Chapter 5:行程排班
Process Scheduling

Chapter 5: Process Scheduling

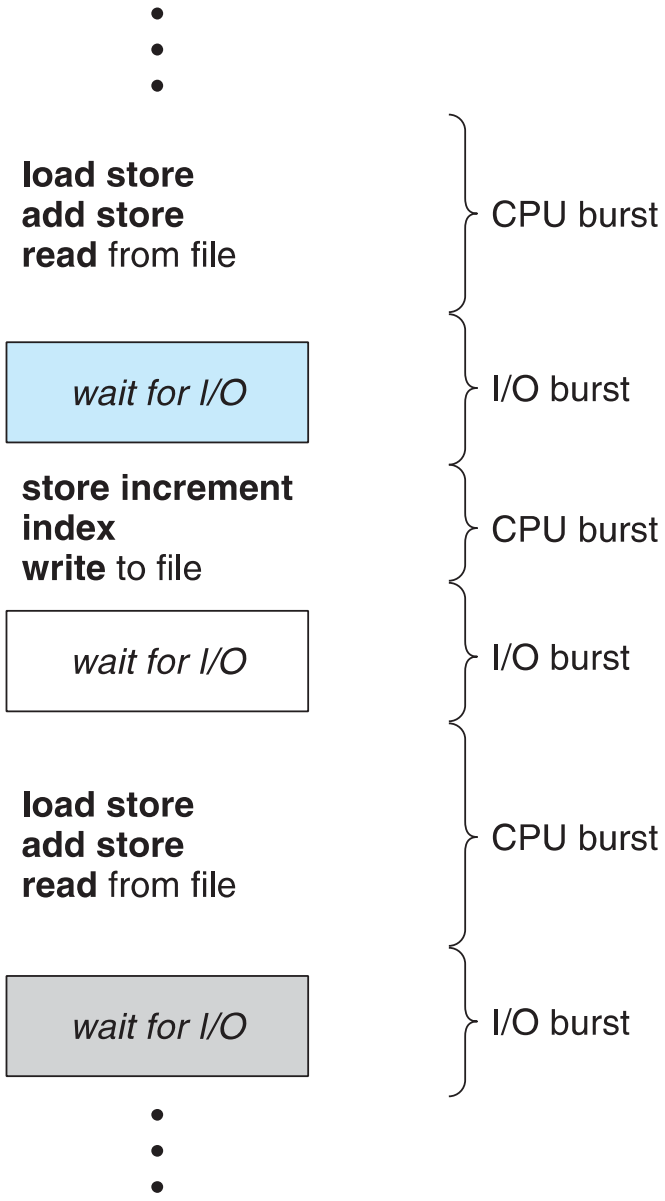
- 基本觀念
- 排班原則
- 排班演算法
- 執行緒排班
- 多處理器的排班問題
- 即時CPU排班
- 作業系統範例
- 演算法的評估

章節目標

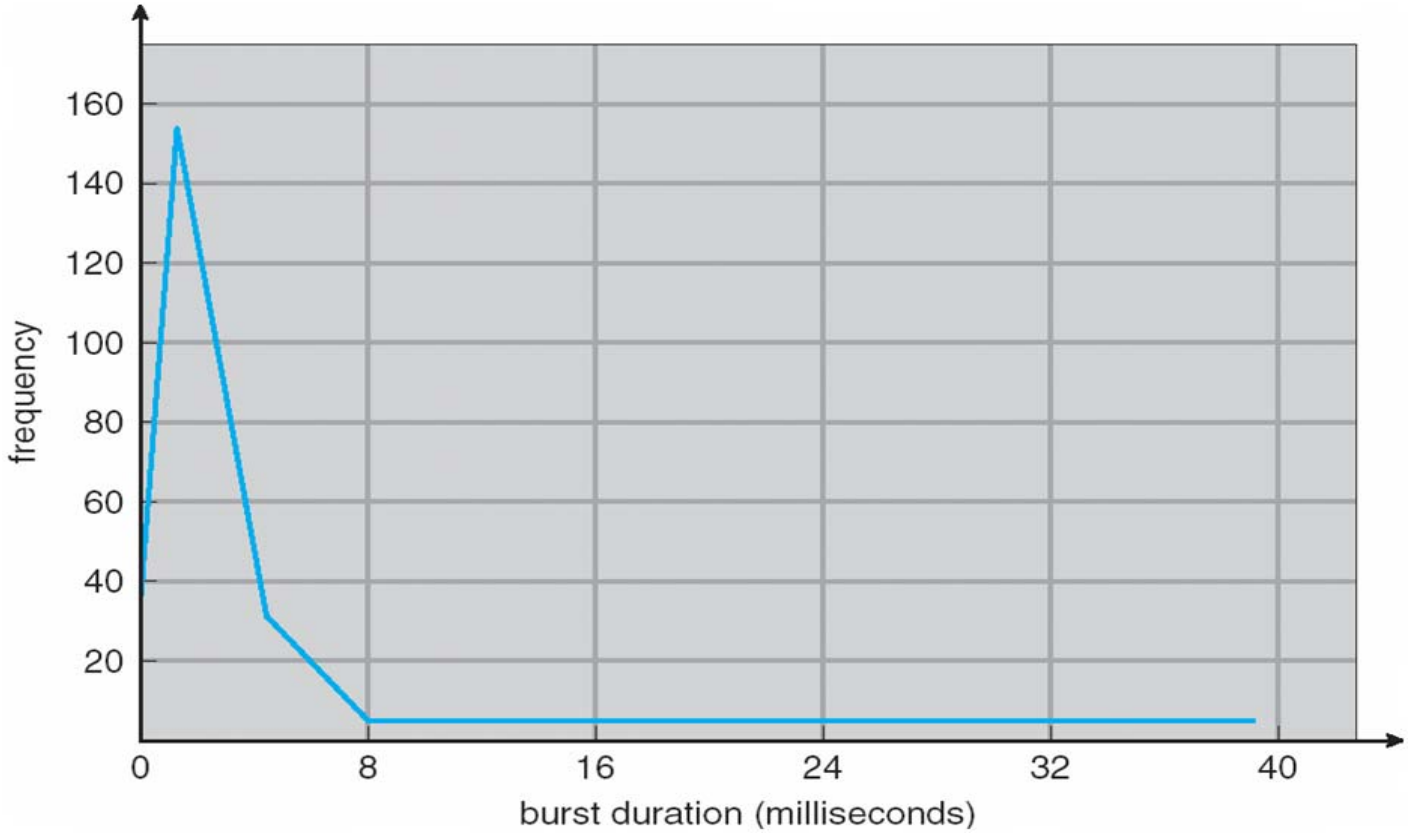
- 介紹CPU排班，CPU排班是多元規劃作業系統的基礎。
- 探討不同的CPU排班演算法。
- 討論為某一個特定系統選擇排班演算法的評估標準。
- 檢視一些作業系統的排班演算法。

基本觀念

- 行程排程是多工系統的基礎
- 排程的概念：當行程在等待某個事件或I/O運算時，因為無法使用到CPU，所以可以將CPU讓出來給其他需要執行的行程使用
- 藉由使用多元程式規畫可以提高CPU使用率
- 行程的執行通常都是在兩種狀況間不斷切換：
  - CPU burst (CPU使用期)：持續地使用CPU
  - I/O burst (I/O使用期)： I/O運算等待
- CPU burst的分布形式，對於如何選擇或設計適當的CPU排程演算法而言，是非常重要的資料



Histogram of CPU-burst Times



CPU排班程式(Scheduler)

- 短程排班程式(Short-term scheduler)從就緒佇列(ready queue)選出一個行程，將CPU配置給它
  - 佇列可以用不同的方式排序
- 當一個行程發生以下四種情狀之一時，即須執行一次CPU的排班決策:
  1. 從執行狀態轉變成等待狀態
  2. 從執行狀態轉變成就緒狀態
  3. 從等待狀態轉變成就緒狀態
  4. 行程終止
- 對情況1及4而言，排班是不可搶先(nonpreemptive)
- 其它情況的排班是可搶先(preemptive)
  - 考慮共用資料
  - 考慮核心模式下的可搶先
  - 考慮在重要OS活動時中斷發生

分派程式(Dispatcher)

- 分派程式將CPU控制權交給短程排班程式所選出的行程
- 分派程式作的事情包括:
  - 內容轉換(context switching)
  - 切換成使用者模式
  - 跳到使用者程式的適當位置繼續執行該程式
- 分派潛伏期(dispatch latency) –分派程式用來停止一個行程，並開始另一個行程所花的時間

排班原則(Scheduling Criteria)

- CPU使用率 (CPU utilization)– 使CPU盡可能地忙碌
- 產量 (Throughput)–單位時間所能完成的行程數目
- 回復時間(Turnaround time) –單一行程執行完畢所需的時間
- 等候時間 (Waiting time) – 一個行程在就緒佇列等待的時間
- 反應時間 (Response time)– 提出一個要求到第一個反應出現的時間(對於分時環境)

排班演算法最佳化的原則

- 最佳的CPU使用率
- 最大產量
- 最短回復時間
- 最短等候時間
- 最短反應時間

先來先做排班演算法（FCFS Scheduling）

- 不可搶先式排程法
- 優點：簡單
- 缺點：等待時間變動很大，而且平均等待時間並不短→可預測性低
- FCFS排程對以CPU為主的程式比較有利
- 可能會發生護送效應(convoy effect)：所有行程都在等待一個長行程離開的情況

先來先做排班演算法(繼續)

行 程	分割時間
P_1	24
P_2	3
P_3	3

- 如果行程到達的順序是P_1 、 P_2、 P_3
  排班的甘特圖:

- 等待時間：$P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- 平均等待時間：$(0 + 24 + 27)/3 = 17$

先來先做排班演算法(繼續)

如果行程到達的順序是 : P_2, P_3, P_1

- 排班的甘特圖:

- 等待時間 : $P_1 = 6; P_2 = 0, P_3 = 3$
- 平均等待時間 : $(6 + 0 + 3)/3 = 3$
- 比前面的情況好很多
- 護送現象(convoy effect) – 短行程在長行程前面
  - 考慮一個CPU傾向和許多I/O傾向的行程

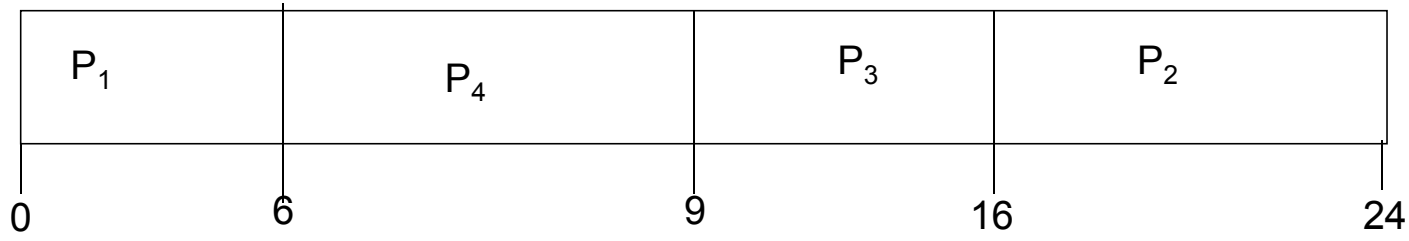
最短工作先做(Shortest-Job-First)排班演算法

- 不可搶先式排程
- 希望選出的是下一次CPU burst最短的行程
- SJF是最佳演算法 –所有不可搶先排班法中平均等待時間最短的一個
  - 將CPU暴衝時間較長的行程延後執行，藉此降低平均等待時間
- 最大問題是如何預先知道每一個行程的下一個CPU burst時間？

Example of SJF

Process	Arrival Time	Burst Time
P1	0.0	6
P2	2.0	8
P3	4.0	7
P4	5.0	3

- SJF scheduling chart



- 平均等待時間 = (0 + 14 + 5 + 1) / 4 = 5

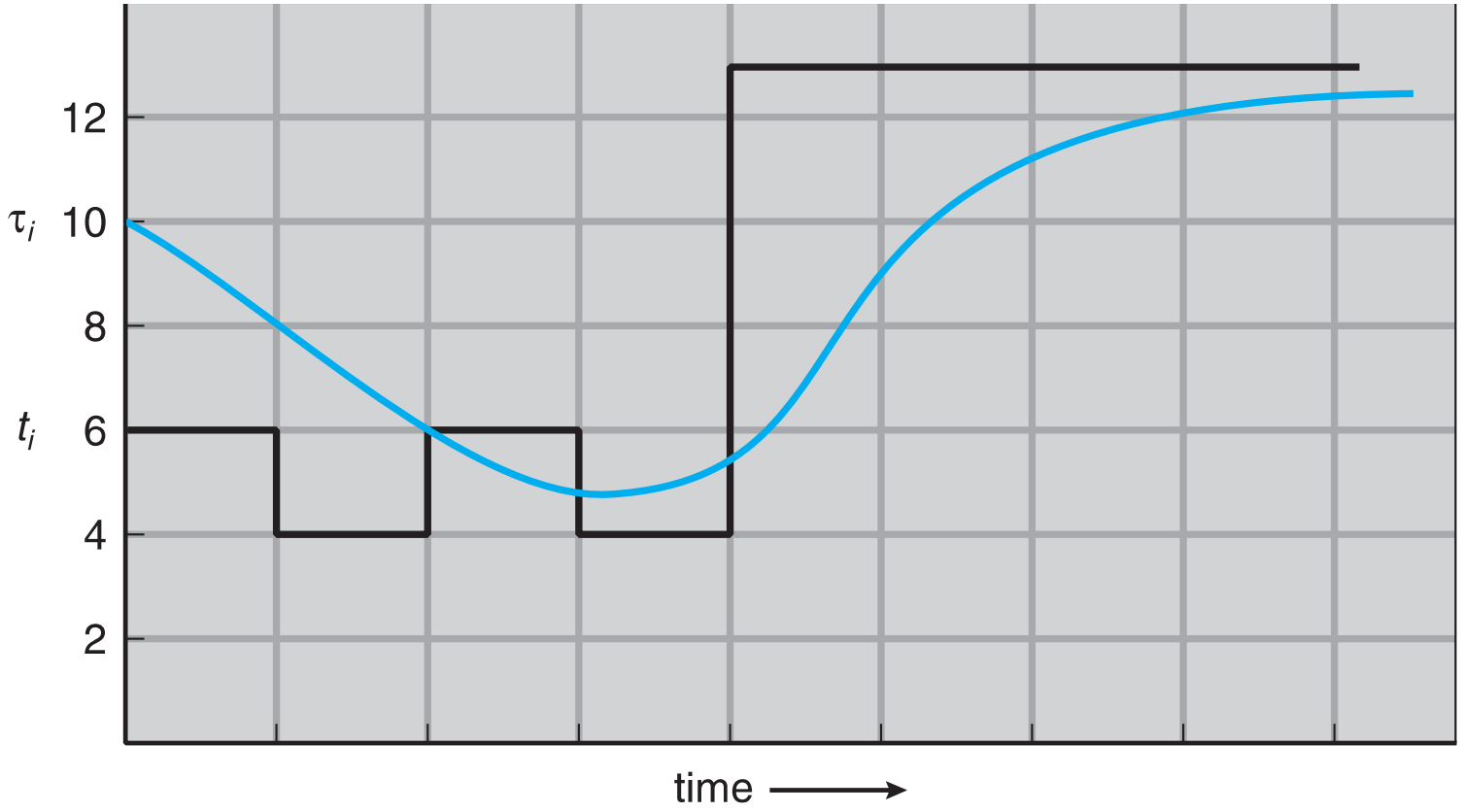
如何決定下一次CPU Burst的長度

- 可利用前一次的長度當作下一次CPU Burst的預估長度
- 根據前幾次CPU分割值的指數平均

1. t_n = 第 n 個 CPU Burst的實際長度
2. τ_{n+1} = 下一個 CPU Burst的預估值
3. $\alpha,\ 0 \le \alpha \le 1$
4. 定義： $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$

- 通常, α 設定成 ½

預估下一個CPU Burst的長度



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

指數平均的範例

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 不考慮最近的記錄
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - 只和上一次的CPU分割有關係
- 如果我們展開公式，會得到:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

- 因為α和$(1 - \alpha)$都小於或等於1，接下來每一項目都比它前一項的比率要低

最短剩餘時間先做 (Shortest-Remaining-Time-First)排班

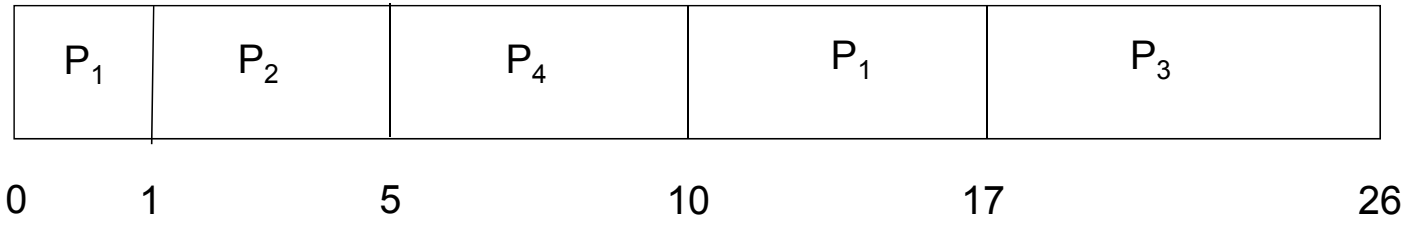
- SJF排班的可搶先版
- 當有新的行程進入就緒佇列時，如果比現在執行中行程所剩下的CPU Burst時間更短，則現在執行的行程會被趕出去，由新行程搶先執行

最短剩餘時間先做排班範例

- 現在我們加入到達時間和可搶先的觀念到分析中

行程	到達時間	分割時間
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- 可搶先 SJF的甘特圖



- 平均等待時間 = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

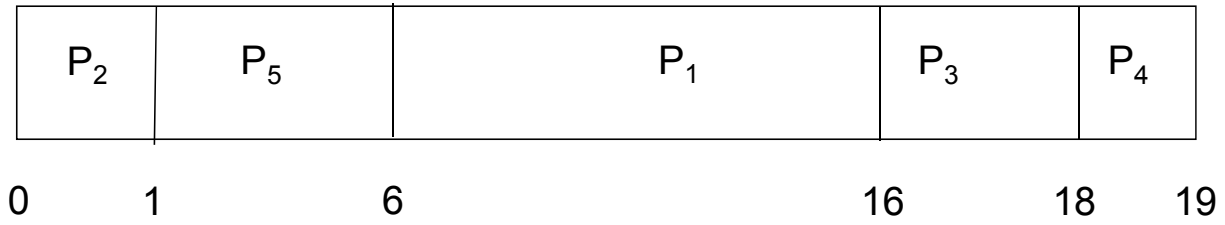
優先權排班(Priority Scheduling)

- 每一個行程都有一個優先權數字 (採取整數)
- CPU優先分配給最高優先權的行程(若最小整數 代表最高優先權)
- 分別可採取可搶先及不可搶先作法
- SJF是優先權排班演算法的一個特例，其優先權就是預估下一次CPU Burst的倒數
- 可能造成飢餓問題(Starvation) –低優先權行程可能永遠無法執行
- 可利用老化機制(Aging)解決飢餓問題–隨著時間逐漸提高行程的優先權

優先權排班範例

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

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

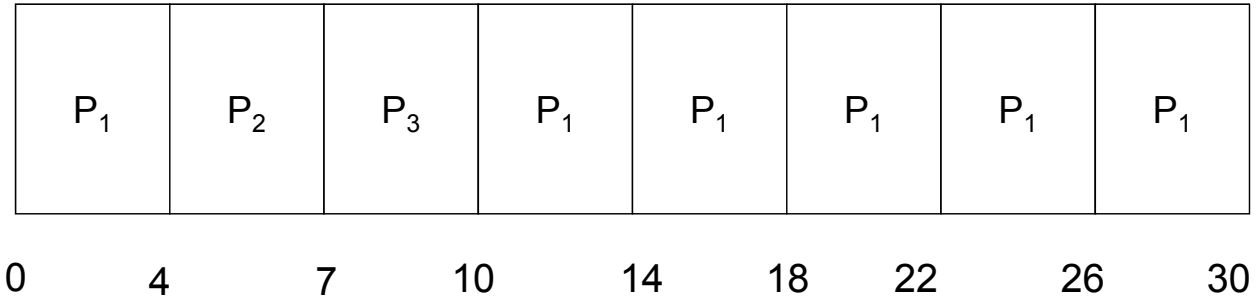
依序循環排班(Round-Robin Scheduling)

- 特別針對分時系統所設計的排程法
- 每一個行程獲得一小段的CPU時間(時間量(time quantum) q)
  - 時間量q一般是10到100個毫秒。
  - 使用完時間量會產生計時器中斷，行程就被搶先，並加到就緒佇列的尾端
- 如果有n個行程在就緒佇列，而時間量是q，每個行程等待的時間不會超過(n－1)xq時間單位
- 平均等待時間比較長，但是能提供較好的反應時間
- 時間量q大小的影響
  - q非常大時，近似FCFS排程的效果
  - q非常小時，會造成內文切換的頻率過高，而影響系統效能
  - q 應該比內文切換的時間長
  - 一般經驗法則：80%的CPU burst時間應該要小於一個時間量的長度

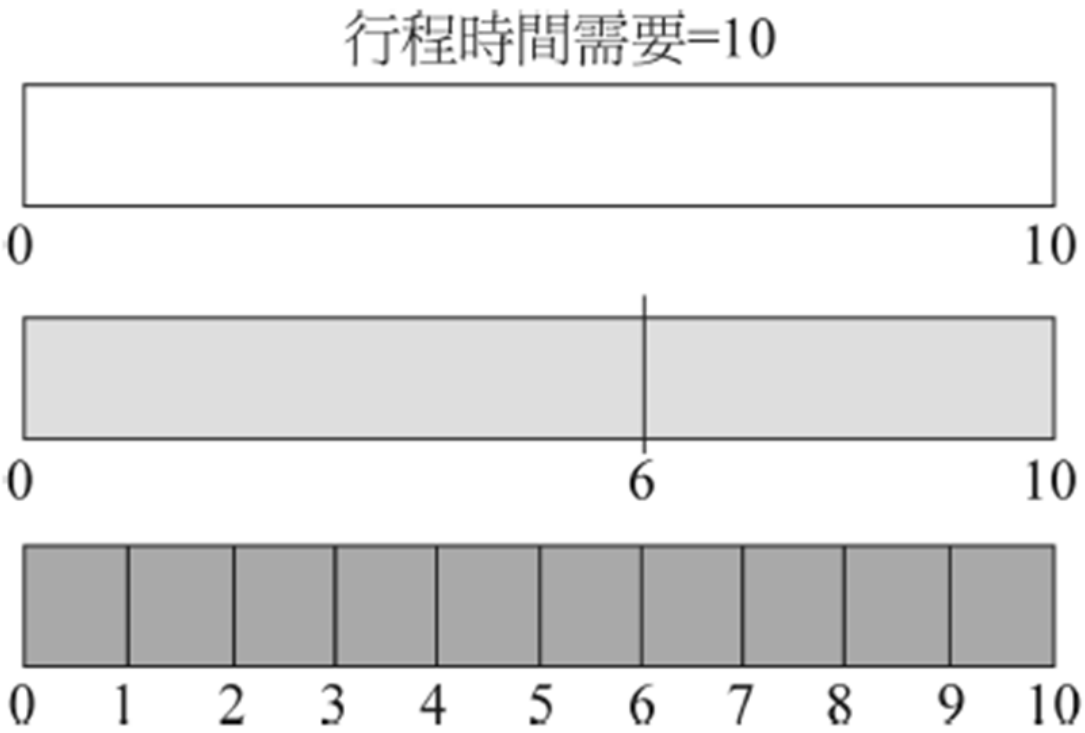
Example of RR with Time Quantum = 4

Process	Burst Time
P1	24
P2	3
P3	3

- The Gantt chart is:

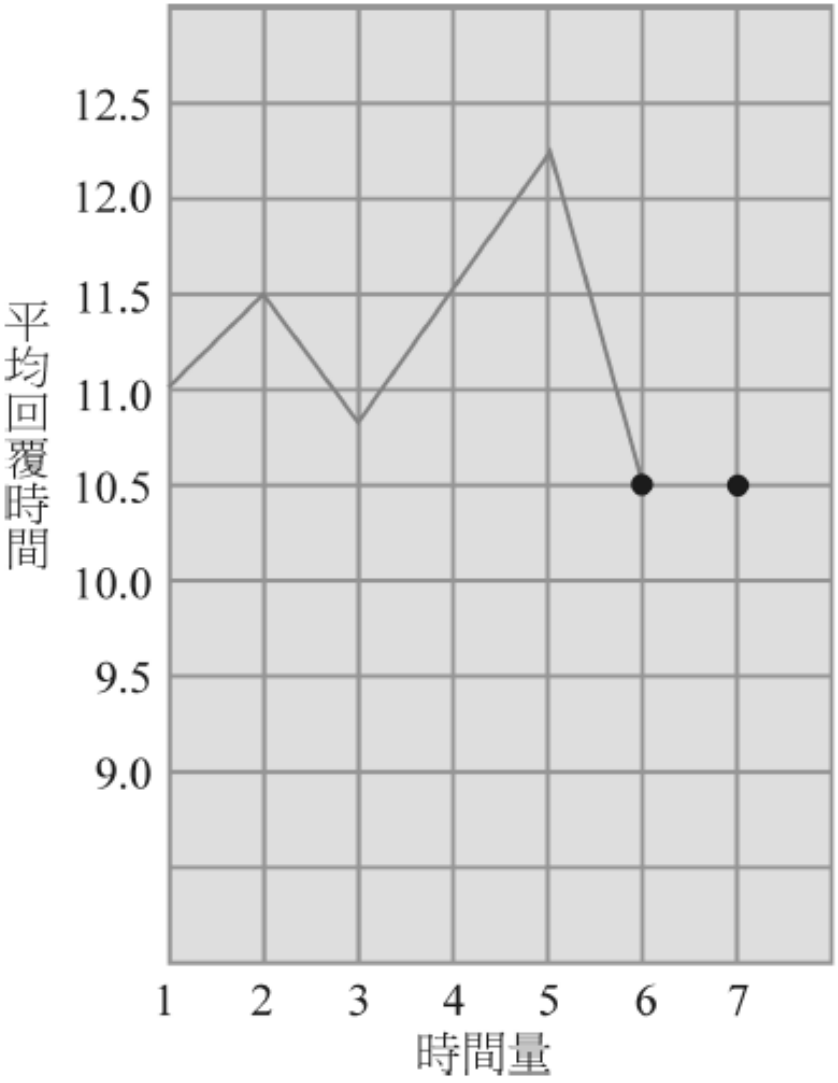


時間量和內容轉換時間



時間量	內容轉換次數
12	0
6	1
1	9

回復時間隨著時間量的不同而變化



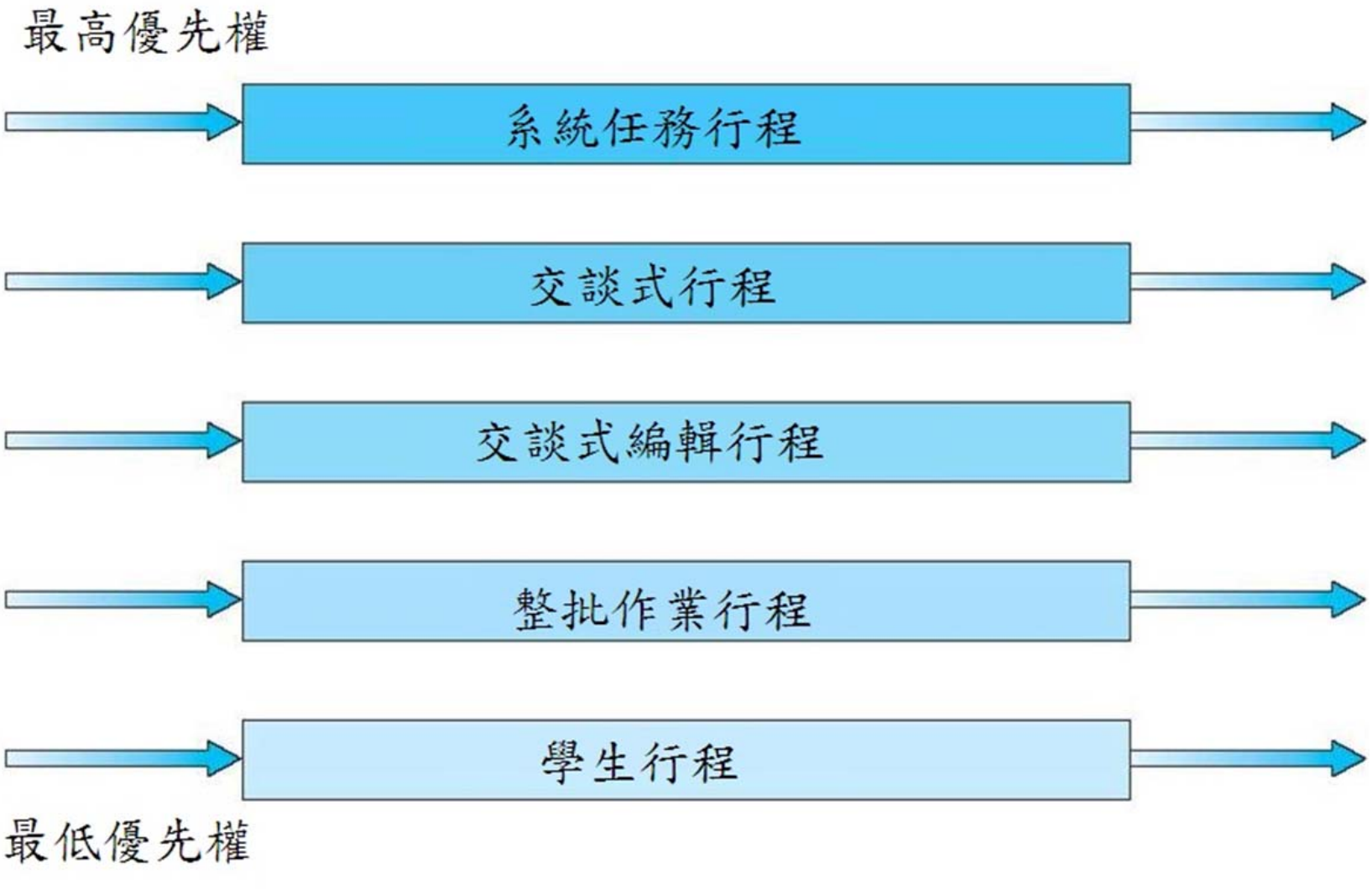
行程	時間
P_1	6
P_2	3
P_3	1
P_4	7

CPU分割的80%應該小於時間量

多層佇列(Multilevel Queue)排班

- 對行程進行分類，將不同類型的行程放入不同佇列中，在每個佇列內使用最適合該類行程的排程方法
  - 例如將行程分成前景行程(foreground process)與背景行程(background process)
- 一個行程永遠只在某個佇列中
- 每一個佇列都有它的排班演算法:
  - 前景 – RR
  - 背景– FCFS
- 在這些佇列之間，需要一個排班演算法
  - 固定優先權排程法
  - 類似RR的排程法
  - 20%的CPU 時間給背景佇列，並以FCFS排班它的行程

多層佇列排班

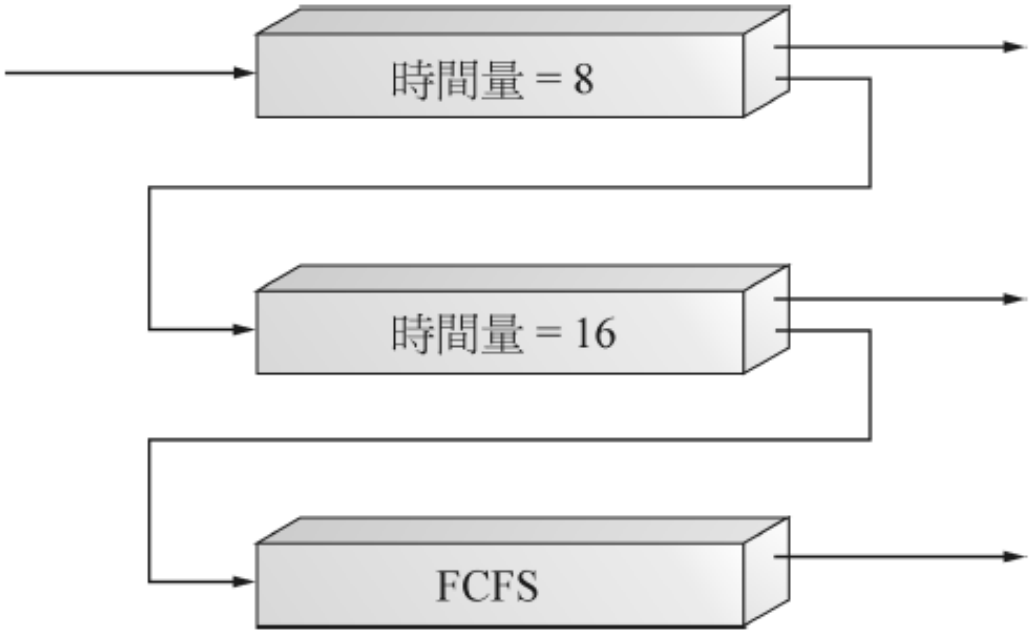


多層回饋佇列(Multilevel Feedback Queue)排班

- 可搶先的優先權排程
- 行程可以在不同的佇列之間移動
- 可以用這種方法實現老化
- 多層回饋佇列排班程式是依據以下參數決定:
  - 佇列個數
  - 每一個佇列的排班演算法
  - 決定什麼時候把行程提升到較高優先權佇列的方法
  - 決定什麼時候把行程降到下層佇列的方法
  - 當行程需要服務時，決定該行程進入那一個佇列的方法

Example of Multilevel Feedback Queue

- 3個佇列:
  - Q_0 – RR 時間量= 8毫秒
  - Q_1 – RR 時間量= 16毫秒
  - Q_2 – FCFS
- 排班
  - 一個新的工作進入以FCFS服務的佇列 Q_0
    - 當他獲得CPU就會有8毫秒
    - 如果它在8毫秒沒有完成，工作就移到佇列Q_1



執行緒排班(Thread Scheduling)

- 區分為使用者層次執行緒和核心層次執行緒
- 如果作業系統支援執行緒，則是執行緒被排班，不是行程
- 多對一和多對多模式，執行緒程式庫排班使用者層次執行緒在LWP上執行
  - 稱為行程競爭範圍(process-contention scope, PCS)，因為排班的競爭是在同一行程內
  - 通常由程式設計者設定優先權完成
- 核心執行緒排班到可取得的CPU上，是系統競爭範圍(system-contention scope, SCS)
  - 競爭發生在系統中所有的執行緒

課堂練習

- 假設有四個行程 P1、P2、P3 和 P4，都在時間 0 到達，順序為 P1、P2、P3、P4
  - 請針對 FCFS、SJF、和不可搶先的優先權排程法：
  - 請畫出與課文類似的行程執行順序圖
  - 計算所有行程的平均等待時間

行程	CPU Burst時間ms）	優先權 （0的優先權最高）
P1	7	1
P2	5	0
P3	3	3
P4	4	1

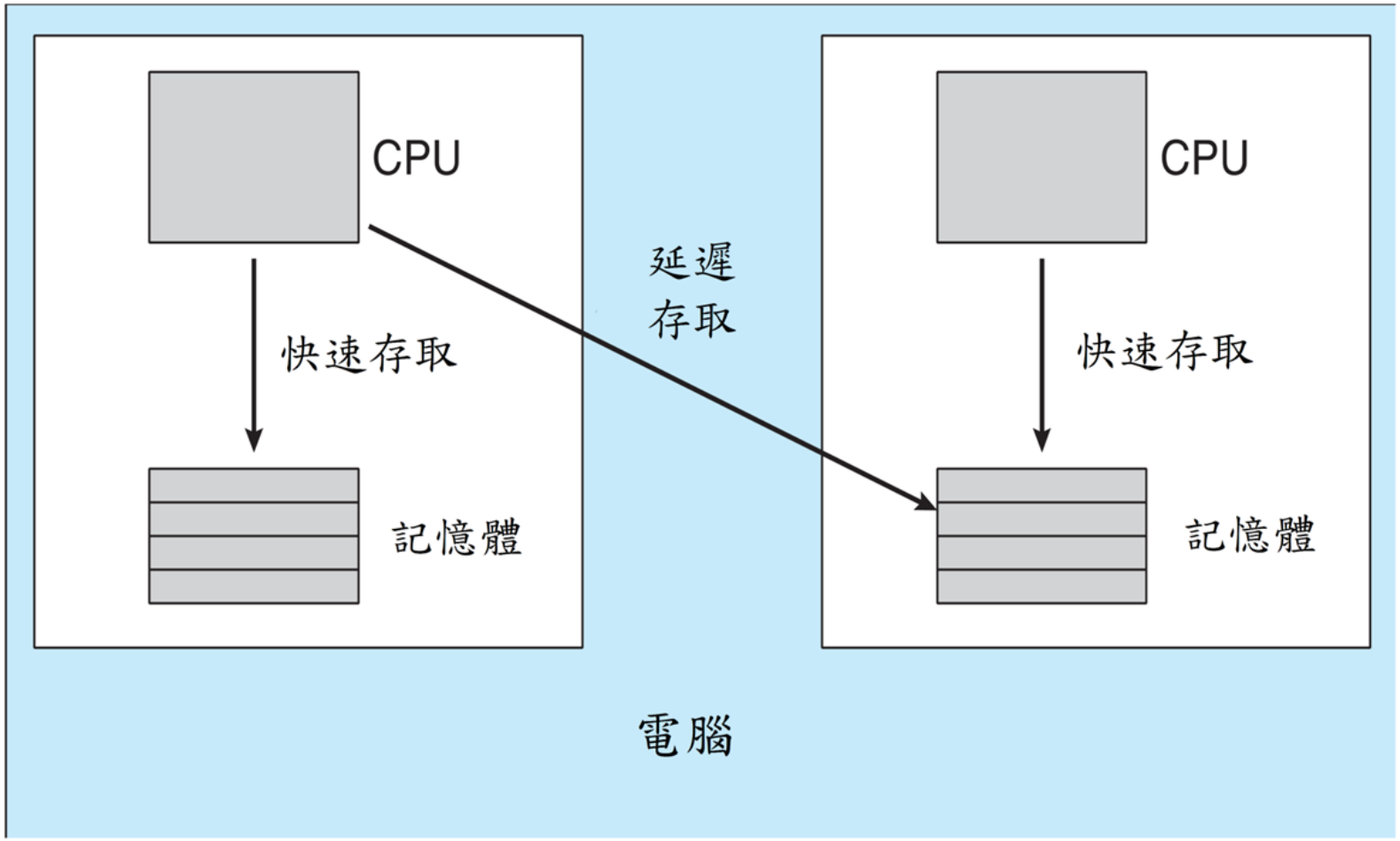
多處理器排班(Multiple-Processor Scheduling)

- 當有多個CPU時，排班問題就變得複雜多了
- 同質處理器(Homogeneous processors)
  - 所有處理器都相同
  - 在此只考慮同質處理器的系統
- 非對稱多元處理 (Asymmetric multiprocessing)
  - 一個主伺服器負責排班決定、I/O處理、其它系統活動
  - 其它處理器只執行使用者程式碼
  - 只有一個處理器存取系統資料結構，減少資料共享的需要
- 對稱多元處理(Symmetric multiprocessing)
  - 每一個處理器自行排班
  - 所有行程共用一個就緒佇列，或每一個處理器有它自己私人就緒佇列
  - 目前最普遍的方式

處理器親和性(Processor Affinity)

- 行程對於它目前執行的處理器較有親和性
  - 因行程最近存取的資料已放在處理器的快取記憶體
- 軟性親和性(soft affinity)
  - 保持一個行程在相同處理器上執行，但不保證它會永遠這麼做
- 硬性親和性(hard affinity)
  - 允許一個行程指定它能夠執行的處理器
- 記憶體架構會影響處理器親和性
  - 在非均勻記憶體存取架構下(non-uniform memory access)亦須考慮親和性

NUMA and CPU Scheduling



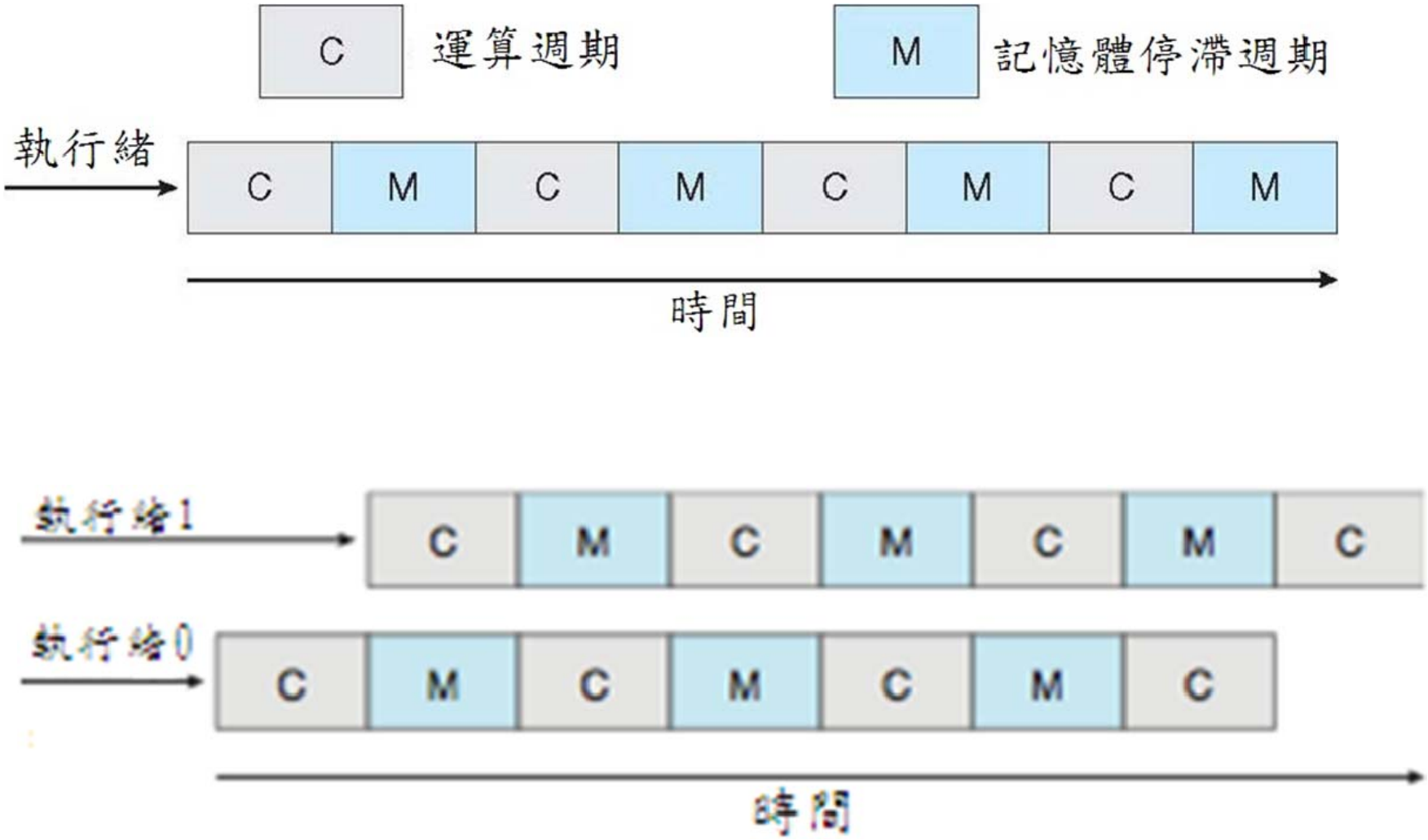
負載平衡(Load Balancing)

- 在SMP系統，為了效率需儘量讓所有的處理器有工作
- 負載平衡(Load balancing)試著讓工作量平均分配，作法有：
  - 推轉移(Push migration) –一特定任務週期性地檢查每個處理器的負載，若發現不平衡，把過度負載的CPU的行程推到其它負載較輕的CPU執行。
  - 拉轉移(Pull migration)–閒置的處理器從忙碌處理器拉出等候執行的行程

多核心(Multicore)處理器

- 最近的趨勢是將多個處理器核心放在同一個實體晶片上
- 速度較快而且消耗較少的能量
- 每一個核心有多個硬體執行緒
  - 利用記憶體存取停滯(memory stall)，讓另一個硬體執行緒在記憶體存取發生時繼續執行

多執行緒多核心系統

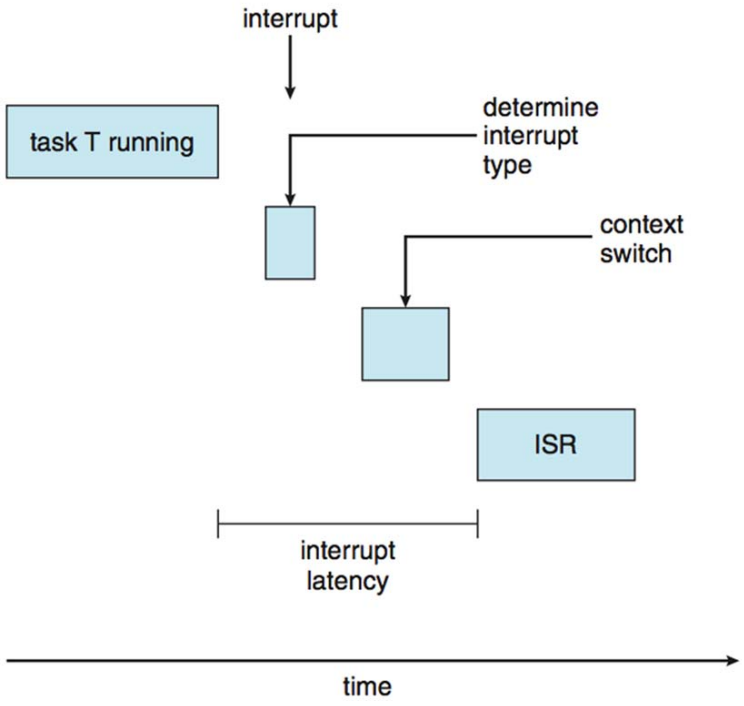

C
運算週期
M
記憶體停滯週期
執行緒
C
M
C
M
C
M
C
M
時間
執行緒1
C
M
C
M
C
M
C
執行緒0
C
M
C
M
C
M
C
時間

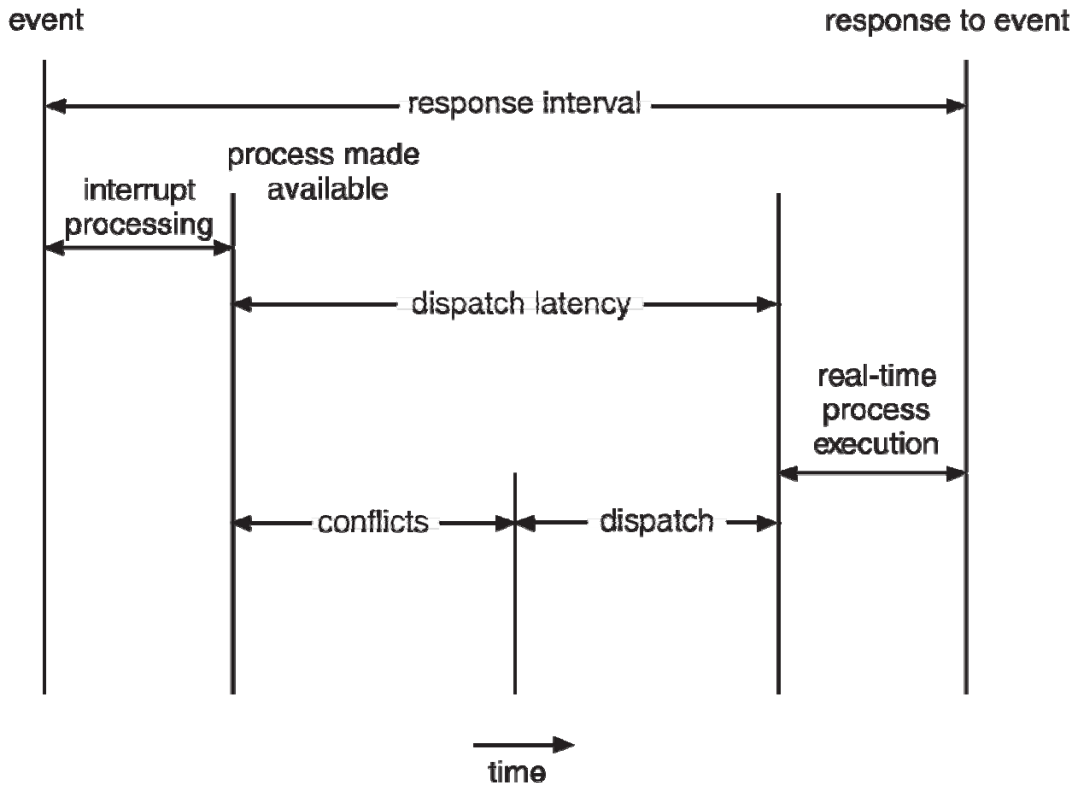
即時(Real-Time)CPU排班

- 可能出現明顯的挑戰
- 軟即時系統(Soft real-time systems)–對於非常即時行程(何時被排班沒有提供保證，僅保證會比非即時行程優先處理。
- 硬即時系統(Hard real-time systems)–任務必須在指定的限期內被服務
  - 所有運算時間的延遲都有其上限, 如存取資料、完成運算的時間。
  - 所採用的硬體與軟體技術會有限制，例如會將資料儲存在記憶體, 而不是儲存在存取時間較不固定的磁碟、光碟上；不會採用作業系統常用的虛擬記憶體(Virtual Memory) 技術, 因為這會比較浪費時間。

即時CPU排班(繼續)

- 兩種延遲時間會影響性能
  - 中斷延遲(Interrupt latency)–中斷到達到開始執行中斷服務常式的時間
  - 分派延遲(Dispatch latency)–停止目前行程到啟動另一個行程的時間



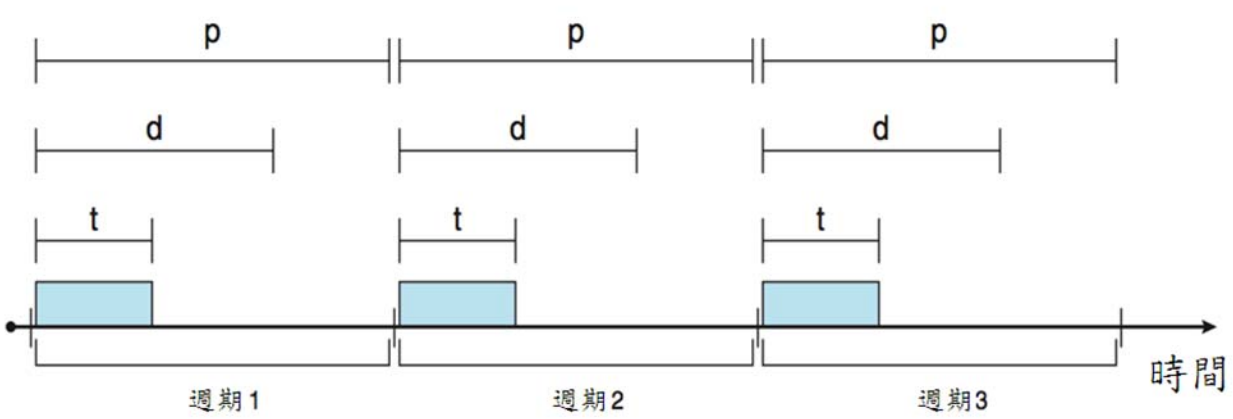


即時CPU排班(繼續)

- 分派延遲的衝突(conflicts)處理:
  - 任何在核心模式執行的行程可被搶先
  - 低優先權行程釋出高優先權行程需要的資源

優先權為基礎的排班(Priority-based Scheduling)

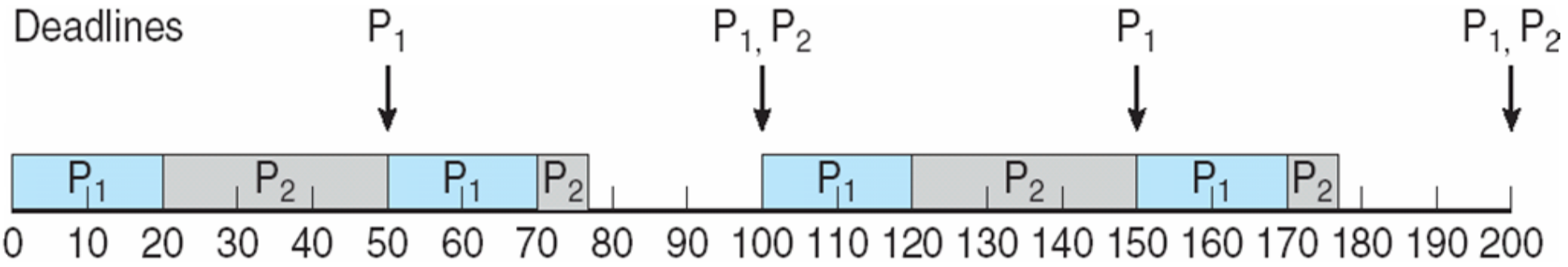
- 對於即時排班，排班器必須支援可搶先式優先權為基礎的排班
  - 但只保證軟即時
- 對於硬即時必須提供其他排班性質以符合截止期限(deadline)的要求
- 被排班的行程的特性:
  - 週期性 (Periodic) –固定的時間間隔發生CPU的使用需求
  - 處理時間 t,截止期限 d,週期 p
    - $0 \le t \le d \le p$
    - 週期任務的:
  -



單調速率排班演算法(Rate-Montonic Scheduling)

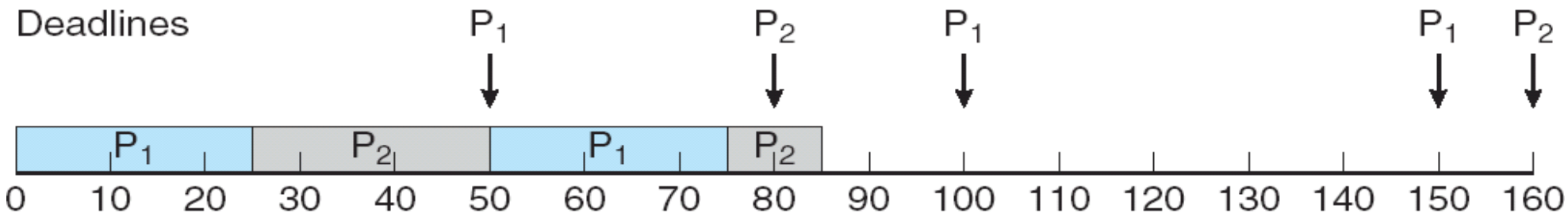
- 行程優先權高低與行程週期成反比
  - 短週期= 較高的優先權
  - 長週期= 較低的優先權

$p_1=50,\ t_1=20,\ p_2=100,\ t_2=35$



超過截止期限範例

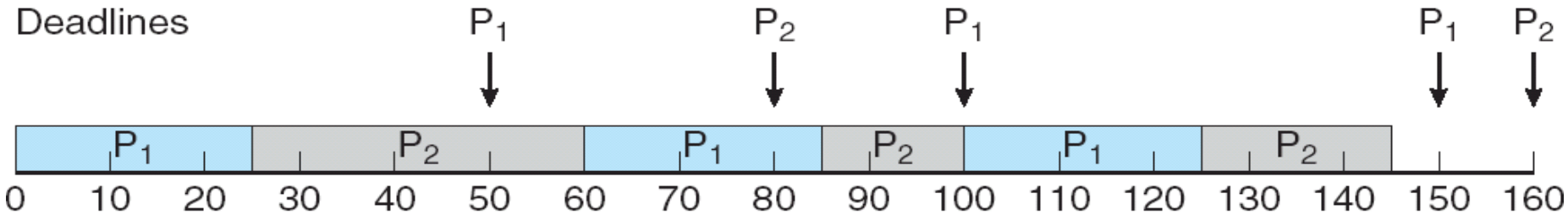
$p_1 = 50$, $t_1 = 25$, $p_2 = 80$, $t_2 = 35$



最早截止期限優先排班(Earliest-Deadline-First Scheduling)

- 優先權是根據截止期限設定
  - 截止期限愈早，優先權愈高
  - 截止期限愈晚，優先權愈低
- 也適用於非週期性行程，及非固定CPU B burst時間長度行程
- 理論上最佳的排班法EDF is theoretically optimal
  - 每一行程可符合其deadline
  - CPU 使用率可達100%

$p_1 = 50,\ t_1 = 25,\ p_2 = 80,\ t_2 = 35$



比例分配排班(Proportional Share Scheduling)

- 將T份時間(share)配置給系統所有應用程式，若一個應用程式可獲得中N份時間，且N < T，則此應用程式將獲得全部處理器時間的N / T。
- 此法必須與許可控制策略(admission-control policy)配合以保證應用程式會得到配置的時間比例。

POSIX即時排班

- POSIX.1b 標準支援即時處理
- 提供API來管理即時執行緒
- 對於即時執行緒定義兩種排班類別：
  - SCHED_FIFO - 使用FCFS 策略來排班FIFO佇列的執行緒。相同優先權的執行緒間沒有時間片段限制
  - SCHED_RR – 和SCHED_FIFO 相似，但相同優先權的執行緒間有時間片段限制
- 定義兩個函數來取得和設定排班策略:
- Pthread_attr_getsched_policy(pthread attr t *attr, int *policy)
- pthread_attr_setsched_policy(pthread attr t *attr, int policy)

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread t tid[NUM THREADS];
   pthread attr t attr;
   /* get the default attributes */
   pthread attr init(&attr);
   /* get the current scheduling policy */
   if (pthread attr getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED OTHER) printf("SCHED OTHER\n");
      else if (policy == SCHED RR) printf("SCHED RR\n");
      else if (policy == SCHED FIFO) printf("SCHED FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
      /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread attr setschedpolicy(&attr, SCHED FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
      pthread join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread exit(0);
}
```

Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

- 在2.5版前，使用傳統UNIX排班演算法的變化
- 2.5版的排班程式被修改成固定級數O(1)的排班
  - 可搶先, 以優先權為基礎
  - 兩個優先權範圍：分時和即時
  - 即時範圍從0 到 99 ，而 nice值從100 到 140
  - 對映到整體的優先權，數值小的表示高優先權
  - 高優先權獲得較大的時間量 q
  - 只要任務的時間片段還沒用完，它就可以執行(active)
  - 如果時間片段用完，就無法執行，直到其他所有有任務都用完它們的時間片段
  - 運作良好,但對於交談式行程的反應時間較差

Linux Scheduling in Version 2.6.23 +

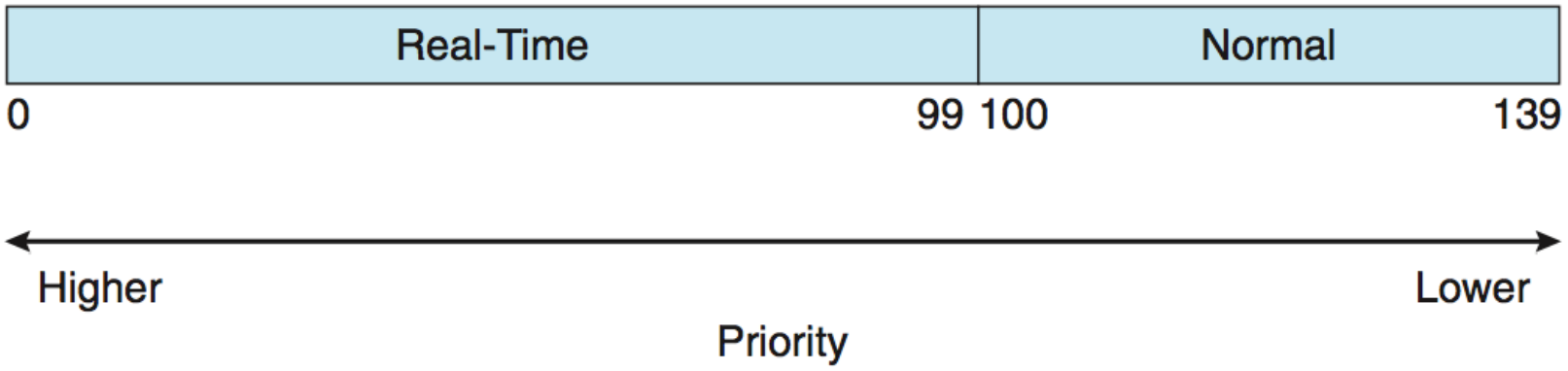
- 完全公平排班程式(Completely Fair Scheduler, CFS)
- 排班類別
  - 每一個排班類別有一個特定的優先權
  - 排班程式挑出最高排班類別的最高優先權任務
  - 取代掉固定時間的時間量，改用CPU時間的比例
  - 排班類別包含以下，但其他類別也可以加入
    - 預設
    - 即時

Linux Scheduling in Version 2.6.23 +

- 時間量的計算是根據nice數值，其範圍從-20到19
  - 較低nice值表示較高的優先權
- 目標潛伏期(target latency)
  - 任務至少應該執行一次的時間間隔
  - 如果活動的任務增加時，目標潛伏期可以增加
- CFS排班器使用vruntime，來維護每一個任務的虛擬執行時間
  - 虛擬執行時間和基於任務優先權的衰減因子相關聯：低優先權有較高的衰減率
  - 一般預設優先權 (nice=0)，虛擬執行時間=真實的執行時間
- 排班器選擇最小vruntime數值的任務為接下來執行的任務

Linux Scheduling (Cont.)

- Linux使用兩個不同的優先權範圍：
  - 即時任務被分配從0到99的靜態優先權
  - 正常的任務被分配從100到139的優先權
- 即時加上正常對映到整體優先權技巧
- 正常任務根據它們的nice值被指定一個優先權
  - Nice值-20對映到整體優先權100
  - Nice值+19整體優先權139



Windows Scheduling

- Windows 使用以優先權為基礎的可搶先排班
- 最高優先權的執行緒先執行
- 分派器(dispatcher)就是排班程式
- 執行緒執行直到 (1)被阻隔,(2)時間量用完,(3)被更高優先權的執行緒搶先
- 即時執行緒可以搶先非即時執行緒
- 32層的優先權技巧
- 可變類別是1-15,即時類別是16-31
- 優先權0是一個記憶體管理執行緒
- 每一個排班的優先權使用一個佇列
- 如果沒有可執行的執行緒，就執行叫做閒置執行緒(idle thread)

Windows Priority Classes

- Windows API會分辨出一個行程是屬於下面那一個優先權類別
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - 除了REALTIME之外，其他類別的行程其優先權類別是可變動的
- 在一個優先權類別內的執行緒有相對的優先權
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- 優先權類別和相對優先權組合成一個優先權的數字
- 在一個類別中基本的優先權是NORMAL

Windows Priority Classes (cont.)

- 如果時間量用完，優先權降低，但不會低於基本優先權
- 如果等待發生時，根據等待的是什麼，提升其優先權
- 前景視窗獲得3倍優先權的提升
- Windows 7加入使用者模式排班(user-mode scheduling,UMS)
  - 應用程式獨立於核心外自行產生和管理執行緒
  - For large number of threads, much more efficient對於大量執行緒，更有效率
  - 排班器是來自類似C++程式語言的ConcRT函數庫

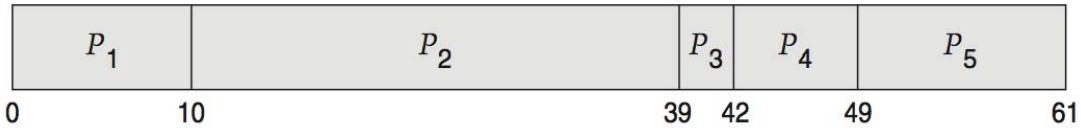
排班演算法效能評估(Algorithm Evaluation)

- 對於一個作業系統如何選擇CPU排班演算法?
- 決定評估的標準，然後評估演算法
- 定量模式(Deterministic modeling)
  - 分析式評估(analytic　evaluation)的型態
  - 以一個特殊預定的工作量進行每種演算法的效能評估。
- 考慮5個行程在時間0到達：

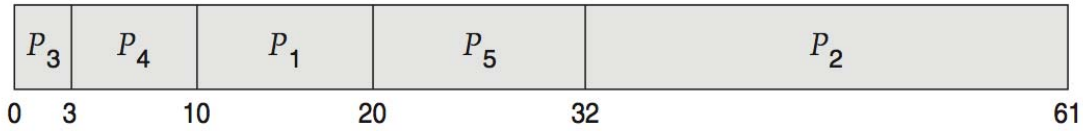
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

定量評估(Deterministic Evaluation)

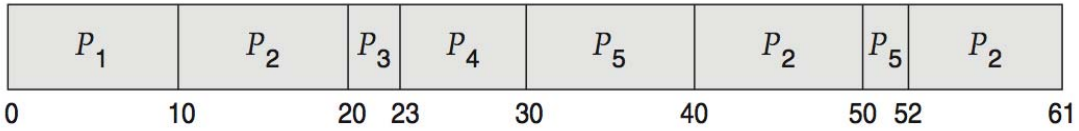
- 對於每一種演算法計算最小平均等待時間
- 簡單且快速，但僅適用這些數據
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



排隊(佇列)模式(Queueing Models)

- 以機率方式描述行程到達、CPU Burst、I/O Burst
  - 通常是指數、並且以平均值描述
  - 計算平均產量、使用率、等待時間等
- 電腦系統描述成一個伺服器組成的網路, 每一個伺服器都有一個等待行程的佇列
  - 知道到達率(arrival rate)和服務(service rate)
  - 計算使用率、平均佇列長度、平均等待時間等

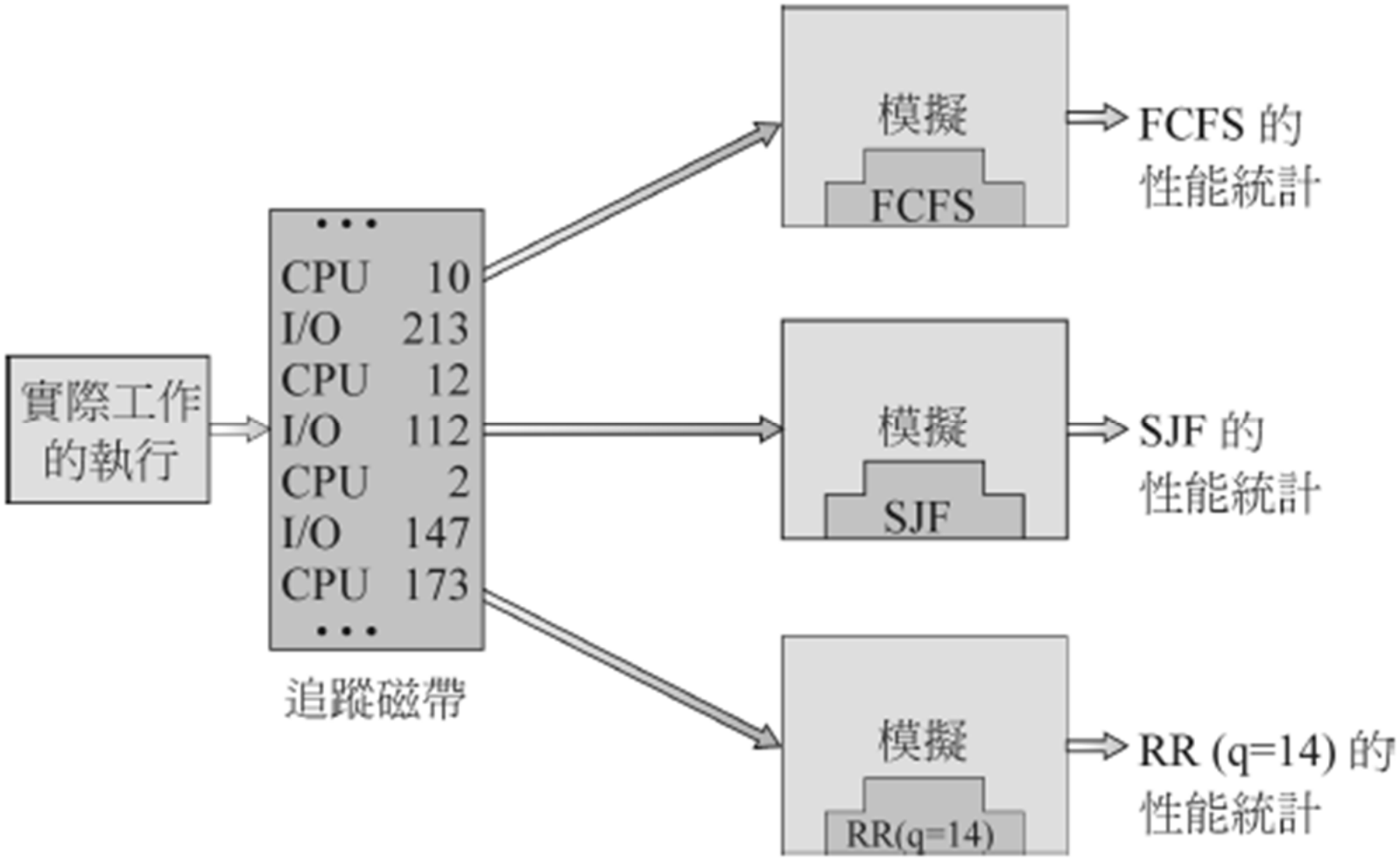
Little's Formula

- n =佇列平均長度
- W =在佇列的平均等待時間
- λ=平均到達佇列的速率
- 李特氏公式– 在穩定狀態，離開佇列的行程必須等於到達行程，所以 n = λ x W
  - 任何排班演算法和到達分布都有效
- 例如，如果平均每秒鐘有7個行程到達，並且一般有14個行程在佇列中，則每個行程的平均等待時間為2秒鐘

模擬(Simulations)

- 佇列模型有所限制
- 模擬則較準確
  - 電腦系統的程式模型
  - 時鐘是一個變數
  - 收集統計資料以指出演算法的效能
  - 使用以下的方式收集模擬資料
    - 依照機率分佈隨機產生模擬資料
    - 分佈的情形可以用數學或是用經驗的方式定義
    - 根據磁帶記錄獲得系統中實際事件的情形

以模擬方式評估CPU排班程式



實作(Implementation)

- 即使是模擬精確度也是有限的
- 實作新的排班器，並在實際系統測試
  - 高代價, 高風險
  - 環境會改變
- 大多數彈性的排班器可以針對各別的電腦或系統做修改