User下命令進入系統，到系統產生第一個回應的時間
  - 通常在User Interactive, Time Sharing System中較被要求。

排班目標

- CPU Utilization (CPU使用率) ⇧
- Throughput (產能) ⇧
- Waiting Time (等待時間) ⇩
- Turnaround Time (完成時間) ⇩
- Response Time (反應時間) ⇩
- Resource Utilization (資源使用率) ⇧
- Fair (公平)
- No Starvation (飢餓)

Preemptive vs. Non-preemptive

- CPU Scheduler的所有演算法則大致可以分為兩大類:
  - 不可搶先 (不可插隊) 的排班 (Non-preemptive):
    - **Def:** 當Process取得CPU在執行時，除非這個Process自願將CPU釋放出去 (如:①Process 結束工作; ②Wait for I/O complete)，其它Process才有機會取得CPU，否則其它的Process無法取得CPU的使用權。
    - 其它Process無法強迫該執行中的Process放棄CPU。
    - 即:Process STD中，從Run state到Wait或是Terminate皆為Non-Preemptive
  - 可搶先 (可插隊) 的排班 (Preemptive):
    - **Def:** 當一個Process取得CPU在執行時，有可能被迫放棄CPU (如:①Highest Priority Process進入系統; ②中斷發生; ③CPU time slice expires)，將CPU交給其它的Process執行。
    - 即:Process STD中，從Run state回到Ready state皆為Preemptive

Preemptive	Non-preemptive
• Def:	• Def:
• 一般而言，排班效益佳 (Avg. Waiting 或 Avg. Turnaround Time ↓)	• Avg. Waiting Time ↑ (∵可能有Convoy Effect)
• Context Switching次數較頻繁	• Context Switching次數較少
• Process的完成時間不可預期	• 可預期
• 較適用於Real-Time或Time Sharing System	• Batch System較適用
• 平均等待時間短	• 平均等待時間長
• 不會發生護衛效應	• 會有護衛效應發生

Starvation (飢餓現象)

- **Def:** 某些Processes因**長期無法取得足夠的資源**來完成其工作，造成**自身無窮停滯的情況 (Infinite Blocking)**。
  - 常發生在不公平的環境，若再加上Preemptive則更易發生
    - CPU Scheduler是公平的(Fair): 對每一個Process的配置都很平均
    - CPU Scheduler是不公平的: 對每一個Process的配置不平均
- **解決方式**:
  - 採用**Fair** 的Scheduling Algorithm
  - **Aging Technique** (老化技術)
    - **Def:** 系統每隔一段時間，會將待在系統內時間很長，且未完成工作的Process，**逐步提高其Priority Value**。因此，經過一段有限時間後，其Priority會為最高，進而取得所須資源以完成工作。
    - **Soft Real Time System不會採用**。

Scheduling Algorithms (排班演算法)

- First Come First Served (FCFS) Scheduling (先到先做排程)
- Shortest Job First (SJF) scheduling (最短工作優先排程)
- Shortest Remaining Time First (SRTF) Scheduling (剩餘時間最短優先排程)
- Priority scheduling (優先權排程)
- Round-Robin (RR) scheduling (循環分時排程)
- Multilevel Queue scheduling (多層佇列排程)
- Multilevel Feedback Queue scheduling (多層回饋佇列排程)

First Come First Served Scheduling (FCFS)

- **Def:** **Arrival Time (到達時間) 愈早 (小)** 的Process，愈優先取得CPU控制權。
- 特質：
  - **簡單，易於製作**
  - **排班效益最差** (∵Avg. Waiting Time 或 Avg. Turnaround Time最長)
  - **會產生Convoy Effect (護衛效應、護送現象)**
    - Def: 很多Processes均在**等待一個需要很長CPU Time來完成工作的Process**，造成**平均等待時間 (Avg. Waiting Time) 大幅增加**的不良現象。
  - **Fair (公平)**
  - **No Starvation (飢餓)**
  - **屬於Non-preemptive (不可插隊、不可搶先)**

計算題

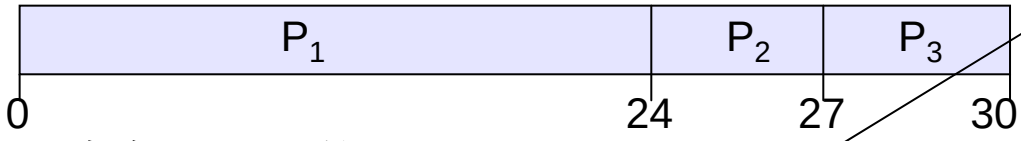
- 畫Gantt Chart
- 求平均Waiting或Turnaround Time

例:給予

Process	CPU Burst Time
P_1	24
P_2	3
P_3	3

- 上述這些Processes的 Arrival Time皆為 0
- Process到達的順序: P1, P2, P3
- 採 FCFS (FIFO) Scheduling Algorithm
- 求 Avg. Waiting Time及Avg. Turnaround Time

Sol:

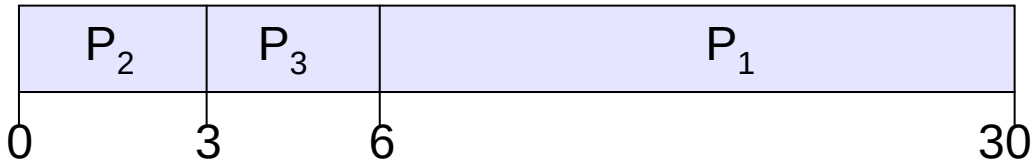


拿到CPU時間-到達時間

- 每個Process的Waiting Time: P1 = 0; P2 = 24; P3 = 27
- Average waiting time: ((0-0) + (24-0) + (27-0))/3 = 17
- Average turnaround time: ((24-0) + (27-0) + (30-0))/3 = 27

完成時間-到達時間

- 假設 processes 到達的順序為 P2, P3, P1.
- 甘特圖如下所示:



  - 每個Process的Waiting Time： P1 = 6; P2 = 0; P3 = 3
  - Average waiting time: ((6-0) + (0-0) + (3-0))/ 3 = 3
  - Average turnaround time: ((30-0)+(3-0) + (6-0))/3 = 13

- 因為**到達順序**不同，而產生**差距頗大的平均等待時間**

Shortest-Job-First (SJF) Scheduling

- **Def:** 若Process的**CPU Burst Time愈少**，**愈優先取得CPU控制權**。
- 特質：
  - **排班效益最佳** (**最理想**)
    - Avg. Turnaround Time與Avg. Waiting Time最小
    - 對Response Time則不保証!!∵若是一個CPU Bound Job會被SJF放到最後面執行。
  - **不會有Convoy Effect** (∵時間花愈長的Process會排在愈後面才進入)
  - **不公平** (∵SJF偏好Short Time Job)
  - **有可能產生Starvation** (for long CPU time job)
  - **Non-Preemptive** (可搶先的SJF被歸成**SRTF Algorithm**)

不適用於Short-Term (CPU) Scheduler

- 在SJF中，每個Process的CPU Burst Time是用預估的方式來求算!!因為要實際得知下一個Process的真正CPU Burst Time是很困難的。
- 然而，此CPU Scheduler (Short-Term Scheduler)執行頻率極高，因為它會動不動就去挑選Job，所以很難在短的時間間隔中去求算每個Process的CPU Burst Time，並挑選出最小值，故不適用於Short-Term Scheduler。

Long-Term Scheduler可採用

- 因為此類型的 Scheduler (Long-Term Scheduler) 執行頻率較低，比較有時間去求算CPU Burst Time的預估值。

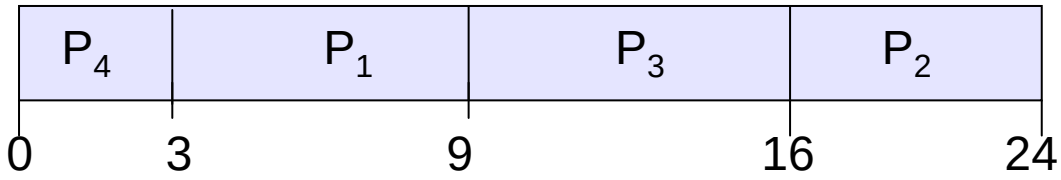
Example of Non-Preemptive SJF (不可搶先的SJF範例)

例 : 給予

Process	CPU Burst Time
P1	6
P2	8
P3	7
P4	3

- 上述這些Processes的 Avg. Arrival Time皆為 0
- Process到達的順序: P1, P2, P3, P4
- 求 Avg. Waiting Time及Avg. Turnaround Time

Sol:



- Average waiting time = ((3-0) + (16-0) + (9-0) + (0-0))/4 = 7
- Average turnaround time = ((9-0) + (24-0) + (16-0) + (3-0))/4 = 13

預測下一個 CPU Burst Time之公式

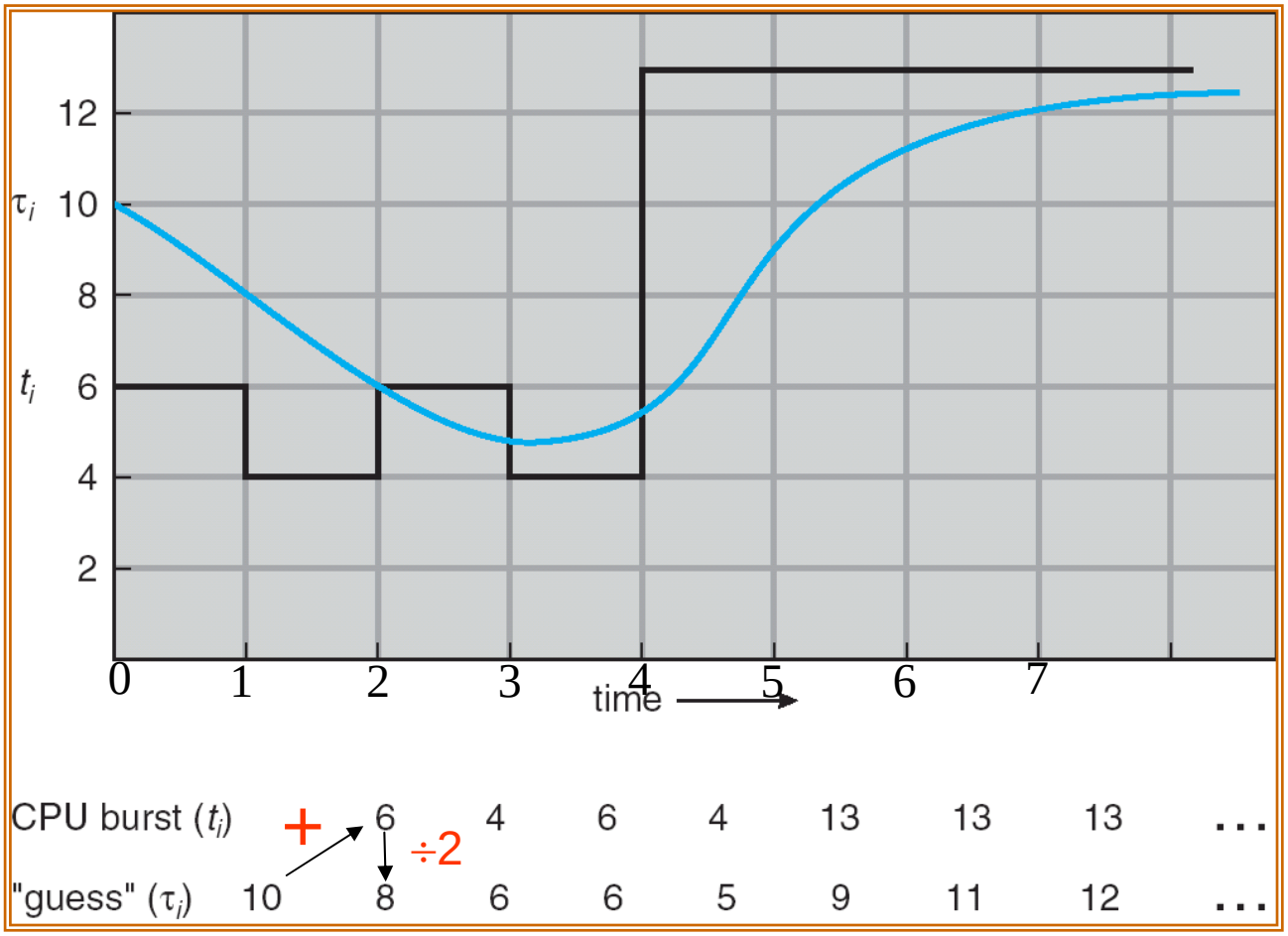
- 下一個 CPU Burst Time的預估值可以設定為**前幾次 CPU Burst Time的指數平均值**。
  - **Def:**

$$\tau_{n+1} = \alpha \times t_n + (1 - \alpha) \times \tau_n$$

  - 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: 參數 (加權機率), $0 \leq \alpha \leq 1$ (More commonly, $\alpha = 1/2$)
    - 若上次預估的時間很準，則α下降
    - 若上次預估的時間不準，則α上升

$\alpha = 1/2$

$\tau_0 = 10$



CPU分割（t_i）

預測（τ_i）

Shortest-Remaining-Time-First (SRTF) Scheduling

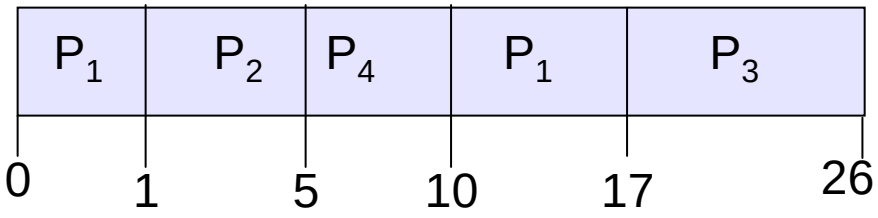
- 為Preemptive SJF Scheduling
- Process的Remaining-Time愈小，可以愈先取得CPU控制權
  - 即：新到達的Process，若其CPU Burst Time小於目前正在執行的Process之Remaining Time，則執行中的Process會被迫放棄CPU，讓新到達的Process插隊執行。
- 特質：
  - Avg. Waiting Time小於SJF Scheduling
  - Preemptive
  - Context Switching次數 (負擔) 較SJF大 (∵插隊次數非常頻繁)
  - 不公平
  - 可能有Starvation

例:給予

Process (行程)	Arrival Time (到達時間)	Burst Time (分割時間)
P1	0.0	8
P2	1.0	4
P3	2.0	9
P4	3.0	5

- 求 SRTF與SJF之Avg. Waiting Time

Sol. 1: Preemptive SJF (SRTF):



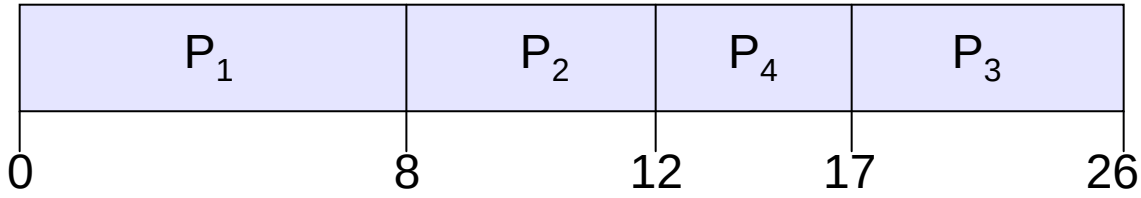
Average waiting time

= [(0-0)+(10-1)+(1-1)+(17-2)+(5-3)]/4

= 26/4

= 6.5

Sol. 2: Non-Preemptive SJF (SJF):



Average waiting time

= [(0-0)+(8-1)+(17-2)+(12-3)]/4

= 31/4

= 7.75

Priority Scheduling (優先權排班)

- **Def:** 具有**較高優先權**的Process愈先取得CPU控制權。
- 特質:
  - 不公平的
  - 可能有Starvation
  - 可以是Preemptive或Non-Preemptive
    - **Preemptive**: 當某個Process到達Ready Queue後，會和目前正在執行的Process比較優先權值。若新的Process之優先權值較高，在Preemptive環境中，**新Process會搶走CPU**來執行。
    - **Non-Preemptive**: 若新的Process之優先權值較高，在Non-Preemptive環境中，新Process不會搶走CPU來執行，它只會將自已**放在Ready Queue的前端**。

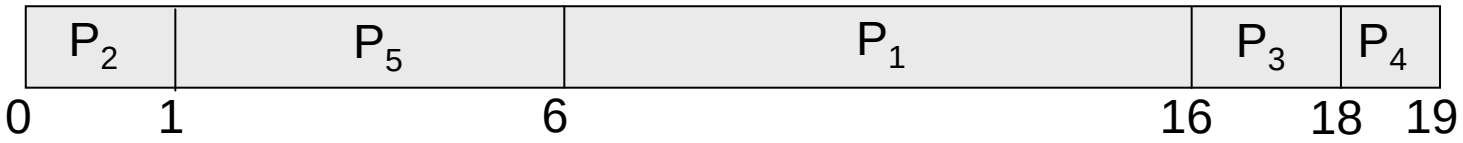
- **Priority Scheduling關鍵在於優先權的定義。**
  - **內部 VS. 外部**
    - **內部**：針對每一個Process**對Resource的需求**為考量，通常是**O.S.** 掌控的。如：對記憶體的需求、時間限制、開啟檔案的數量...等。
      - 假設以**Arrival Time**的大小來決定Process的優先權：若Arrival Time愈小，則優先權愈高 ⇒ 退化成**FIFO** (∵ FIFO ⊆ Priority)
      - 假設以**CPU Burst Time**的大小來決定Process的優先權：若CPU Burst Time愈小，則優先權愈高 ⇒ 退化成**SJF** (∵ SJF ⊆ Priority)
    - **外部**：針對**政策面**的考量，通常是由**人**來掌控。如：Process的重要性、支付的電腦費用...等。
  - **Static VS. Dynamic (指優先權值可否更改)**
    - **Static**：當優先權設定給Process後，就**不能再更改**了。如：Soft Real Time System的Process。
    - **Dynamic**：當優先權設定給Process後，**可依需求再更改**。

例:給予

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

- 上述這些Processes的 Avg. Arrival Time皆為 0
- Process的到達順序: P_1, P_2, P_3 , P_4 , P_5
- 求 Avg. Waiting Time及Avg. Turnaround Time

Sol:



Average Waiting Time：（6+0+16+18+1）/ 5 = 8.2

Average turnaround time：（16+1+18+19+6）/ 5 = 12

Round Robin (RR) Scheduling(依序循環排班)

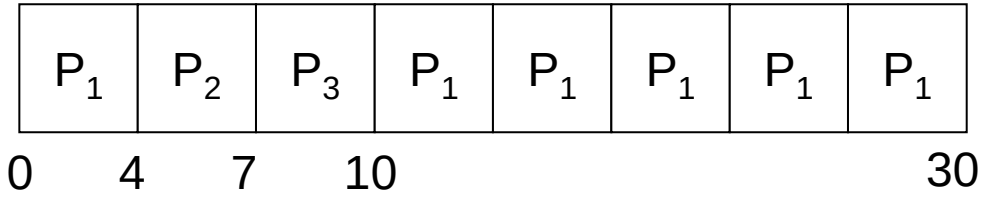
- **Def:** O.S.會規定一個**CPU Time Slice** (時間片段，或稱**Time Quantum**)。當某Process獲取CPU執行，若未能在CPU Time Slice內完成，則此Process會**被迫放棄CPU** (即：該Process的狀態會從**Running →Ready**)，並等待下一輪迴再使用CPU。
  - 通常用在**Time-Sharing System**居多。
- 製作：須要Hardware的支援 – **Timer**
- 作法：當Process取得CPU後，Timer的初值設為Time Slice的值，隨著Process的執行，Timer的值逐次遞減，直到Timer的值為0，會發出 **"Time out" 中斷**通知O.S.，O.S.會強迫目前的Process放棄CPU。

Example of RR with Time Quantum = 4

- 所有Processes的 Arrival Time皆為 0
- Process到達的順序: P1, P2, P3
- 採 RR Scheduling Algorithm
- 求 Avg. Waiting Time

Process	Burst Time
P1	24
P2	3
P3	3

  - The Gantt chart is:



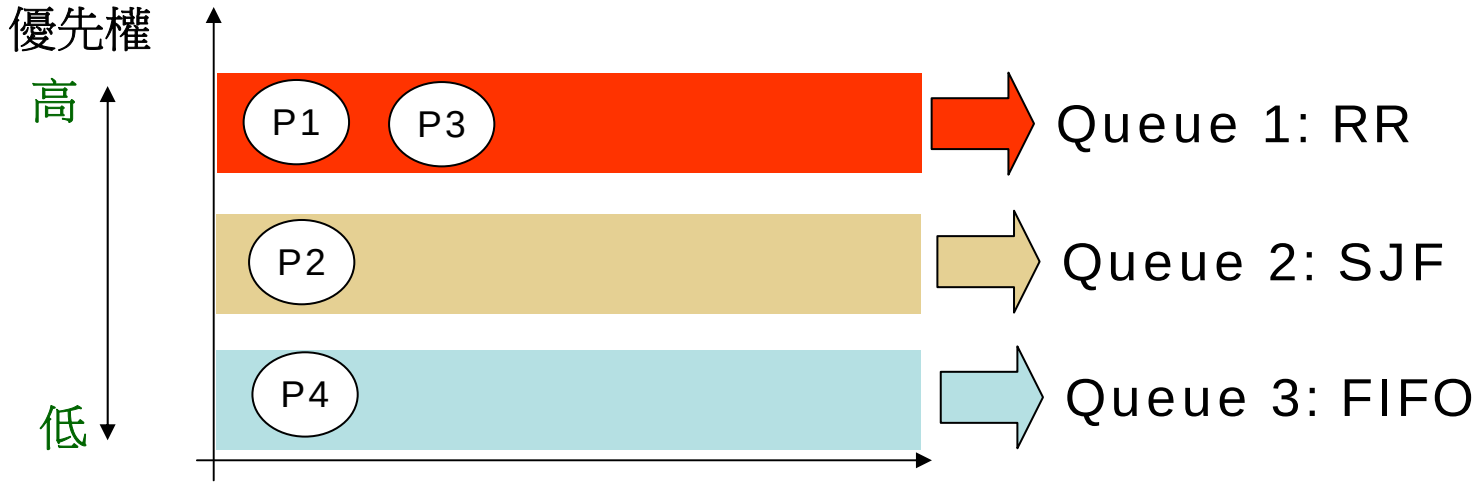
  - Average Waiting Time= [0+(10-4)+(4-0)+(7-0)]/3 = 5.667.

P1 P2 P3

- 特質：
  - 公平的
  - **No Starvation**
  - **Preemptive**
  - 適用於**Time Sharing System**
  - **RR的排班效益(Performance)取決於CPU Time Slice之定義**
    - ***q* large** (≅∞) ⇒退化成 **FIFO**
    - ***q* 極小** ⇒ **Context switch**太頻繁，**CPU Time**未真正用在**Process Execution**上，∴**Throughput (產能) 低**。
    - 通常**80%的工作**可以在**Time Slice**內完成，效果較好。

Multilevel Queue (多層佇列)

- **Def:**
  - 根據**Process的不同特性**，將主記憶體中所形成的單一**Ready Queue**分成**不同優先等級的Ready Queues**。
  - 圖示：



  - 每個**Queue**可有自已的**Scheduling Algorithm**。
  - **Queue**與**Queue**之間採取 "**Preemptive Priority**"之排班方式。
  - 不允許**Process**在各個**Queue**之間移動。

特質

- 排班設計/調整之Flexibility高
  - 可參數化的項目 : Queue的數目、各Queue之法則、Queue之間的法則、Process放入何種Queue的標準。
- 不公平
- 會有Starvation
- Preemptive

Multilevel Feedback Queue (多層回饋佇列)

- **Def:** 與Multilevel Queue的定義相似，差別在於**允許Process在各佇列之間移動**，以避免**Starvation**的情況。
- 作法：
  - 採取類似 "**Aging**"技術，每隔一段時間就將Process往上提升到上一層Queue中。∴在經過有限時間後，在Lower Priority Queue中的Process會被置於Highest Priority Queue中。故無Starvation。
  - 亦可配合**降級**之動作。當上層Queue中的Process取得CPU後，若未能在Quantum內完成工作，則此Process在放棄CPU後，會被置於較下層的Queue中。

特質

- 排班設計/調整之Flexibility高
- 不公平
- **No Starvation**
- **Preemptive**

Scheduling Algorithm Summary

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

公平、可插隊、不會餓死的排班法則是：RR

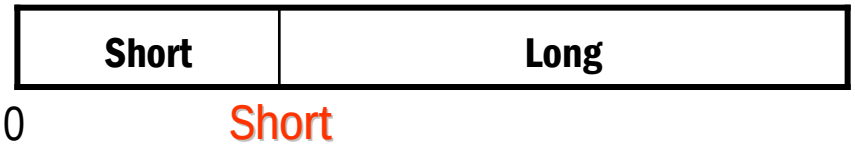
補　充

■如何解釋SJF的排班效益最佳

- 即：**有最低的Avg. Waiting Time** (不論是Non-Preemptive或Preemptive)

說明: 假設原先的Gantt Chart有下列型式存在：

在經過SJF排班後，Gantt Chart變為：



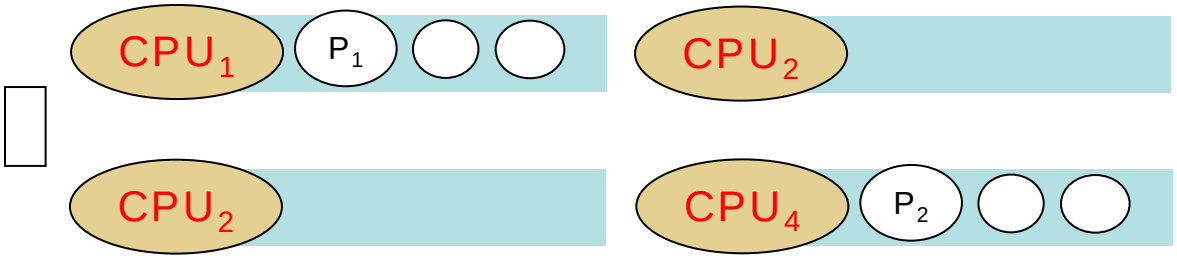
∵改變後所產生的**Long-Time Job之Waiting Time**，**小於等於Short-Time Job原本的Waiting Time** (∵**Long-Time Job**的CPU Burst Time ≥ **Short-Time Job**的CPU Burst Time)。所以，若將每一個Short-Time Job都依此前移，必使Avg. Waiting Time會最小。(以**Waiting Time**的角度來看)

Multiprocessor System的排班考量 (設計原則)

- **Question (Issue):**
  - **Process要分配**到哪一顆CPU上執行？
  - **排班的決定**是由哪一顆CPU負責？

Issue 1: Process 要分配到哪一顆CPU上執行？

- 可能的解法
  - **靜態：**
    - Process被分配到固定的CPU上執行，表示每個CPU有各自的Ready Queue，用自己的排班法則來排班。



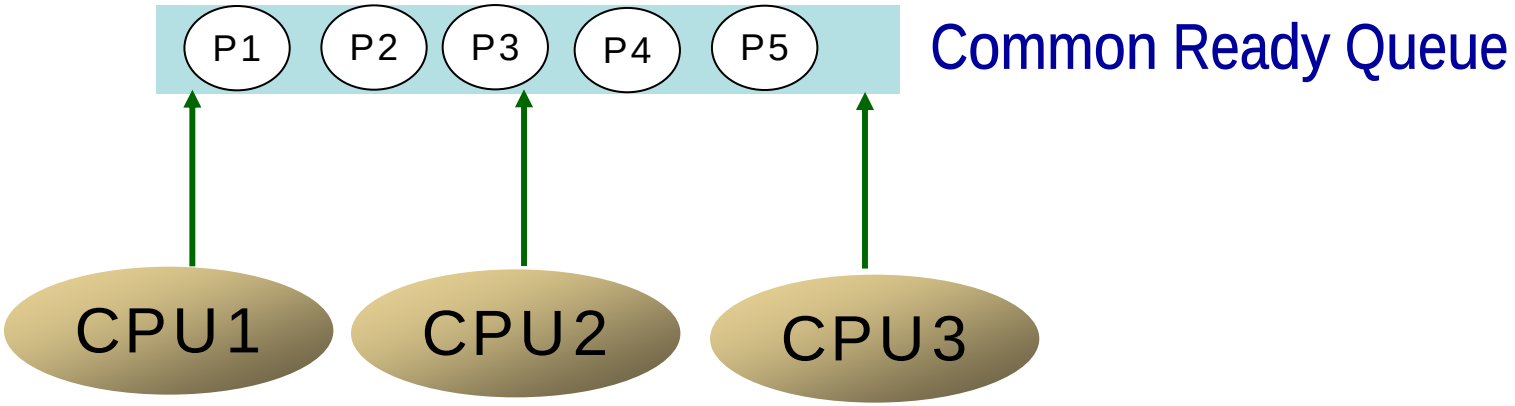
    - 優點：在Uniprocessor上的Scheduling易於被移植到此，製作的Cost低。
    - 缺點：有可能造成某些CPU為Busy，而某些CPU Idle，喪失Multiprocessor Speed up的好處。
  - **動態：**
    - "Load Sharing"。
    - 利用一個Common Ready Queue，所有的Process進入系統時，先放到此Queue中，再依據CPU的Load狀況 (由Master CPU或Peer結構) 分派到某個CPU上執行。

排班的決定是由哪一顆CPU負責？

Issue 2: 排班的決定是由哪一顆CPU負責？

- [法一]: 由Master CPU決定Process Scheduling，assign to which CPU resource conflicts resolution.
- 優點:
  - 易於製作
  - 對於Common Ready Queue等Scheduling Information不須特別提供同步(互斥存取)機制。
- 缺點:
  - Master壞了，System Crash。
  - Master會是效能的瓶頸。

- [法二]:Peer結構



- 必須要對**Common Ready Queue** 提供**互斥存取的同步機制**，確保**不同的CPUs都抓同一個Process執行**或**Process Lost**不會發生。
- 難實作!!

Real Time System的排班考量 (設計原則)

- Real Time System的設計考量可分為 :
  - Hard Real Time System Scheduling Design Principle
  - Soft Real Time System Scheduling Design Principle

Hard Real Time System Scheduling Design Principle

- **排班考量**:在於能否**保証Process在其Deadline之前完成**。若可以，就接受其排班要求;否則，就拒絕。
- **Scheduler必須知道**:
  - 每個**Process**所宣稱的**CPU Computation Time**執行**多少O.S. Service**
  - 每個**O.S. Service花多少時間**
  - 此**Process**的**Deadline**

  ...等資訊。

- **Schedulable (可排程化)的公式(補充)：**
  - **Def:** Process i的週期時間為 T_i，所花的CPU Burst Time為 C_i，$C_i < T_i$。
    若 $\sum_{i=1}^{n} C_i/T_i < 1$，則稱此n個Processes為Schedulable。
- 例：

Process	1	2	3	4
T_i	200ms	80ms	100ms	60ms
C_i	40ms	20ms	20ms	? ms

若可以Schedulable，則P4的CPU Burst Time要小於多少ms?

<sol>: 40/200+20/80+20/100+?/60 < 1

$\Rightarrow$ 0.2+0.25+0.2+?/60 < 1

$\Rightarrow$?/60 < 1-0.65 = 0.35

$\Rightarrow$? < 21.

Soft Real Time System Scheduling Design Principle

- 必須確保以下兩條件之成立:
  - 即時性Process具有最高優先權，且維持不遞減直到完成。
  - 降低O.S.的Dispatch Latency Time。

針對條件 1:

- O.S.須能支援“Priority Scheduling”
- 不提供“Aging”技術

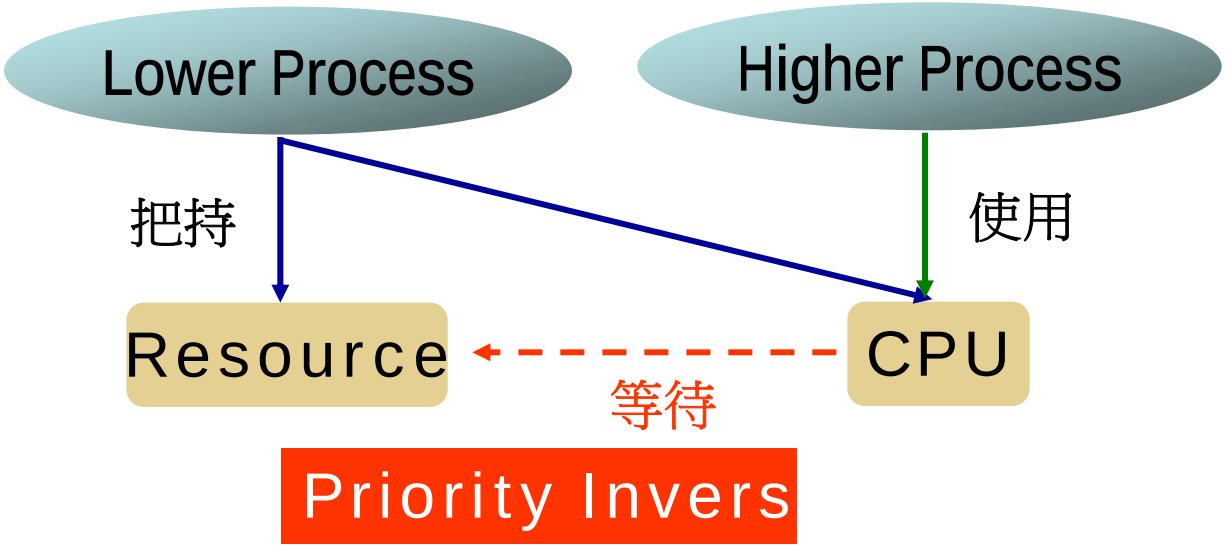
針對條件 2:

- 要去推翻 "**System Call不允許被Preemptive以確保Kernel Data Structure的正確性**"所造成的過長Dispatch Latency Time。
- [法一]:
  - 在System Call (Service, Process)的執行過程中，找出**安全的"Preemption Point"(可搶先點)**。
    - 當System Call執行時，遇到Preemption Point時，則會檢查是否有高優先權的Real Time Process要執行；若有，則允許在此時間點插隊。
  - **缺點**：在System Call中的**Preemption Point很少**，實際的Latency Time還是稍長。

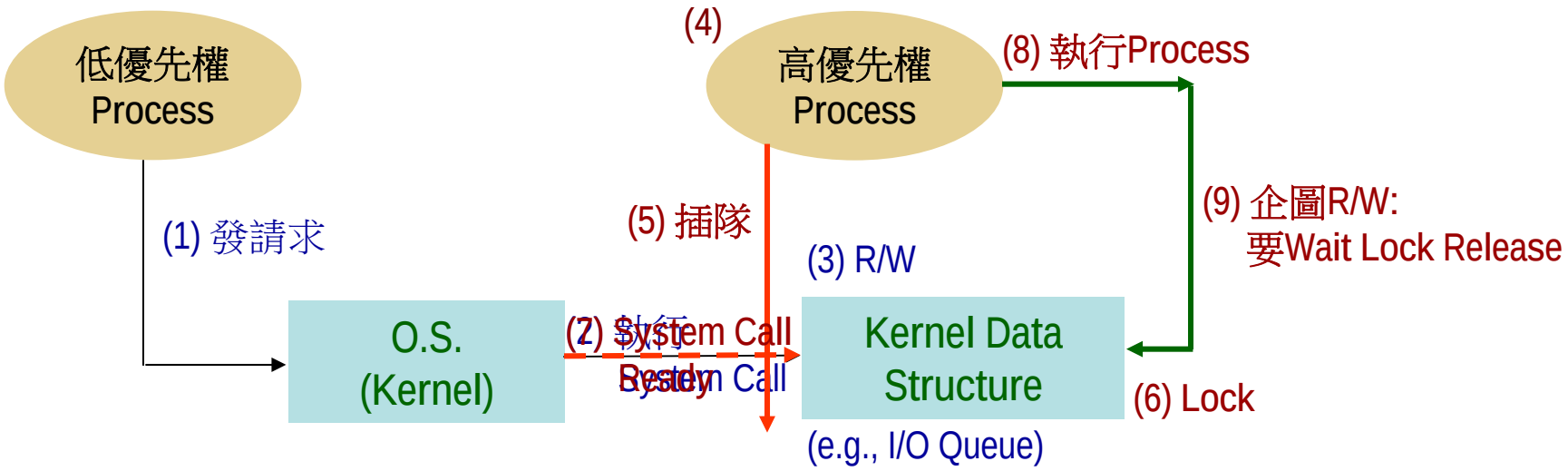
- [法二]:
  - 允許**Kernel (指 : System Process)整個被Preemptive**。會提供**對Kernel Data Structure的同步(互斥控制)**之機制，以防止高優先權的Process去修改Kernel Data Structure，造成資料不正確的情況。
  - 缺點 : 會發生**Priority Inversion**

什麼是Priority Inversion問題

- **Priority inversion occurs when a higher-priority process is blocked by one or more lower-priority processes for a long time.**



- **Def: 高優先權Process企圖Read/Write的Data Structure被某些低優先權的Process所把持(Lock, Protect)，造成高優先權Process須等待低優先權Process釋放Lock的情況稱之。**


低優先權
Process
(4)
高優先權
Process
(8) 執行Process
(1) 發請求
(5) 插隊
(3) R/W
(9) 企圖R/W:
要Wait Lock Release
O.S.
(Kernel)
(2) System Call
Kernel Data
Structure
(6) Lock
(e.g., I/O Queue)

- 解決：**Priority Inheritance (優先權繼承)**
  - 暫時性地調高**Lower Priority Process**的**Priority**到和**Higher Priority Process**一樣，使得這些**Lower Priority Process**可以儘快釋放共用的**Kernel Data Structure**，一旦釋放後，**Higher Priority Process**可以立即執行，且這些**Lower Priority Process**的權值恢復成原先舊值。