

# 考古

## 2022

### 第七題

7. (15pt) Answer the following questions regarding synchronization. The simplest software tool to solve the critical-section problem is the *mutex lock*. Instead of implementing mutex locks using atomic hardware instructions, we can simply implement two operations for acquiring and releasing the lock as follows:

```
acquire()
{
    while (!available)
        ;
    available = false;
}
release()
{
    available = true;
}
```

- (1) What's the problem with this implementation? Please compare the effects of such implementation on uniprocessor and multiprocessor systems, respectively. (5pt)
- (2) What changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available? (5pt)
- (3) What is the idea of *semaphore*? Please compare its difference in usage with mutex locks. (5pt)

1. 在 uniprocessor 會有效，因為一次只有一個執行緒可以運行。但multiprocessor會出現下列問題：

1. 競爭條件 (Race Condition)：當多個執行緒同時檢查 `available` 變量時，可能會出現競爭條件。例如，兩個執行緒都檢查 `available` 為 true，然後同時進入臨界區域，導致競爭條件的發生。
2. 忙等待 (Busy Waiting)：在 `acquire()` 中，當資源不可用時，執行緒會進入一個無限循環，不斷檢查 `available` 變量，這將導致 CPU 資源的浪費。在單處理器系統上，這可能會導致整個系統變得非常緩慢，因為其他執行緒無法獲得 CPU 時間片。在多處理器系統上，這可能會導致系統的整體效能下降，因為多個執行緒在忙等待。
3. 可見性 (Visibility)：在多處理器系統上，由於 CPU 緩存的存在，一個執行緒對 `available` 的修改可能不會立即被其他 CPU 緩存中的執行緒看到，這可能導致不一致的狀態。

2. 為了避免忙等待，需要引入阻塞原語。當一個進程嘗試獲取已被持有的鎖時，它應該被阻塞(sleep)並加入等待隊列，而不是忙式等待。當鎖被釋放時，等待隊列中的某個進程



會被喚醒並獲得鎖。這樣可以避免CPU周期的浪費。

```

acquire()
{
    // 如果鎖可用，則直接獲取鎖
    if (available) {
        available = false;
    } else {
        // 如果鎖不可用，則將進程放入等待隊列中
        add_to_waiting_queue(current_process); // current_process 表示當前執行的進程
        block(current_process); // 將當前進程阻塞
        // 當鎖變為可用時，會通過某種機制（例如鎖被釋放）喚醒進程
    }
}

```

3. ~

1. 計數機制：信號量的計數可以大於一，允許多個線程或進程同時訪問共享資源，但訪問數受到計數限制。互斥鎖通常只允許一個線程或進程同時獲取鎖。
2. 使用方法：互斥鎖主要用於互斥，即一次只有一個線程可以訪問代碼的臨界區段，以防止競爭條件。信號量則可以用於更廣泛的同步任務，包括控制對多個資源實例的訪問、在線程之間發出信號以及同步生產者-消費者方案。
3. 操作：信號量支援兩個主要操作：‘等待’（也稱為 ‘P’ 或 ‘down’），它減少信號量計數並阻塞調用者，如果計數變為零；‘信號’（也稱為 ‘V’ 或 ‘up’），它增加信號量計數並喚醒被阻塞的進程（如果有）。互斥鎖通常只支援‘鎖定’（獲取）和‘解鎖’（釋放）操作。
4. 資源管理：信號量可以用於管理除互斥以外的資源，例如控制對資源池的訪問或限制訪問特定資源的線程數量。互斥鎖主要用於臨界區段的保護，不支援內置的資源計數。

文  
A

## 2023

### 第一題

1. (10pt) Modern computers are designed based on *von Neumann architecture*.
  - (1) Please list the major components of a modern computer system, and describe how they are connected. (5pt)
  - (2) Please give the detailed steps of actions taken by these components during an I/O operation (e.g. reading or writing a file). (5pt)

1. CPU、memory、輸入/輸出設備(IO)和系統匯流排(BUS)。它們通過系統匯流排相互連接。
2. 應用程序發出I/O請求->操作系統捕獲並處理該請求->檢查設備狀態->與設備控制器協調傳輸->實際傳輸數據->處理完成中斷->返回應用程序。

### 第二題

2. (10pt) What's the purpose of *interrupts*? How do we usually handle interrupts in OS kernel? Why are modern operating systems usually interrupt-driven? What are the benefits of this design?

- 中斷的作用是允許外部事件及時響應，不至於長時間占用CPU等待。

- 通常通過中斷服務例程(ISR)來處理各種中斷。儲存原本 process 的進度，從另一個 process 一半的進度繼續。
- 現代OS通常採用中斷驅動設計,主要好處是提高了實時響應能力和系統效率。

## 第三題

3. (20pt) Regarding the following questions about *process and thread management*, please indicate whether each statement is *true or false*. If a statement is incorrect, please explain the reasons to get the full score. (\*not\* just correcting the error)
- (1) Multithreaded programs can always provide better performance than a single-threaded solution. (4pt)
  - (2) Multiple user-level threads, as managed by programmers, can always be run in parallel on multicore systems. (4pt)
  - (3) Kernel-level threads are directly supported and managed by OS kernel. (4pt)
  - (4) The dominant factor of performance gains from adding additional CPU cores to a system is the number of cores. (4pt)
  - (5) A multithreaded program using multiple user-level threads achieves better performance on a multiprocessor system than on a single-processor system. (4pt)

1. 錯誤。多線程無法每次都比單線程解決方案性能更好,這取決於問題的特性和並行度等因素。
2. 錯誤。用戶級線程的並行運行受限於單CPU的硬件並行度。
3. 正確。
4. 錯誤。多核CPU的性能提升主要受限於問題的並行度、高速緩存一致性開銷等多方面因素。
5. 正確。在多處理器系統上,適當使用多線程能夠更好地利用硬件資源,從而提高性能。

## 第四題

4. (15pt) Answer the following questions regarding *deadlocks*:

(1) Among the different methods for handling deadlocks, what are the differences between *deadlock prevention* and *deadlock avoidance*? What are the possible issues when applying each method? (5pt)

(2) Consider the following snapshot of a system with five processes P<sub>0</sub> through P<sub>4</sub> and four resource types A, B, C, and D:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C D	A B C D	A B C D
P <sub>0</sub>	2 0 0 1	4 2 1 2	3 3 2 1
P <sub>1</sub>	3 1 2 1	5 2 5 2	
P <sub>2</sub>	2 1 0 3	2 3 1 6	
P <sub>3</sub>	1 3 1 2	1 4 2 4	
P <sub>4</sub>	1 4 3 2	3 6 6 5	

Answer the following questions using the *banker's algorithm*:

(a) Determine whether or not the system is safe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe. (5pt)

(b) If a request from process P<sub>1</sub> arrives for (1, 1, 0, 0), can the request be granted immediately? What about a request from process P<sub>0</sub> for (1, 1, 0, 0), can the request be granted immediately? Please justify your answers. (5pt)

文

1. 死鎖預防是確保四個死結必要條件(互斥、佔有並等待、非搶占式和循環等待)其中至少一個永遠不會滿足,這樣就根本不可能發生死結。這種方法簡單直接,但可能會導致資源利用率低和系統吞吐量降低。死鎖避免是通過持續監控系統狀態,確保不進入死鎖狀態。常用方法是銀行家算法,它通過分析進程的資源需求和資源可用情況,確保系統不進入不安全狀態。雖然這種方法提供了更好的資源利用和系統效率,但可能需要較高的計算和監控成本。此外,確定進程的最大資源需求可能會很困難,因此在某些情況下可能會面臨設計挑戰

2. ~

## Max - Allocation

	<u>Need</u>			
	A	B	C	D
P <sub>0</sub>	2	2	1	1
P <sub>1</sub>	2	1	3	1
P <sub>2</sub>	0	2	1	3
P <sub>3</sub>	0	1	1	2
P <sub>4</sub>	2	2	3	3

文

- P<sub>0</sub> → P<sub>1</sub> → P<sub>2</sub> → P<sub>3</sub> → P<sub>4</sub>, safe state
- 將 available - (1,1,0,0), P[i]\_Allocation + (1,1,0,0) 進行banker's algorithm 查看是否 safe

## 第五題

5. (10pt) Answer the following questions regarding synchronization. The simplest software tool to solve the critical-section problem is the *mutex lock*. One simple implementation of mutex lock is *spinlocks*.

- (1) What's the problem with spinlocks? Please compare the effects of such implementation on uniprocessor and multiprocessor systems, respectively. (5pt)
- (2) What is the idea of *semaphore*? Please compare its differences from mutex locks. (5pt)

1. 自旋鎖的問題在於它在取得鎖失敗時會一直主動循環查詢(自旋)是否可以獲得鎖,從而浪費大量CPU週期。在單處理器系統中這種開銷是很大的,但多處理器系統上自旋時可以被調度出去執行其他進程。
2. semaphore的想法是維護一個可被多個進程存取的整數值, 對其進行P/V操作來獲取/釋放資源。它與互斥鎖最大的區別是一個semaphore可以同時被多個進程持有(值大於1),支援更複雜的同步需求。互斥鎖是一個二值訊號量,一次只允許一個thread訪問resource。

## 第六題

6. (30pt) Answer the following questions regarding memory management.
- (1) What are the differences between *pure paging* and *pure segmentation* in terms of the following aspects: address translation structures, fragmentation, ability to share code across processes, and memory protection? (15pt)
  - (2) Consider a paging system with the page table stored in memory. Assume that a memory reference takes 80 nanoseconds. If we add TLBs, and we want the effective memory access time to be less than 10% slowdown, what is the minimum percentage of cache hit in the TLBs? (Assume that finding a page-table entry in the TLBs takes 4 nanoseconds, if the entry is present.) (5pt)
  - (3) What are the possible impacts of a large logical address space in modern computer systems (e.g. 64-bit address) on the structure of page tables? Please list some possible solutions. (5pt)
  - (4) What is the idea of virtual memory? How do we usually implement it with *demand paging* technique? (5pt)

1. ~

- address translation structures: 純分頁用page table，純分段用segment table
- fragmentation: 純分頁存在內部碎片化，因為頁面大小固定，而純分段有外部碎片化，因為段的大小可變
- ability to share code across processes: 分頁方便共享，分段也支持但要共享段
- memory protection: 分頁按頁設定保護權限，分段按段設定

2. ~

有效記憶體存取時間 (EMAT) = TLB命中 (TLB時間 + 主記憶體時間) + TLB未命中 (TLB時間 + 2 × 主記憶體時間)

$$\begin{aligned}
 p \times 84 + (1-p) \times (4 + 80 \times 2) &\leq 84 + (84 \times 0.1) \\
 \Rightarrow 84p - 164p &\leq 92.4 - 84 \\
 \Rightarrow -80p &\leq -11.6 \Rightarrow p \geq 0.895 \quad \text{#}
 \end{aligned}$$

文A

3. ~

- 影響: 增大頁表大小，需要更多層級，導致更大的記憶體占用
- sol: multi-level page tables、壓縮頁表、大頁面

4. ~

- 概念: 虛擬記憶體允許程序表現出超過實體記憶體容量的記憶體使用量，增強multi process運行的安全性和效率
- 實現方式: 通過demand paging實現，只在程序訪問時才將所需頁面從硬碟加載到記憶體，並在記憶體滿時進行頁面置換

## 第七題

7. (20pt) Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Priority	Burst Time
P <sub>1</sub>	2	2
P <sub>2</sub>	1	1
P <sub>3</sub>	4	8
P <sub>4</sub>	2	4
P <sub>5</sub>	3	5

The processes are assumed to have arrived in the order P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>5</sub>, all at time 0. Note that a *larger* priority number denotes a *higher* priority.

Answer the questions using the following scheduling algorithms: **SJF**, **non-preemptive priority**, and **RR** (time quantum=2).

(1) What is the *turnaround time* of each process for each of the scheduling algorithms? (10pt)

(2) What is the *waiting time* of each process for each of the scheduling algorithms? Which of the algorithms results in the minimum average waiting time (over all processes)? (10pt)

文

1. 各進程在三種調度算法下的周轉時間(turnaround time)為:

- SJFS: P<sub>2</sub>(1) P<sub>1</sub>(3) P<sub>4</sub>(7) P<sub>5</sub>(12) P<sub>3</sub>(20)
- 非剝奪優先級: P<sub>3</sub>(8) P<sub>5</sub>(13) P<sub>1</sub>(15) P<sub>4</sub>(19) P<sub>2</sub>(20)
- RR(q=2): P<sub>1</sub>(2) P<sub>2</sub>(3) P<sub>4</sub>(13) P<sub>5</sub>(18) P<sub>3</sub>(20)

2. 各進程在三種調度算法下的等待時間(waiting time)為:

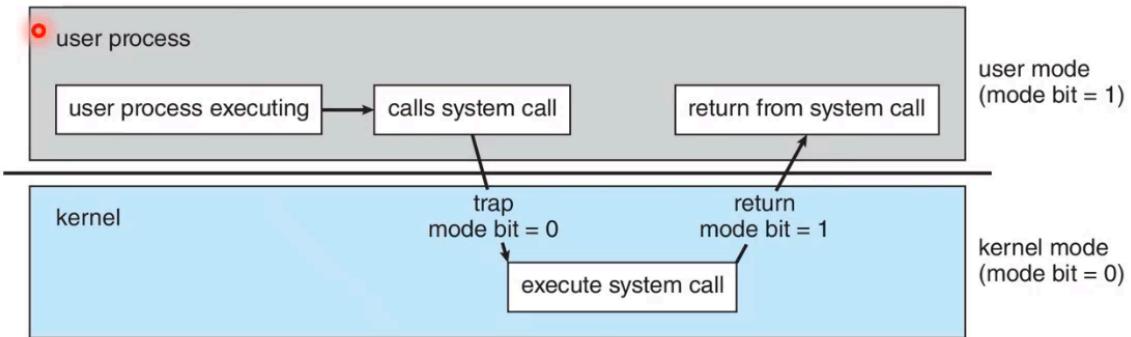
- SJFS: P<sub>2</sub>(0) P<sub>1</sub>(1) P<sub>4</sub>(3) P<sub>3</sub>(7) P<sub>5</sub>(12) => 23/5 = 4.6
- 非剝奪優先級: P<sub>3</sub>(0) P<sub>5</sub>(8) P<sub>1</sub>(13) P<sub>4</sub>(15) P<sub>2</sub>(19) => 55/5 = 11
- RR(q=2): P<sub>1</sub>(0) P<sub>2</sub>(2) P<sub>4</sub>(9) P<sub>5</sub>(13) P<sub>3</sub>(12) => 36/5 = 7.2

SJF waiting time 最小

## Chapter 1 : Introduction

- Interrupt : 可以讓 CPU 跟 I/O 同時做事情
- Dual-mode : OS 區隔權限成兩種模式(User mode, kernel mode)

- CPU 裡有些指令集需要權限才能執行



- Timer : 避免程式當機、無窮迴圈資源被限制住
- 如何同時執行多種作業系統?
  - Emulation : 進行不同作業系統之間的轉換
  - Virtualization : 透過虛擬機

## 作業系統中的CPU資源分配

### 1. 多重程式(Multiprogramming)

文

- 概念:
  - 當某程式暫時不需使用CPU的時候，監控程式會啟動另外的正在等待CPU資源的程式。目的是讓CPU資源能充分被使用。
- 優點:
  - 看似原始但在當時確實大幅提高了CPU使用率。
- 缺點:
  - 程式之間的排程策略太過粗糙，因為CPU是前一程式不使用時才能輪到其他需要CPU資源的程式使用CPU，也就是程式沒有優先序高低的差別；這會讓一些急需使用CPU完成任務的程式(例如使用者互動的任務)，可能需要等待很長的時間才分配到CPU資源。這會造成你在你的laptop按了一下滑鼠、或者在鍵盤上敲了幾個鍵，結果過了10分鐘系統才有反應。

### 2. 分時系統(Time-Sharing System)

- 概念:
  - 經過稍微改進後，程式執行模式變成一種合作模式：每個程式執行一段時間後，會主動讓出CPU給其他程式。
- 優點:
  - 在一小段時間內讓每個程式都有機會被執行到，對一些互動式任務來說很重要，可以解決前述問題，按下鍵盤滑鼠時，程式需要處理的任務可能不多，但需要系統盡快處理任務，讓使用者能立即看到結果。
- 缺點:
  - 如果遇到程式在進行很耗時的計算，一直霸佔著CPU資源，那作業系統也沒辦法，其他程式都只能等著。整個系統會看起來像當機一樣卡住。例如程式進入

while(1)無窮迴圈，會讓整個系統都卡住。

### 3. 多工(Multi-tasking)系統

因為前兩種排程方式的缺點，因而演進成我們現在很熟悉的多工(Multi-tasking)模式。

- 概念：使用一種叫先佔式(Preemptive)的CPU分配方式
  - 作業系統接管了所有硬體資源，且本身是執行在受硬體保護的級別。
  - 所有程式都以行程(Process)的方式執行在比作業系統權限更低的級別，每個行程都有自己獨立的位址空間、有單獨的記憶體，使行程間的位址空間互相隔離。
  - CPU由作業系統統一分配：
    - 每個行程根據行程優先序都有機會得到CPU資源，但如果執行時間超過一定時間，作業系統會暫停該行程，將CPU資源釋出給其他等待執行的行程。
    - 作業系統可強制剝奪CPU資源並分配給它認為目前最需要的行程。
    - 如果作業系統分配給每個行程的時間都很短，即CPU在多個行程間快速切換，看起來會像是很多行程都在同時執行。
  - 目前幾乎所有現代OS都是採用這種方式，比如UNIX、Linux、Windows NT以後的版本、Mac OS X以後的版本。

### 各方法資訊整理

	Batch	Multi-programming	Time-sharing (Multi-tasking)
System Model	Single user Single job	Multiple prog.	Multiple users Multiple prog.
Purpose	Simple	Resource utilization	Interactive Response time
OS features	N.A	CPU scheduling Memory Mgt. I/O system	File system Virtual memory Synchronization Deadlock

# Management

## Process Management

- Program 是靜態的， process 是動態的
- 管理 CPU, memory, I/O 等
- Single-threaded process
  - program counter : 紀錄下個指令的位置
- Multi-threaded process
  - program counter : 指向某個 thread

## Memory Management

- 有效配置有限的記憶體空間

## File-System Management

- 有效地儲存檔案資料
- 管理檔案權限

文

## Mass-Storage Management

- 有效配置有限的硬體空間
- Disk scheduling : 探討 HDD 如何進行有效的讀寫頭移動

## Caching

- CPU 的 caching 是把從 memory 讀到 register 的資料暫存
- 硬碟的 caching 是把經常使用的資料暫存在 memory 的 caching

## I/O Subsystem

- buffering
- caching
- spooling

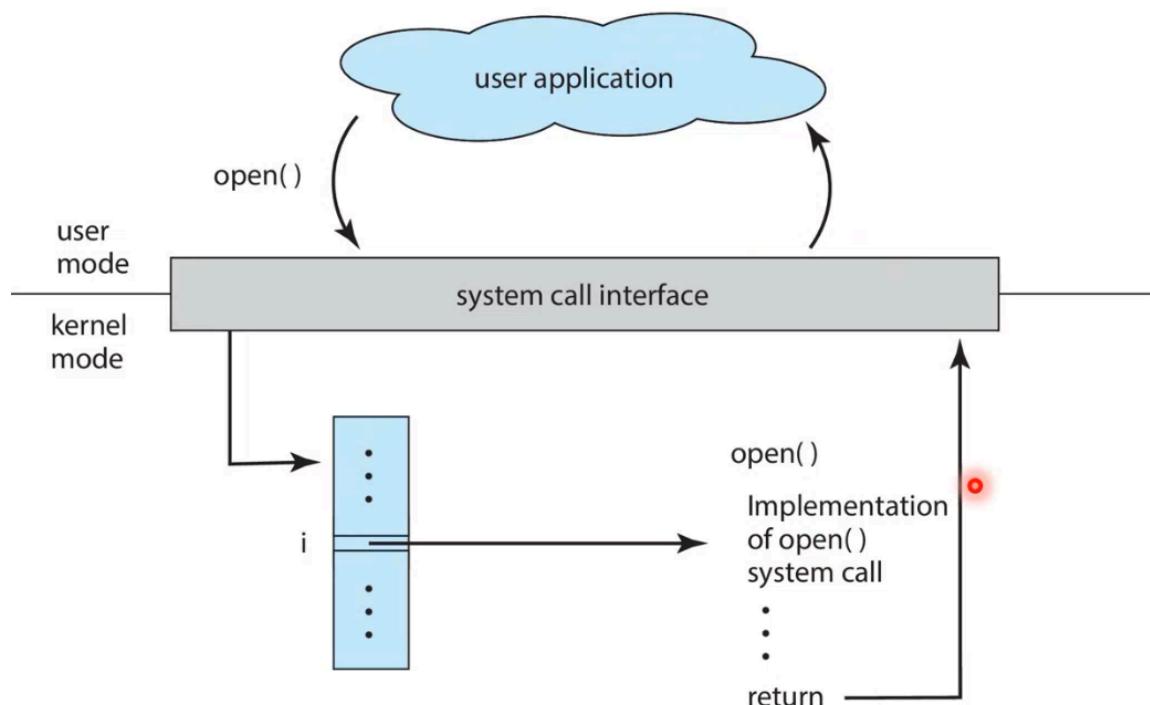
# Chapter 2 : Operating-system Services

- 使用者角度
  - User interface : CLI, GUI, touch-screen, Batch
  - Program execution
  - I/O operations

- File-system manipulation
- Communications
- Error detection
- 系統角度
  - Resource allocation
  - Logging
  - Protection and security

## System Calls

- 寫高階程式語言並透過 API 呼叫
  - Win32 API for Windows
  - POSIX API for UNIX, Linux, and Mac OS X...



## Types of System Calls

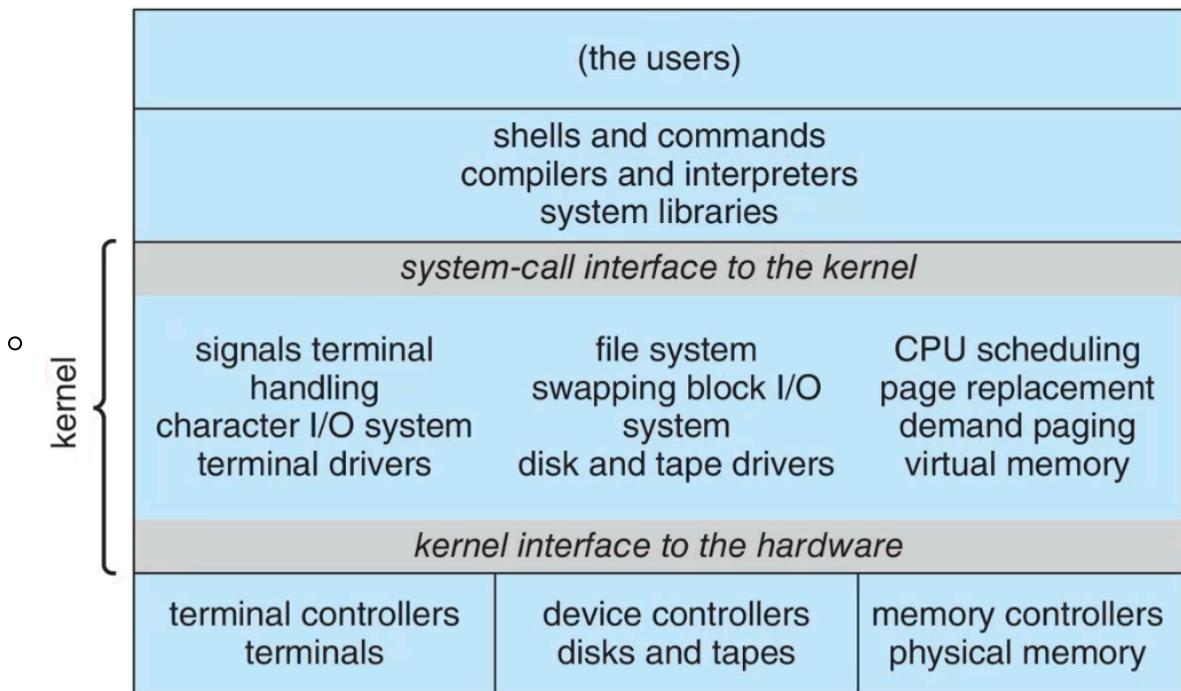
- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

## DMA ( Direct Memory Access , 直接存儲器訪問 )

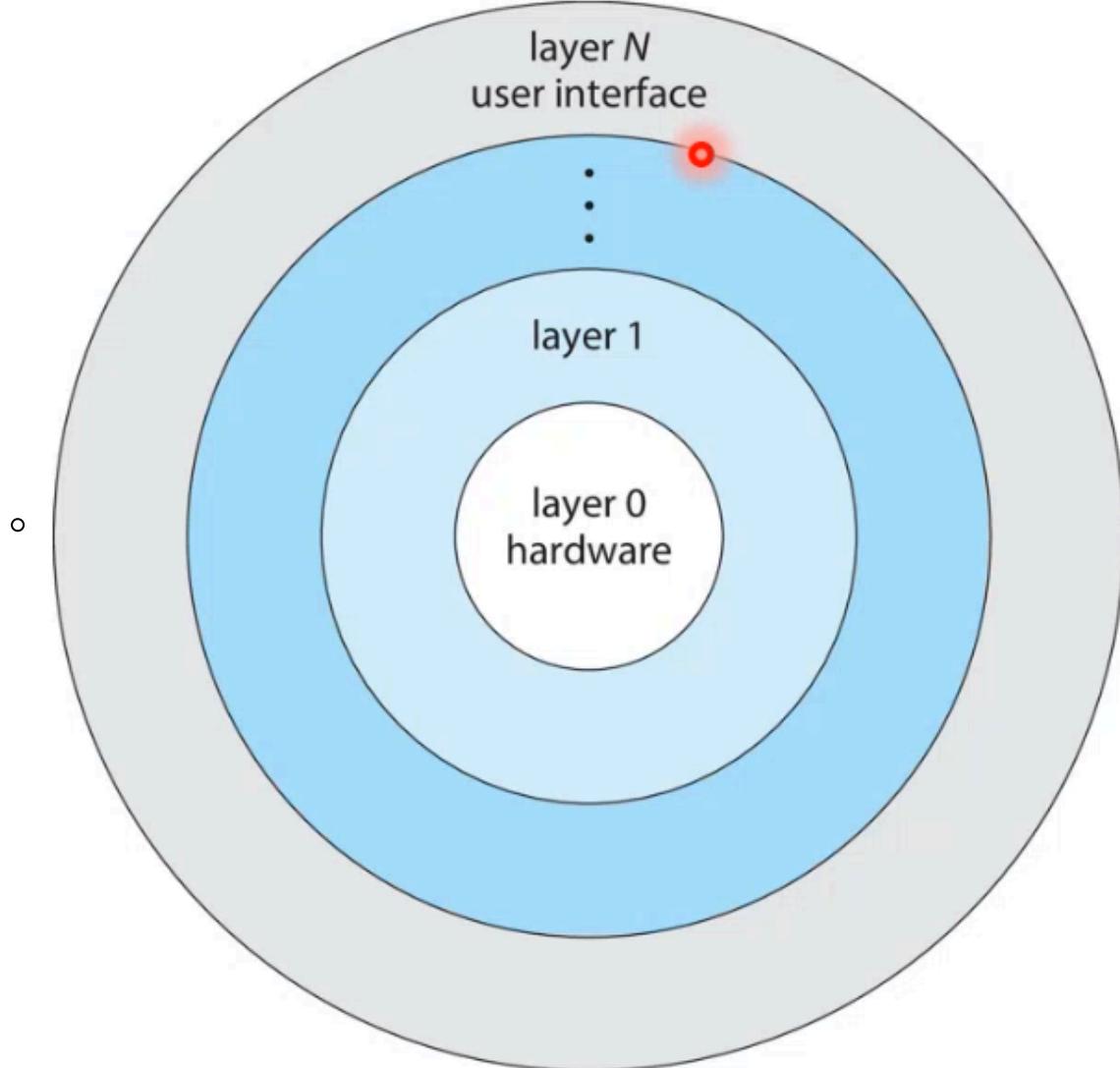
- DMA 允許外部設備 ( 如硬盤驅動器、網絡接口卡等 ) 直接訪問主記憶體，而不需要 CPU的介入，提高數據傳輸的效率。

## Operating System Structure

- Simple structure(最簡單的) - MS-DOS
- More complex(還是很簡單、沒有層次) - UNIX
  - Linux 屬於這種進階，因為 kernel 有模組化較方便理解，但沒有層次
  - kernel 功能全部混在一起

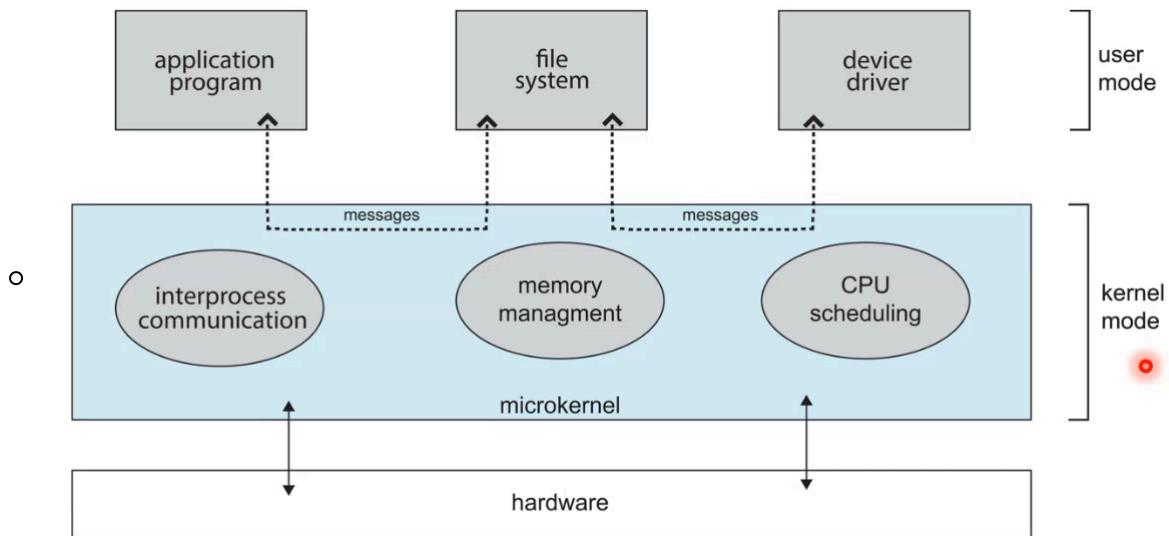


- Layered(有層次) - an abstraction
  - 一層一層包住，裡面最底層，外面最上層
  - 架構較嚴謹
  - 每一層只能使用上一層提供的功能，第二層使用第一層硬體提供的基本功能，第三層使用第二層的功能，以此類推



文

- Microkernel(精簡化) - Mach
  - 只保留最重要的功能在 kernel
  - 其餘功能移到 user mode
  - 優點：
    - 容易擴充，不用寫 kernel mode 的 code，只需要寫 user mode 的 code
    - 較可靠、安全
  - 缺點：
    - user, kernel 需一直切換，導致效率降低



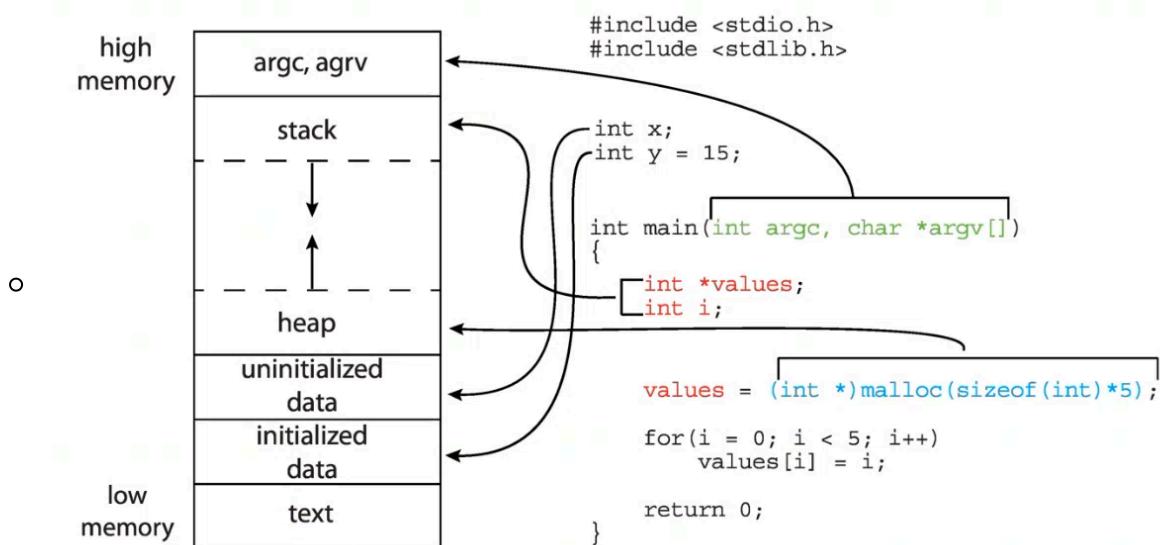
- Hybrid Systems
  - 將上面幾種方式混合，某部分用 micro，某部分用 layered 等等

## Chapter 3 : Processes => 執行中的程式

文

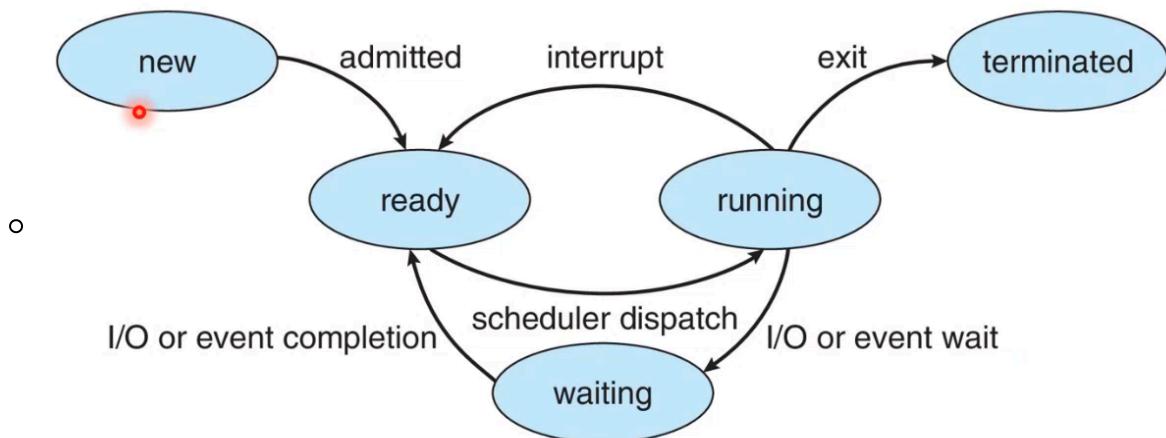
### Process Concept

- Process in Memory
  - stack : 先進後出 => function 概念
  - heap : 先進先出 => 動態配置
  - data : 分配固定的位置給變數等等
  - text : 放 code

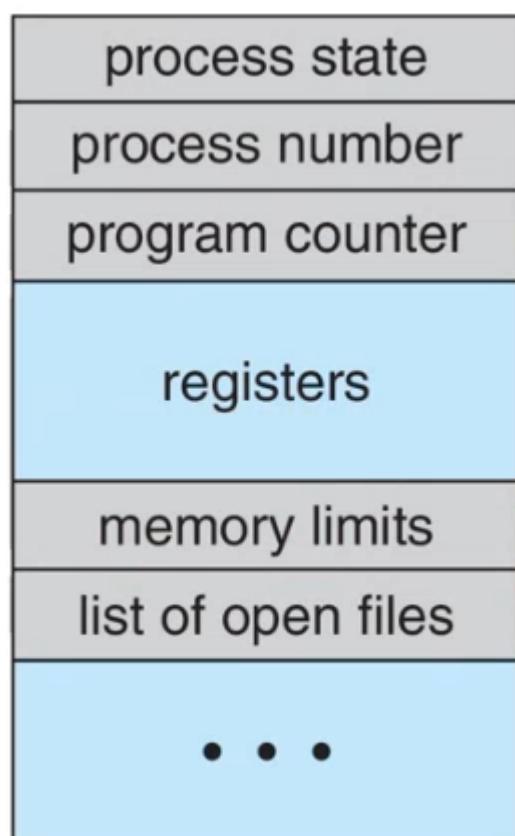


- Process State
  - New : 程式被 load 到 memory，剛開始執行的狀態
  - Ready : 會有好幾個程式等待被執行
  - Running : 執行程式

- Waiting : 遇到事件時的等待，假設有很多程式執行到一半都需要用 I/O，就需要一一等待
- Terminated : 結束



- Process Control Block (PCB)
  - Linux 系統叫 task\_struct
  - 紀錄執行中的各種狀態

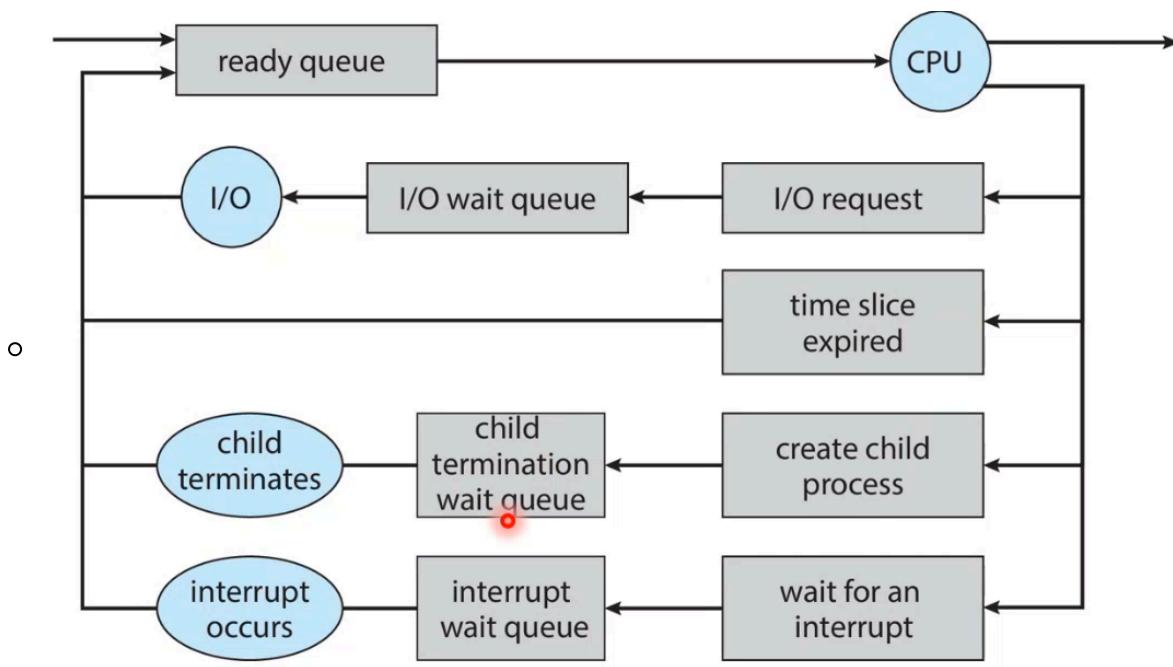


文

## Process Scheduling

- Process Scheduling
  - 目的 : 使 CPU 效率最大化
  - 決定接下來執行哪個程式
  - scheduling queues

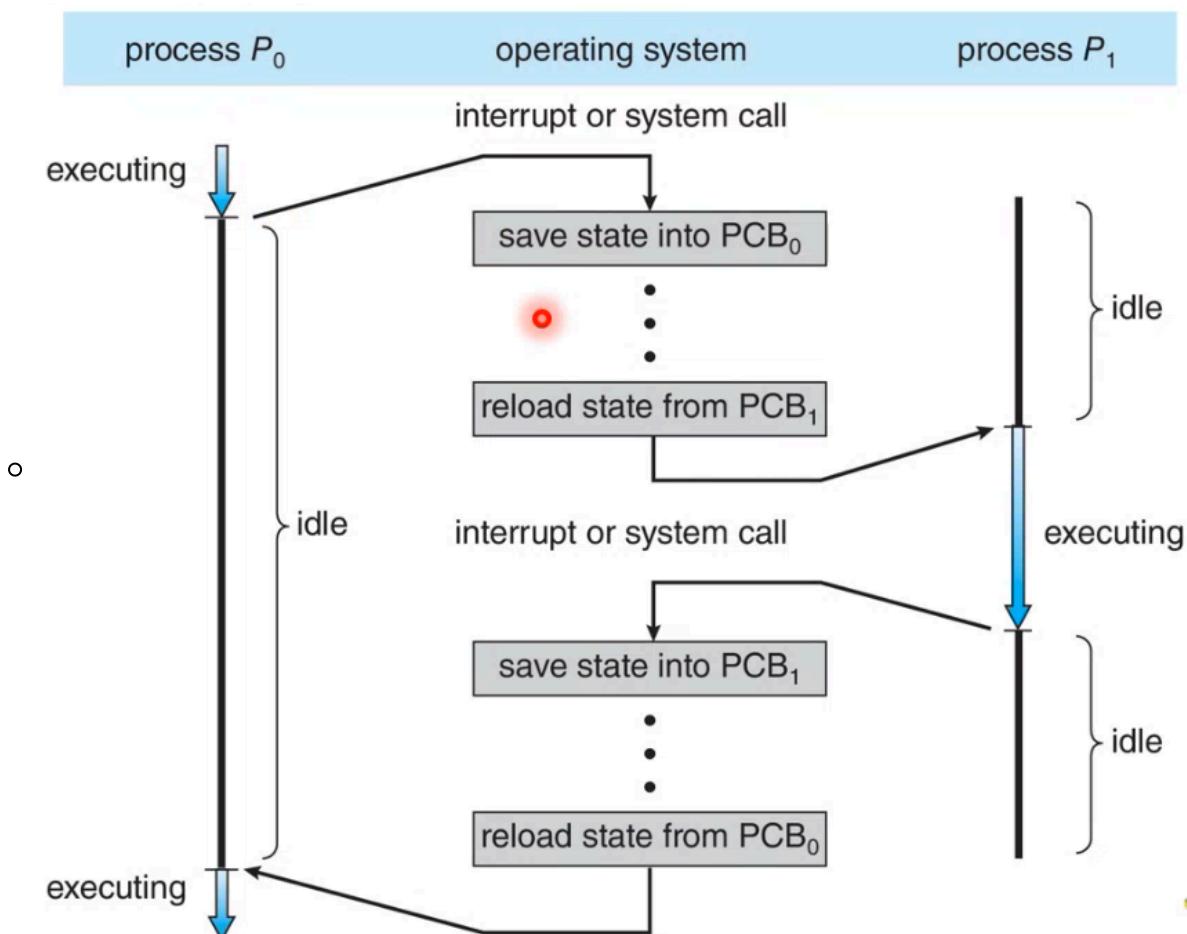
- queue 裡面存放還沒被執行的程式
- Ready queue
- Wait queue



文

- CPU Switch From Process to Process

- 儲存原本程式執行到一半的進度
- 從另一個程式一半的進度繼續執行

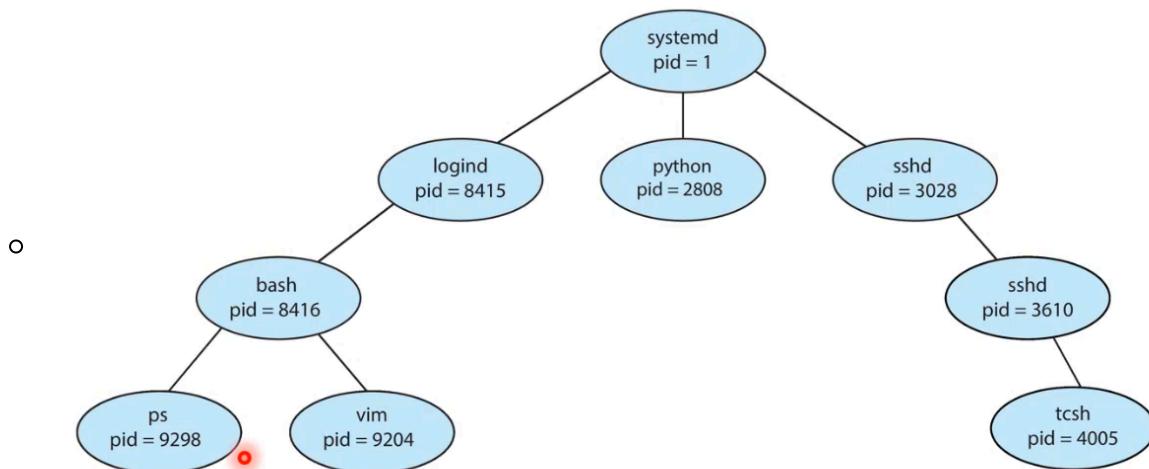


- ISR (Interrupt service routine)

- 中斷服務程序，用於處理系統中斷事件的程序。當系統中發生了某些特定事件（例如硬體中斷、軟體中斷等），CPU會停止正在執行的任務，並轉而執行ISR。
- 硬體中斷**：當硬體設備需要與 CPU 進行通訊時，例如設備完成了數據傳輸，硬體錯誤等，會觸發硬體中斷。系統會執行相應的 ISR 來處理這些中斷事件。
- 軟體中斷**：這些是由 CPU 內部或軟體觸發的中斷。例如，當操作系統需要執行某些臨時操作時，它可以發起軟體中斷，以執行相應的 ISR。
- Interrupt-driven**
  - 處理器不需要不斷地輪詢設備狀態，而是可以響應設備的異步事件。這種方式使得處理器能夠更有效地利用時間，而不必等待某些事件的完成。

## Operations on Processes

- Process creation
  - 程式執行時會有 Parent 跟 children 形成 tree
  - 每個動作都會有一個 pid (process identifier)
  - parent 跟 children 可以共享資源
  - 可以針對執行中的程式做調整，parent, children 同時做或只能一次做一個等等



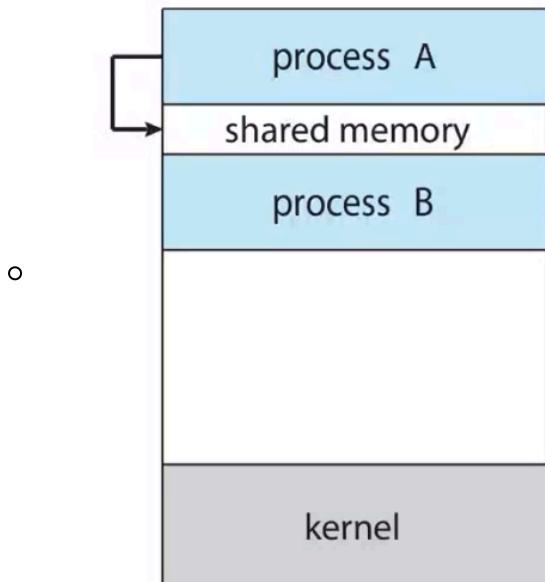
- Process termination
  - 正常結束時，OS 收到 exit() 後就會釋放記憶體空間等等
  - 執行錯誤、佔用太多資源等等發生，OS 就會收到 abort() 結束
  - 如果 parent 結束，children 仍還在執行
    - cascading termination : 強制等待 children 結束，parent 才能結束
    - 如果 parent 沒有寫 wait，OS 會將 children 認為是 **zombie**
    - 如果 parent 結束了，children 還沒結束，OS 會將 children 認為是 **orphan**

## Interprocess Communication

- 有很多的 process 需要執行，所以彼此需要溝通

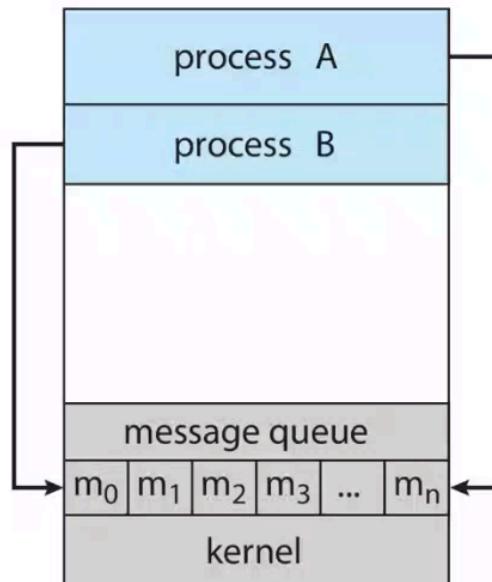
- Producer-Consumer Problem
  - unbounded-buffer
    - 沒有限制的 buffer · Producer 可以一直丟東西進去
  - bounded-buffer
    - 有限制的 buffer · 如果 buffer 滿了 · Producer 就有 wait
- 分為兩種方法

(a) Shared memory



(a)

(b) Message passing



(b)

文A

## IPC in Shared-Memory Systems

- 進程間共享同一塊物理內存區域 · 使得數據的共享和通信變得容易。
- 創建一個共享的內存區域 · 進程將數據寫入共享內存時 · 其他進程可以立即讀取該數據。
- good: 效率&速度
- bad: 數據一致性&並發控制

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}

## IPC in Message-Passing Systems



- 進程之間通過發送和接收消息來進行通信，而不共享內存空間。
- 處理分離，進程彼此之間相對獨立，進程通過發送消息將數據發送到目標進程，目標進程通過接收消息來獲取數據。
- 可同步or非同步
- good:更好的模塊化&抽象
- bad:更消耗資源
- 要考慮的點很多，分很多種
  - Direct or indirect
    - Direct : A 的訊息直接傳到 B 手上
    - indirect : A 把訊息放在郵箱裡，B 再去郵箱把訊息拿出來。
  - ordinary or named pipe:
    - ordinary : 一種在父子進程或者具有共同祖先的兄弟進程之間進行通信的機制，半雙工，意味著數據只能在一個方向上流動。(pipe():一端寫入，一端讀取)
    - named pipe : 也稱FIFO，是一種通用的進程間通信機制，可在無關的進程之間使用，通常通過在文件系統中創建一個特殊的文件來實現。(mkfifo() 或 mknod()，然後像標準文件一樣被打開、讀取和寫入。)
  - Socket
    - 一種通信機制，可於不同主機，不同進程，不同協定間通信，全雙工
  - Synchronization
    - Blocking :
      - send : A 送完訊息需確認 B 收到
      - receive : B 一直守在信箱前面等
    - Non-blocking :

- send : A 送完訊息直接走
- receive : B 確認過沒信就走
- Buffering
  - buffer 要多大

## Communication in Client-Server Systems

- Sockets
- Remote Procedure Calls(RPC)

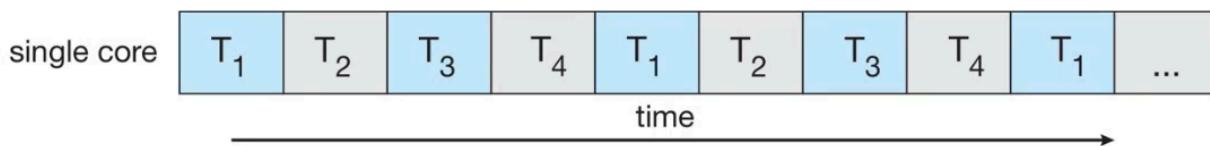
## Chapter 4 : Threads & Concurrency

- User thread : 用程序或用戶空間的程式庫所創建和管理的線程，僅存在於應用程序的地址空間。
  - good 創建和調度開銷較小，可以根據需要自行管理線程，實現更高的彈性和自主性
- Kernel thread : kernel所創建和管理的線程，存在於操作系統的內核空間中。
  - good : 可利用如調度、同步和內存管理，提高系統的可靠性和穩定性，可跨平台。

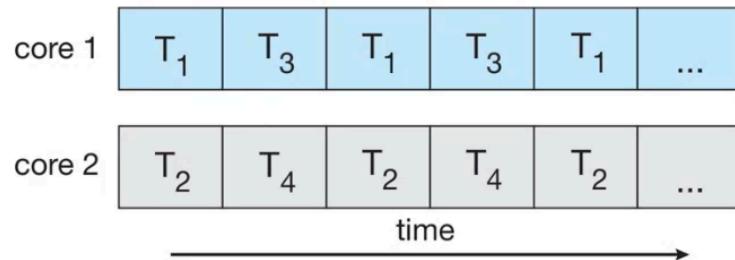
文

## Multicore Programming

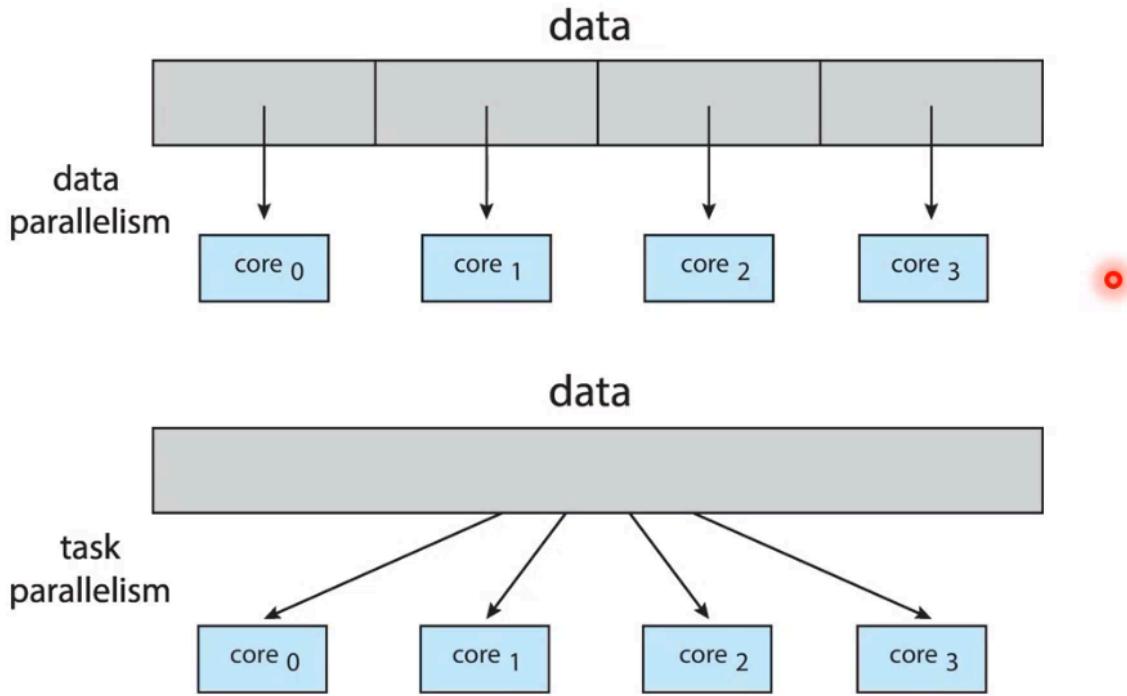
- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



## Types of parallelism



- Data parallelism : 將一個 data 切割成多個 data 執行，每個 core 執行同樣的任務
- Task parallelism : 將一個 data 切割成多個 task 執行，每個 core 執行不同的任務

文

## Amdahl's Law : 評估多核的效率

S 表示必須依序完成的部分(不能同時進行) · N 表示有幾個 cores

若 N 為無窮大，則能夠快  $1/S$  倍

S 為決定效率最重要的部分

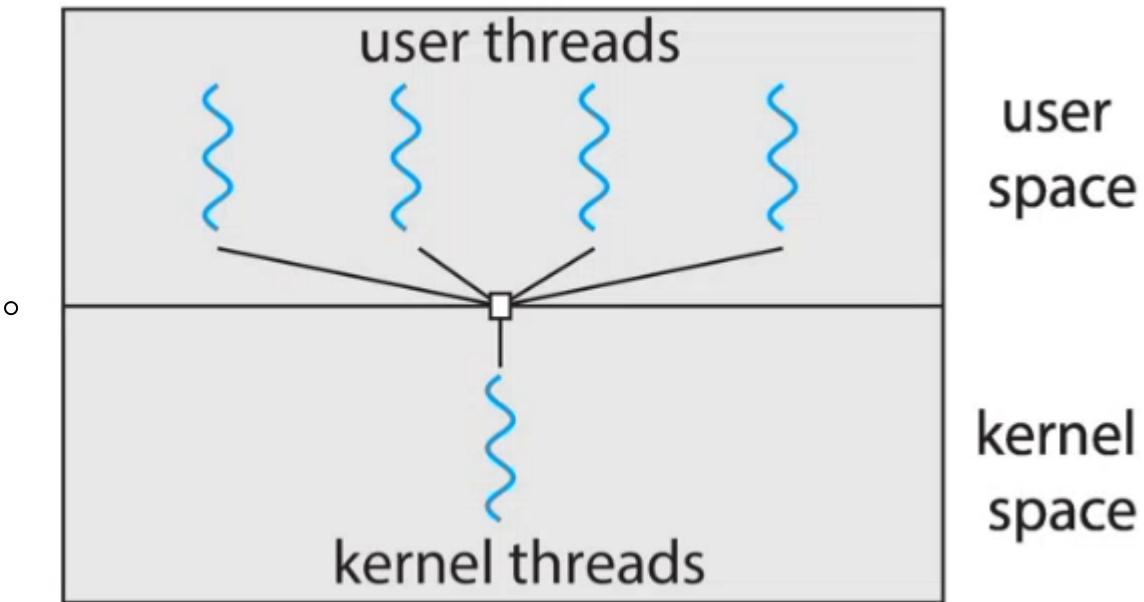
**S is serial portion, N processing cores**

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

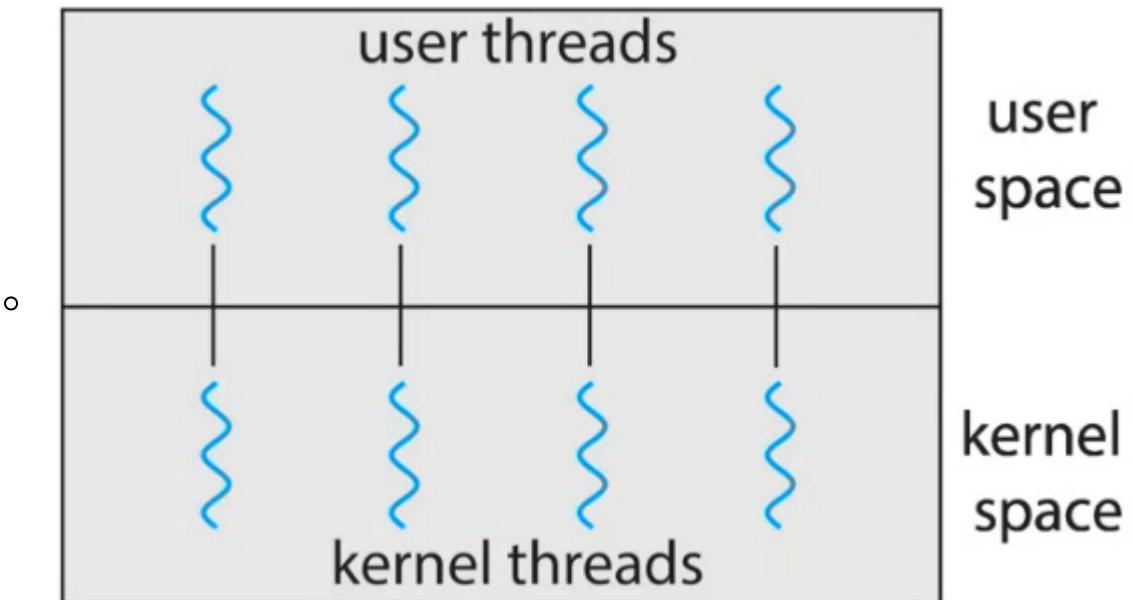
- Q : 假設一個應用程式有 75% parallel and 25% serial，並且有 2 cores
- A :  $1/(0.25 + (0.75 / 2)) = 1.6$  倍，即 2 cores 比 1 core 快 1.6 倍

## Multithreading Models

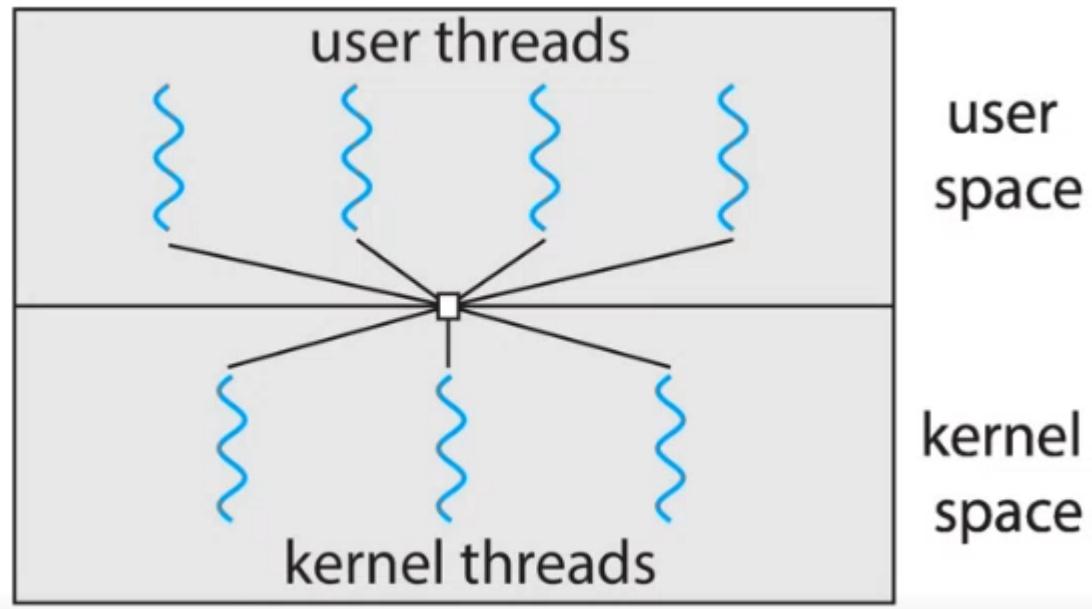
- Many-to-One
  - 較早期的作法，因為只有單核，所以 user threads 必須輪流進行



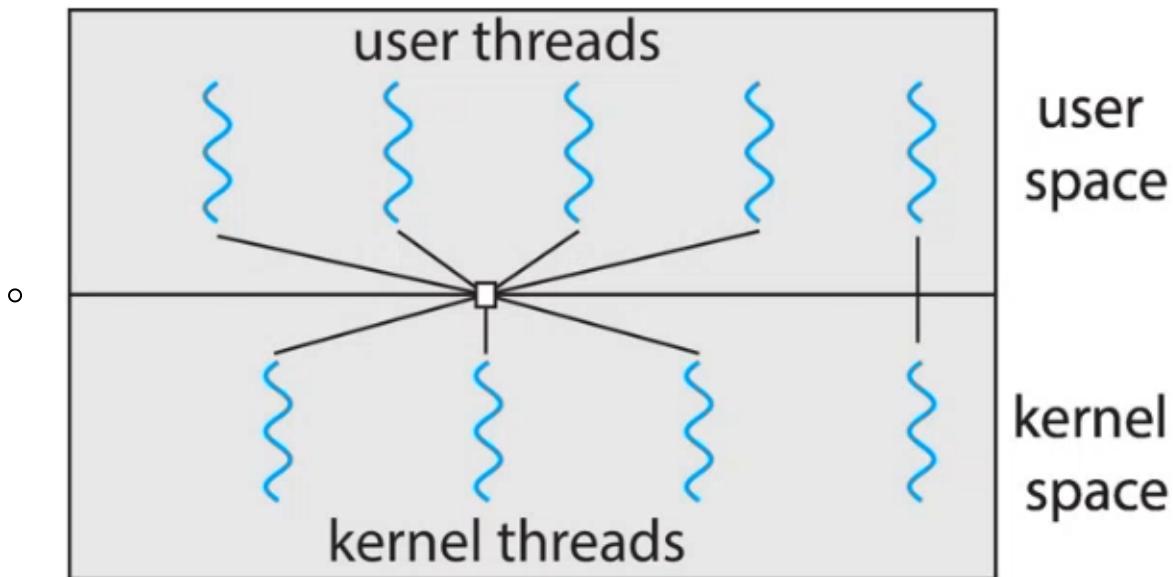
- One-to-One
  - 能夠平行處理，但缺點是效率有所限制 => user threads 不能超過核心數



- Many-to-Many
  - 較複雜，需要進行處理



- Two-level Model
  - 較常見，融合 One-to-One, Many-to-Many



文

## Thread Libraries

提供一個 API 控制哪些程式的 Function 要分配給哪些 threads 執行

- Pthreads : POSIX API · 主要用在 UNIX 系統
- Windows 也有提供相關 API · 只是沒有固定名稱
- Java Threads

## Operating System Examples

- Windwos Threads
  - kernel space
    - ETHREAD : 指向 KTHREAD

- KTHREAD : 指向 TEB
  - user space
  - TEB
- Linux Threads
  - Linux 叫 tasks 而不是叫 threads

# Chapter 5 : CPU Scheduling

CPU 空閒下來時，選一個 process 執行

## Scheduling Criteria

- CPU utilization : 讓 CPU 越忙越好
- Throughput : 單位時間內完成任務的數量越多越好
- Turnaround time : 執行時間越短越好
- Waiting time : 等待時間越短越好
- Response time : 回應時間越短越好

文  
A

## Scheduling Algorithms

- FCFS (First Come First Served)
  - 特色：按照作業到達的順序來進行排程，先到達的作業先被執行。
  - 方法：將作業依照到達順序放入佇列中，依序執行佇列中的作業。
  - 優點：簡單易懂，公平性高。
  - 缺點：可能產生饑餓現象（一個長時間執行的作業可能導致其他作業長時間等待）。

## 範例 1

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

## 範例 2

Suppose that the processes arrive in the order:

$P_2, P_3, P_1$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$ 
  - Much better than previous case

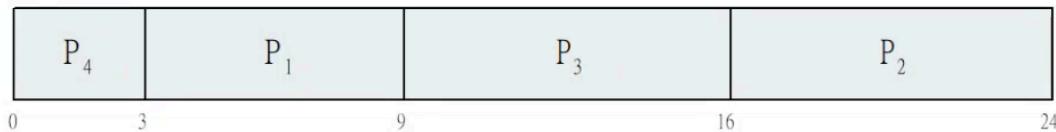
範例 2 比範例 1 更好，因為順序的原因導致 waiting time 較少

- SJF (Shortest-Job-First)
  - 特色：選擇最短的作業優先執行，以減少平均等待時間。
  - 方法：將所有待執行的作業按照執行所需的時間長短排序，每次選擇執行執行時間最短的作業。
  - 優點：平均等待時間短，效率高。
  - 缺點：可能出現饑餓現象（執行時間長的作業可能一直被推遲）。

## 範例 1

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

## 範例 2 (preemptive)

如果有新的 process 進來會重新進行評估，讓最短的插隊

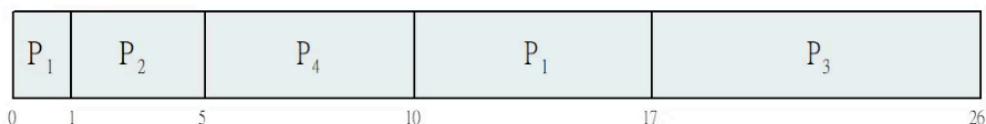
waiting time : 每個 process 用全部跑完的時間 - 抵達的時間 - 執行時間 舉例

process 1 : 從 10 開始全部跑完 - 最一開始執行到 1 被中斷

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive SJF Gantt Chart



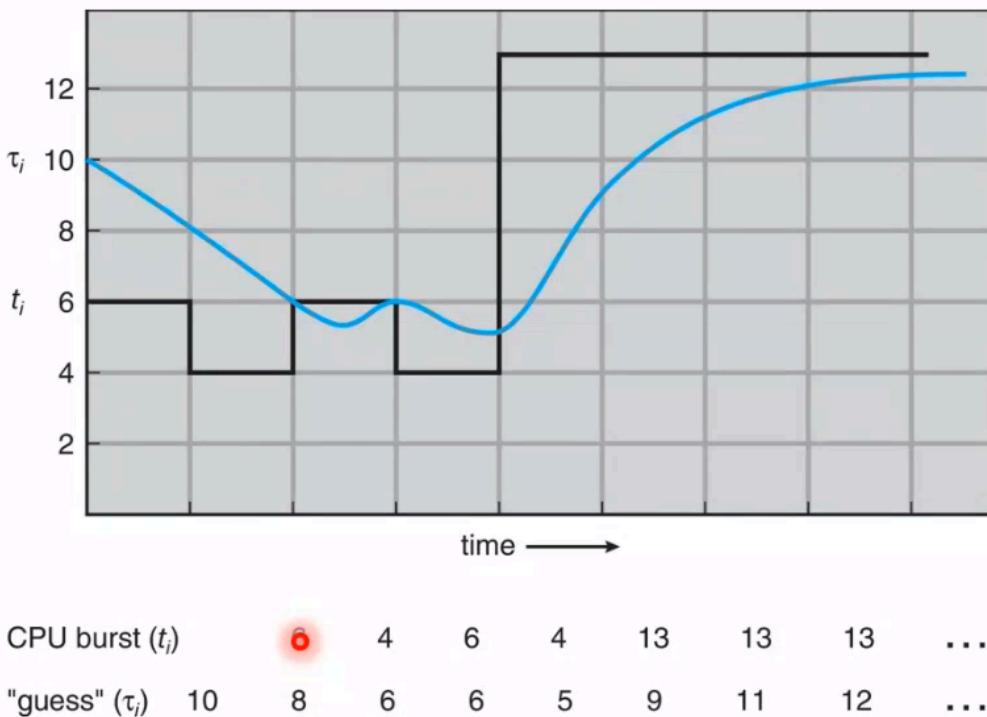
- Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$

- 問題：我們無法知道下個 process 的時間多少 => 只能用預測 (exponential averaging)
  - 從上一個預測的值或上一個實際的值做預測， $\alpha = 1/2$  為兩者以一樣的權重預測(0~1)

- 1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define :  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$

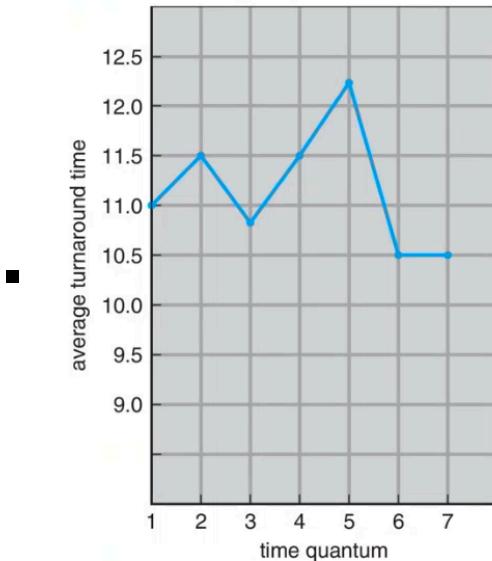
Commonly,  $\alpha$  set to  $1/2$

範例



文

- RR (Round Robin)
  - 一個時間單位叫做 time quantum
  - 主要在減少 response time 的時間
  - 特色：按照順序分配固定時間片的方式執行作業，當一個作業的時間片用完後，執行下一個作業。
  - 方法：將所有待執行的作業按照到達順序放入佇列中，每次執行一個作業一個固定的時間片，然後執行下一個作業。
  - 優點：公平性高，適合時間片短的情況。
  - 缺點：可能產生上下文切換過多的情況，效率下降。
  - Performance
    - q large => FIFO
    - q small => 太頻繁切換導致 overhead 太大
    - q 最好大過於 80% 的 process



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

\* Rule of thumb:  
80% of CPU bursts  
should be shorter than  $q$

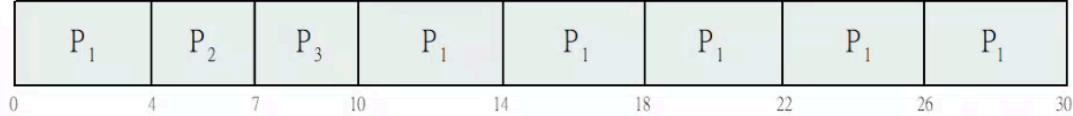
### 範例

Quantum = 4

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

文

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
  - $q$  usually 10 milliseconds to 100 milliseconds
  - Context switch < 10 microseconds

- Priority Scheduling
  - 特色：根據作業的優先級來進行排程，優先級高的作業先被執行。
  - 方法：每個作業都有一個優先級，作業被選擇執行的順序取決於其優先級。
  - 優點：能夠根據作業的重要性來進行排程，適用於多種場景。
  - 缺點：可能出現優先級反轉的情況（低優先級作業長時間等待高優先級作業執行完畢）。
  - Preemptive
  - Nonpreemptive
  - Problem : Starvation (priority 小到一直有權限比他大的插隊)
    - Solution : Aging (動態提升 priority => 等得越久，priority 越高)

## 範例 1

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

Priority scheduling Gantt Chart



Average waiting time = 8.2

## 範例 2

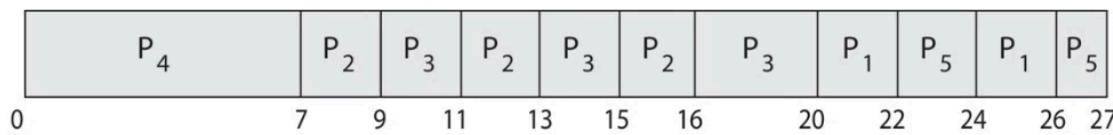
priority 值相同的情況使用 Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

Run the process with the highest priority

Processes with the same priority run round-robin

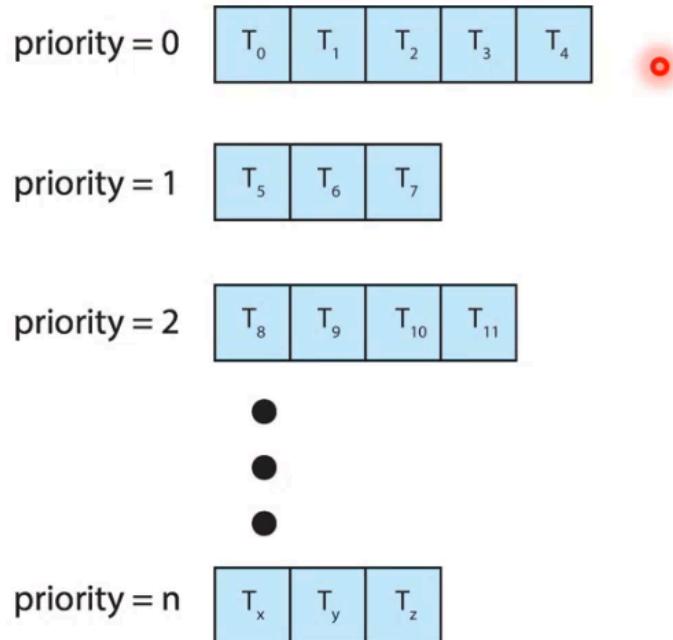
Gantt Chart with time quantum = 2



- Multilevel Queue (Priority Scheduling 延伸)

With priority scheduling, have separate queues for each priority

Schedule the process in the highest-priority queue!



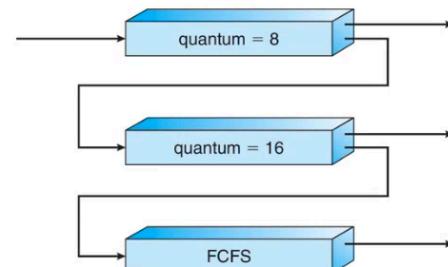
- Multilevel Feedback Queue

- 動態調整 priority

範例

- Three queues:

- Q<sub>0</sub> – RR with time quantum 8 milliseconds
    - Q<sub>1</sub> – RR time quantum 16 milliseconds
    - Q<sub>2</sub> – FCFS



- Scheduling

- A new process enters queue Q<sub>0</sub> which is served in RR
      - ▶ When it gains CPU, the process receives 8 milliseconds
      - ▶ If it does not finish in 8 milliseconds, the process is moved to queue Q<sub>1</sub>
    - At Q<sub>1</sub> job is again served in RR and receives 16 additional milliseconds
      - ▶ If it still does not complete, it is preempted and moved to queue Q<sub>2</sub>

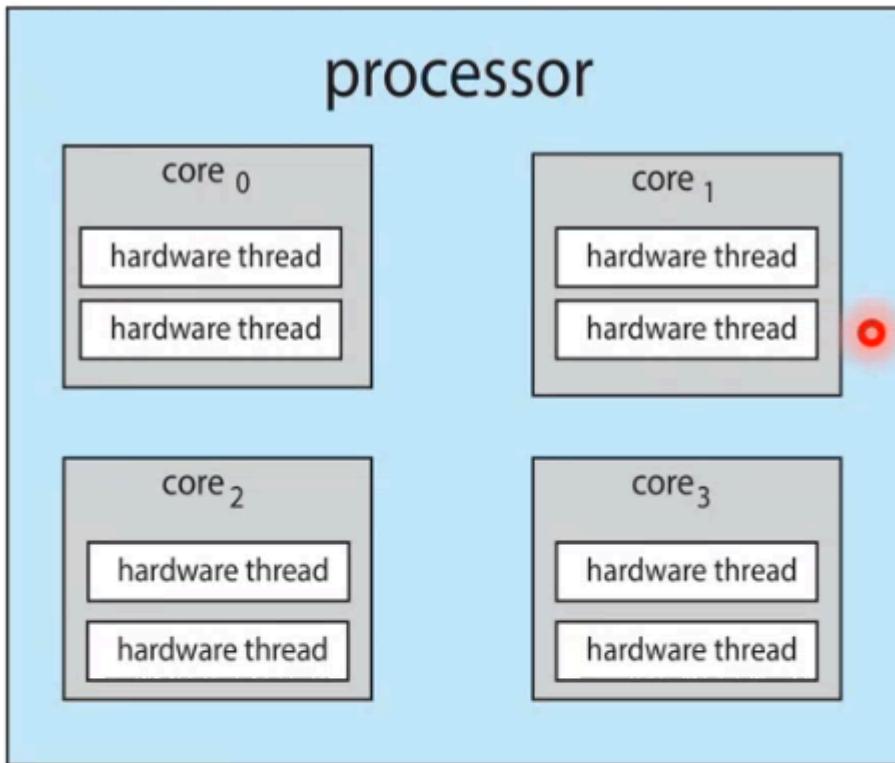


## Thread Scheduling

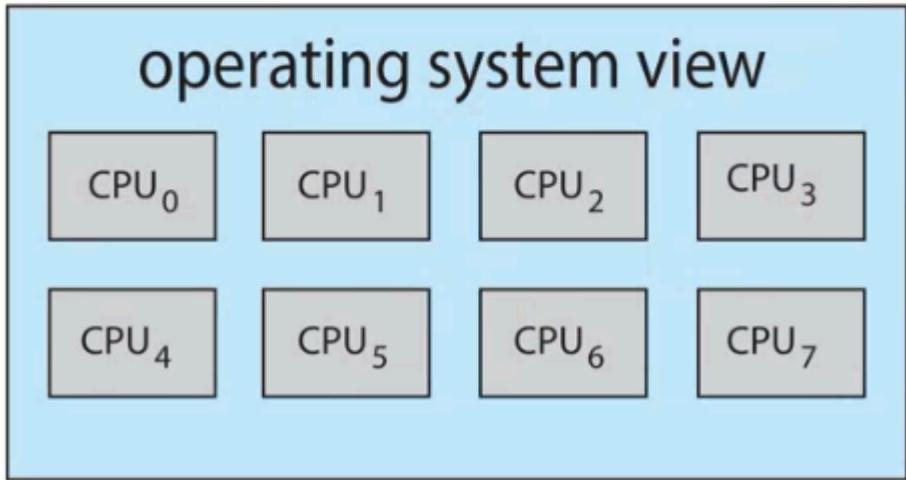
- process-contention scope (PCS) : 同一個 process 內的 thread 競爭
- system-contention scope (SCS) : 不同 process 的 thread 競爭

# Multi-Processor Scheduling

一個 CPU 有 8 個核心，每個核心又有兩個 hardware threads

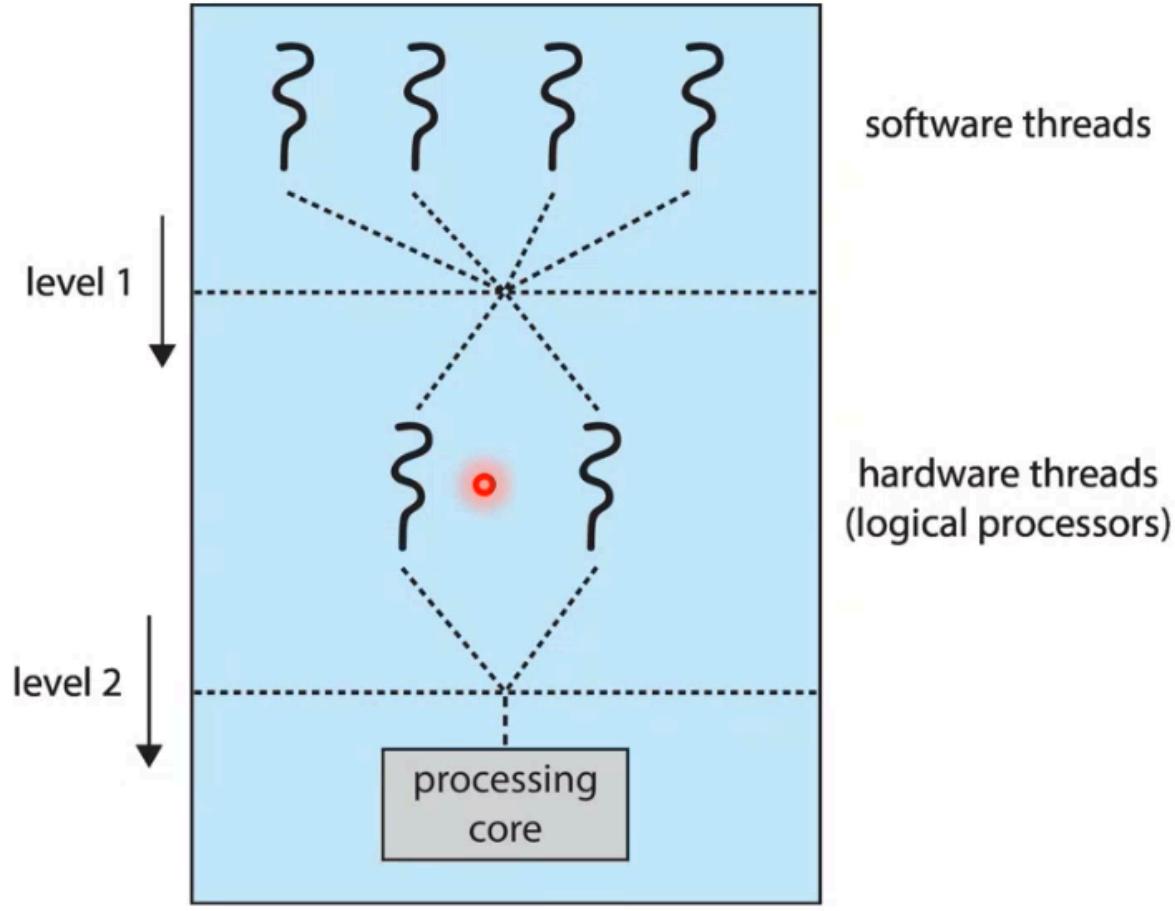


文



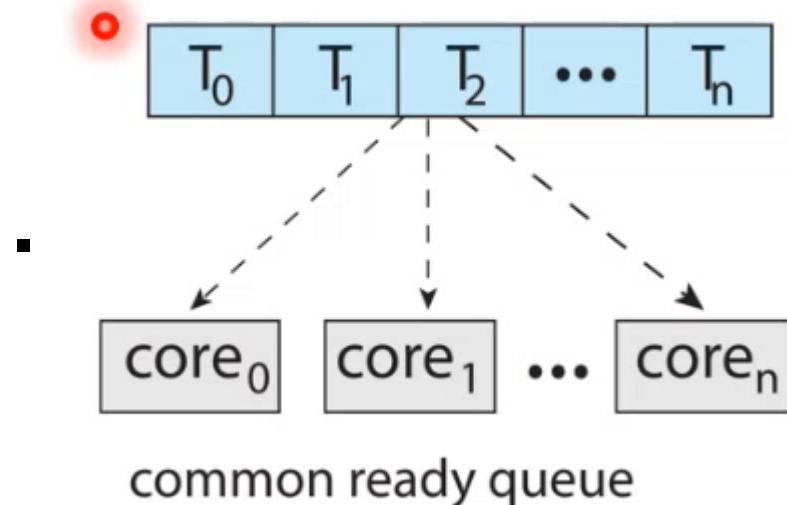
第一層為 hardware threads 要如何分配執行 software threads

第二層為 processing core 要如何分配執行 hardware threads

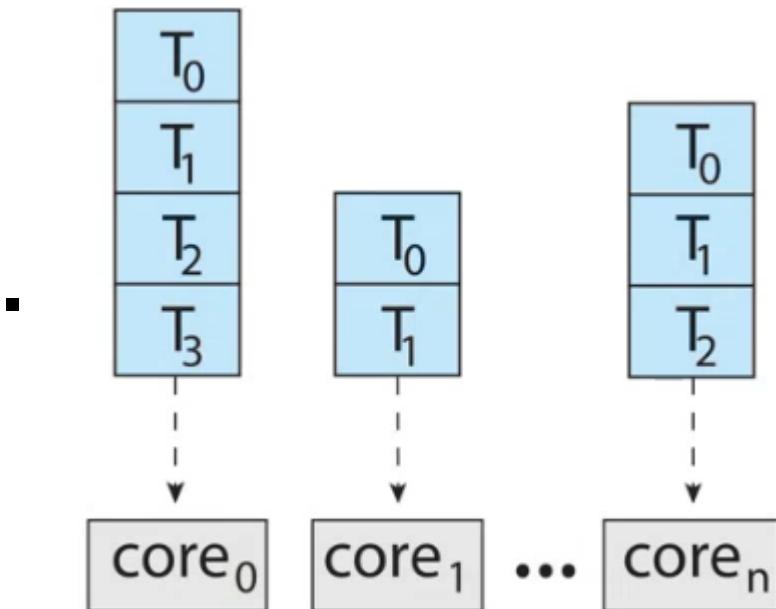


文

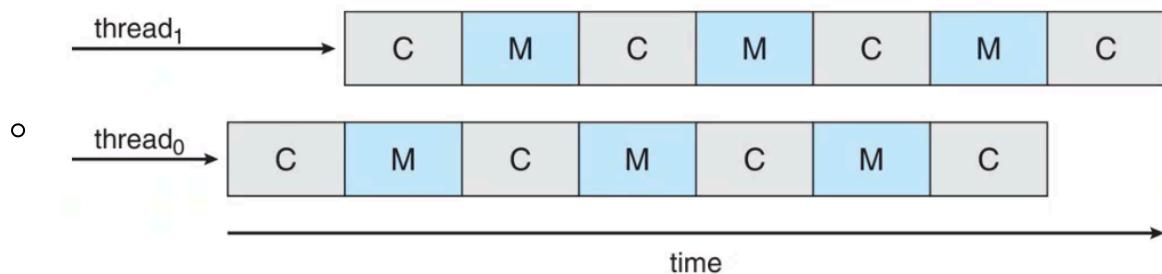
- Symmetric multiprocessing (SMP)
  - common ready queue



- per-core queues



- memory stall



文A

## Load Balancing : 如何讓每個 CPU 都很忙

- Push migration : 很忙的 core 丟任務給空閒的 core
- Pull migration : 空閒的 core 去很忙的 core 抓任務執行

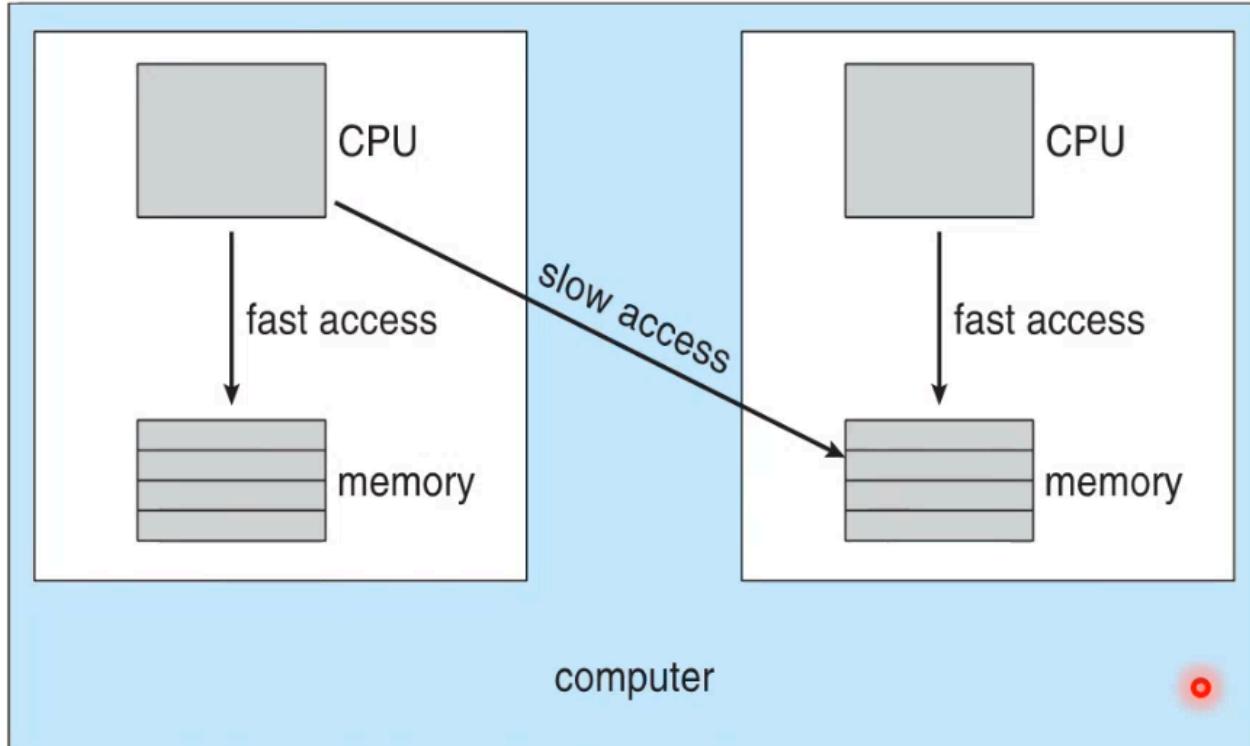
## Processor Affinity

舉例 : core1 執行到一半的任務丟給 core2 執行 , 但是 core1 已經處理完前面的 data 存放在 cache 了 , 這樣反而會讓 core2 重頭執行 , 如果全部 data 丟過去會消耗更多資源

- Soft affinity : 盡量讓同個任務在同個 core 執行 , 但是不保證 , load 時間太久還是會移走
- Hard affinity : 強制讓同個任務在同個 core 執行 , 不會被移走

## NUMA and CPU Scheduling

用到其他核心的 memory



## Algorithm Evaluation

- Deterministic modeling
  - 假設固定的 workload 用不同的演算法測試計算 waiting time, response time...
  - 優點：簡單
  - 缺點：因為指針對預先固定的 Workload，所以有可能在其他的 workload 上不適用
- Queueing Models
  - 不固定 workload，用機率方式描述 workload
  - 優點：較有系統化的測試，預測各種 workload 進行評估
  - 缺點：實際上的案例跟機率計算的案例不一定符合
- Simulations
  - 蒐集各種真實數據，模擬實際情況
  - 優點：相較於前兩種方法較準確
  - 缺點：麻煩，準確度也僅限蒐集到的資料
- Implementation
  - 不模擬，實際操作
  - 優點：最為準確，能夠正確評估各種 process 的效率
  - 缺點：成本高、風險高(如果程式寫錯，會導致整台電腦 error)

# Chapter 6 : Synchronization Tools

## Background

- 資料不一致：如果一個 process update data, 另一個 process read data

## The Critical-Section Problem

- 設計一個 protocol 當一個 process 在使用 data 時，其他的 process 要先發出 entry section 訪問存取權，當 process 使用完 data 時要發出 exit section 通知其他 process
- 為了實作 protocol 需要設計一些要求
  - Mutual Exclusion：當一個 process 在使用 data 時，其他的 process 不能訪問 data
  - Progress：沒有 process 使用 data 時，要在有限的時間內決定下一個 process
  - Bounded Waiting：等待中的 process 有上限，不能等太久

文

## Interrupt-based Solution

- 當 Entry section：把 interrupts 關閉，別人不能發出 interrupts
- 當 Exit section：把 interrupts 打開，別人能夠發出 interrupts
- 可能遇到的問題
  - 成功 Entry section 的 process 執行太久，其他 process 不能用
  - 等待的 process 一直被插隊，導致一直不能使用
  - 如果有兩個 CPU 的話，會有更多的問題

## Software Solution 1

- 從 code 上做檢查
- 兩個假設簡化問題
  - 假設只有兩個 process
  - 假設執行過程不會被 interrupt

```

while (true)
{
    turn = i;
    while (turn == j)
        ;
    /* critical section */
    turn = j;
    /* remainder section */
}

```

```

while (true)
{
    turn = j;
    while (turn == i)
        ;
    /* critical section */
    turn = i;
    /* remainder section */
}

```

- 問題
  - 可能不符合 Progress · 假設 j 速度很快 · i 就一直輪不到他執行



## Peterson's Solution

- 兩個假設簡化問題
  - 假設只有兩個 process
  - 假設執行過程不會被 interrupt
- 多了 flag 表示 process 是否已經準備好 entry section

```

while (true)                                while (true)
{
    flag[i] = true;                         {
    turn = j;                               flag[j] = true;
    while (flag[j] && turn                  turn = i;
        == j)                                while (flag[i] && turn
        ;                                    == i)
    ;                                         ;
    /* critical section */                  /* critical section */
    flag[i] = false;                        flag[j] = false;
    /* remainder section */                /* remainder section */
}

```

- 在現代架構中可能無法 work
  - processors or compilers 可能會因為效率問題 reorder

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100

### Memory Barrier (硬體 support)

- data 改變時會立即告訴 CPU
- 確保 load and store 能夠完全做完，即使 reorder 也無妨

- Thread 1 now performs
 

```
while (!flag)
    memory_barrier();
    print x
```
- 
- Thread 2 now performs
 

```
x = 100;
memory_barrier();
flag = true
```

## Mutex Locks (spinlock)

- 兩個模式 : acquire, release
- 兩個模式必須 atomic (不會被 interrupt)
- acquire : 進去前取得許可
- release : 出來前釋放資源
- 問題
  - busy waiting : 要不斷在 while loop 不斷 check
    - 如果在雙核心以上，一個 core 在 busy waiting，其他的 core 還可以動
    - 但在單核心如果沒有適當機制切換的話就會卡住

## ★ Semaphores

1. 計數機制：信號量的計數可以大於一，允許多個線程或進程同時訪問共享資源，但訪問數受到計數限制。互斥鎖通常只允許一個線程或進程同時獲取鎖。
2. 使用方法：互斥鎖主要用於互斥，即一次只有一個線程可以訪問代碼的臨界區段，以防止競爭條件。信號量則可以用於更廣泛的同步任務，包括控制對多個資源實例的訪問、在線程之間發出信號以及同步生產者-消費者方案。
3. 操作：信號量支援兩個主要操作：‘等待’（也稱為 ‘P’ 或 ‘down’），它減少信號量計數並阻塞調用者，如果計數變為零；‘信號’（也稱為 ‘V’ 或 ‘up’），它增加信號量計數並喚醒被阻塞的進程（如果有）。互斥鎖通常只支援‘鎖定’（獲取）和‘解鎖’（釋放）操作。
4. 資源管理：信號量可以用於管理除互斥以外的資源，例如控制對資源池的訪問或限制訪問特定資源的線程數量。互斥鎖主要用於臨界區段的保護，不支援內置的資源計數。

- 兩個模式 : wait, signal
- 兩個模式必須 atomic (不會被 interrupt)

- wait : check S 的值是否大於 0，如果小於 0 就 wait，會把 S-
- signal : 把 S++
- Binary semaphore : 值在 0, 1 之間切換，如同 Mutex Locks
- Counting semaphore : 可以計算

## Mutex Locks vs Semaphores

面向	Semaphores	Mutex Locks
定義	一個整數變量，允許多個線程同時訪問一個資源。	一個二進制變量，一次只有一個線程可以訪問一個。
所有權	不被任何特定線程擁有。	由獲取該鎖的線程所有。
複雜性	更加複雜。	較少複雜。
效率	更加高效，因為多個線程可以同時訪問一個資源。	通常效率較低。
等待操作	減少信號量的值。如果值為非負數，線程繼續；否則它將被阻塞，直到信號量變為非負。	如果互斥鎖當前被另一個線程鎖則阻塞該線程。
信號操作(V)/解鎖	增加信號量的值。如果有線程被阻塞在信號量上，其中一個將被解除阻塞。	釋放互斥鎖，允許另一個線程獲得。

## Semaphores Implementation with no Busy waiting

- use waiting queue
- block : if a process waiting, let it go the block
- wakeup : if a process signal, from block choose one wakeup

```

wait(semaphore *S) {

    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

•

signal(semaphore *S) {

    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

```

文

## Problems with Semaphores

- 先 signal 後 wait
- 只有 wait 沒有 signal

## Monitors

- 類似於 class 的方式，能夠更輕鬆實作
- 自動幫 process 增加保護機制(signal, wait...)

# Chapter 7 : Synchronization Examples

## Bounded-Buffer Problem

- 共用的有限 buffer 問題
- mutex : 控制同一時間只有一個 process 能夠 access，初始值為 1

- full : 目前有多少個空間被占用，每次 +1，初始值為 0
- empty : 有 n 個空間，每次 -1，初始值為 n

## Readers-Writers Problem

- 資料不同步問題
- rw\_mutex : 控制讀或寫，初始值為 1
- mutex : 控制 buffer 存取，初始值為 1
- read\_count : 用整數紀錄目前 read 的 process 數量，初始值為 0

## Readers-Writers Problem Variations

- Problem
  - First reader-writer
    - 如果太多 read，導致沒有辦法 write
- Solution
  - Second reader-writer
    - 如果 write 進 wait，後面 read 要等 write 結束
    - 但可能會變成太多 write 導致不能 read

文

## Dining-Philosophers Problem

- 假想問題
  - 5 個哲學家、5 個碗、5 支筷子，他們必須要 share 筷子



- Solution
  - 如果某個哲學家要用餐的話，要先問過左右邊的人
    - 假如左上跟下面一組，右上跟左下一組，他們輪流吃飯，會導致右下角一直吃不到
    - 改成如果要用餐時，左右邊有人吃飯就設為 hungry 並等待。
    - 吃完的人確認左右邊是否有人 hungry，有的話 signal 他
    - 還是有可能會有人沒吃到的問題，並沒有完全解決

## Kernel Synchronization - Windows

- spinlocks
- dispatcher objects (類似 mutex, semaphore)
  - Events (類似 condition variable)
  - signaled-state
  - non-signaled-state

## Linux Synchronization

- Atomic integers
- Mutex locks
- Spinlocks, Semaphores
- Reader-writer versions of both

## Alternative Approaches

- Transactional Memory
  - 硬體 support
- OpenMP
  - 能夠告訴編譯器需要做一些特殊處理
- Functional Programming Languages
  - 必須透過 function 才能執行
  - 假設一個變數，一定需要 function 才能夠變更他的 value

## Chapter 8 : Deadlocks

- Deadlock with Semaphores

## ■ Data:

- A semaphore **s1** initialized to 1
- A semaphore **s2** initialized to 1

## ■ Two processes P1 and P2

### ■ P1 :

◦

**wait(s1)**

**wait(s2)**

### ■ P2 :

文A

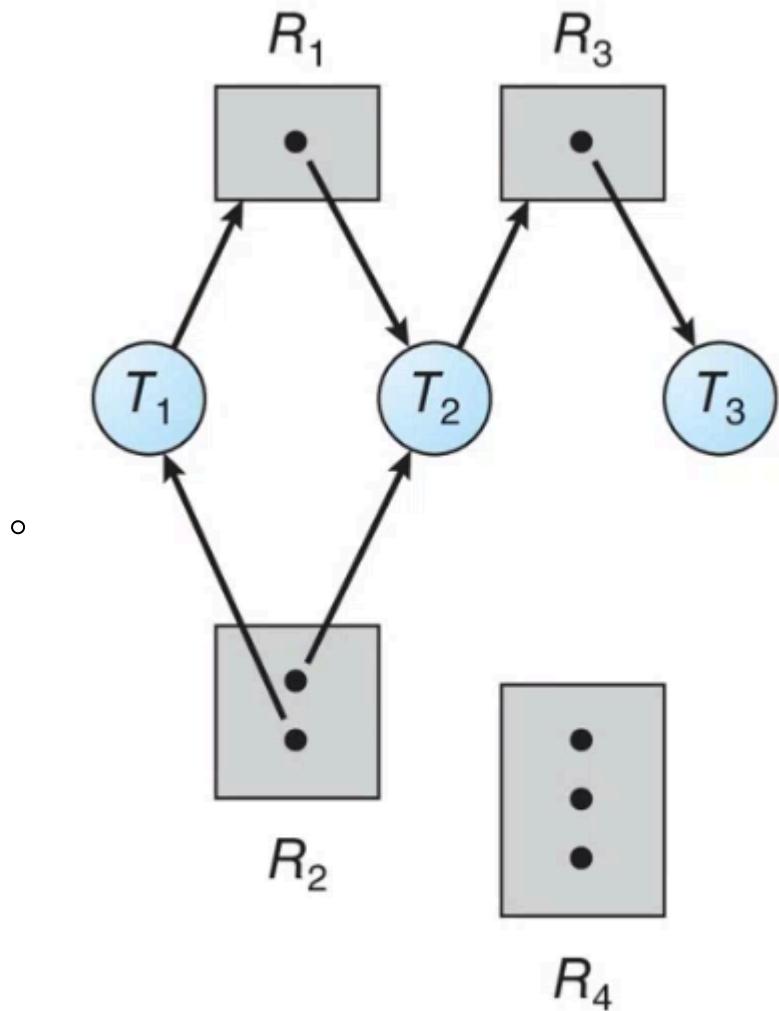
**wait(s2)**

**wait(s1)**

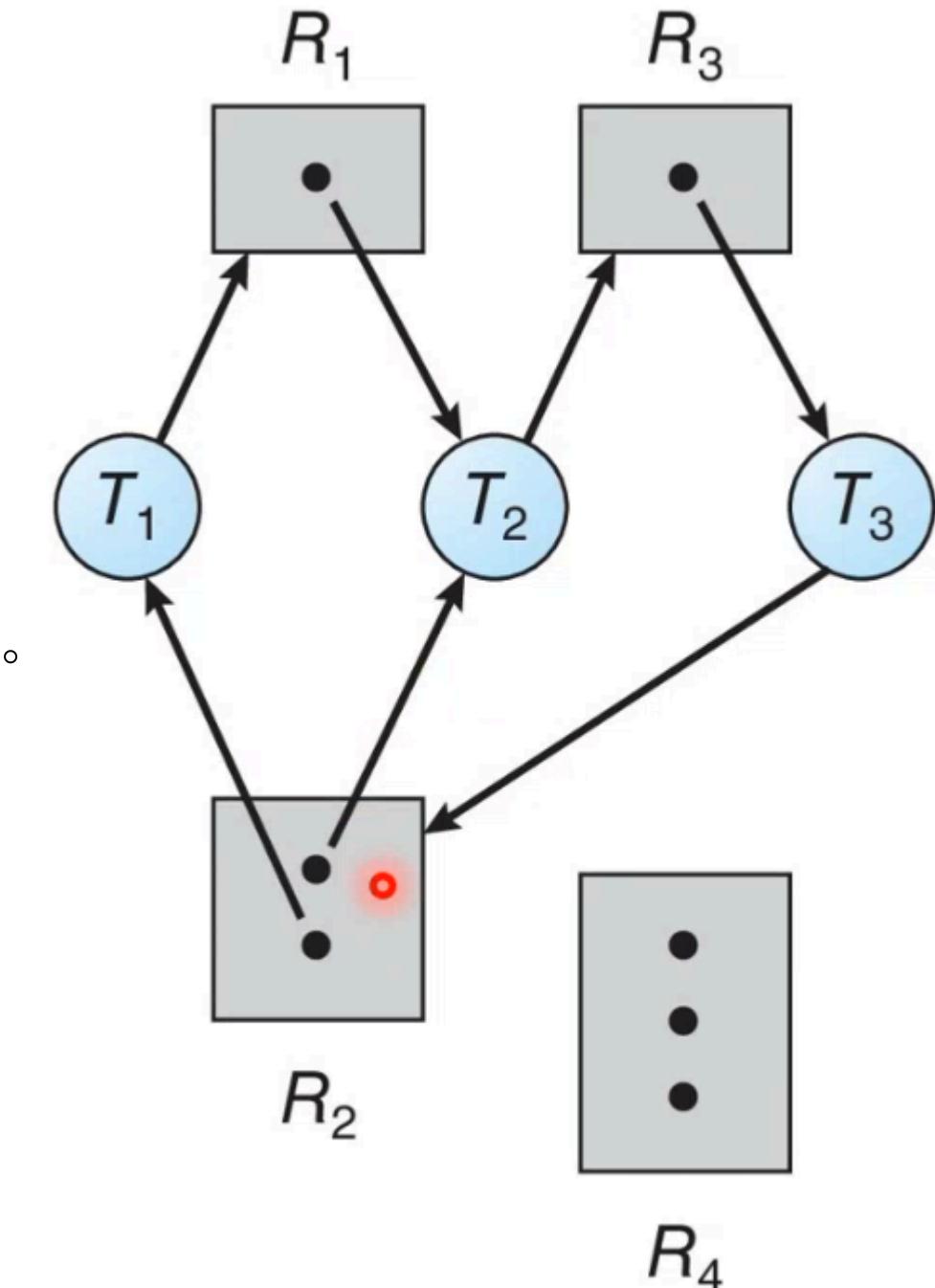
- Deadlock Characterization
  - Mutual exclusion : 同一個時間只能有一個 process 使用
  - Hold and wait : 一個 process hold 一個資源，並且再 wait 其他 process
  - No preemption : 除非主動釋放，否則其他 process 不能動用資源
  - Circular wait : 多個 process 都在 hold and wait

## Resource-Allocation Graph

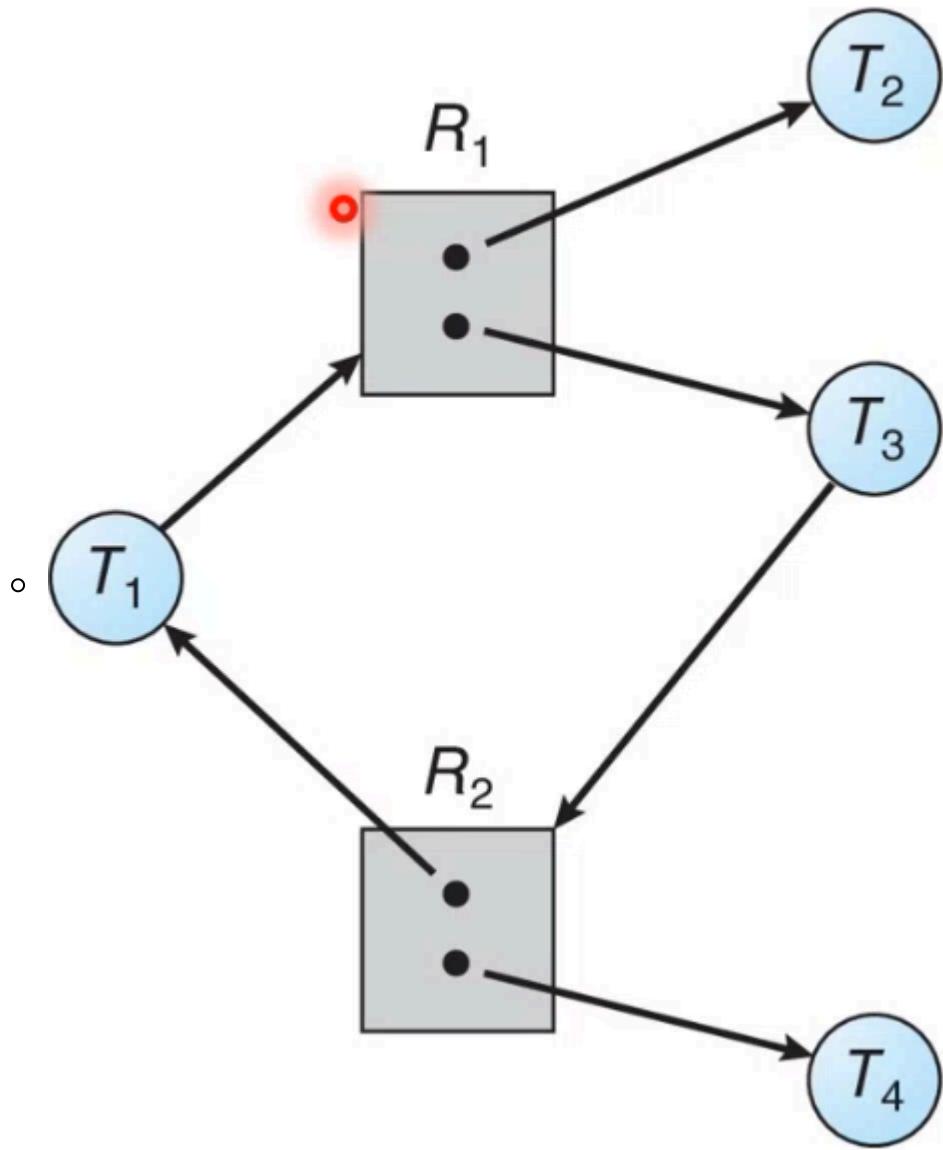
- V(vertices) :
  - P : 多個 process
  - R : 多個 resource
- E(edges) :
  - request edge :  $P_i \rightarrow R_j$
  - assignment edge :  $R_j \rightarrow P_i$
- Example 1 :



- o if  $T_3 \rightarrow R_2$  will appear deadlock
- o cuz  $R_2$  only 2 resource, already allocate to  $T_1$  and  $T_2$ . And  $T_1$  wait  $T_2$ ,  $T_2$  wait  $T_3$ .
- o so  $T_3$  has to wait  $T_1$  or  $T_2$  release resource



- Example 2 :
  - Graph with a Cycle But no Deadlock
  - cuz  $T_1$  only wait  $T_2$  release resource, don't wait  $T_3$ . And  $T_3$  both



文A

- Basic Facts
  - if graph no cycles => no deadlock
  - if graph have a cycle
    - if only one instance => deadlock
    - if several instances => **possibility** of deadlock

## Methods for Handling Deadlocks

- 如何確保系統不會 deadlock ? Deadlock prevention : 盡可能讓上面的四個形成 deadlock 必要條件不成立 Deadlock avoidance : 監控 process 運作時所需要的 resource 並適時調整
- 等 deadlock 再解決
- 不管 deadlock => 目前大部分作業系統採用這個，遇到就自行重新開機

### Deadlock Prevention

- Mutual exclusion (較難解決)

- 如果請求全部都是 read，因為不會更動到 data，所以可以讓 process 同時取用
- Hold and wait (較難解決)
  - 在程式最一開始就配置好所有 resource
    - 可能在過程中還是要 share data，無法在一開始就全部分配好
- No preemption (較難解決)
  - 如果無法立即拿到資源的話，就釋放手中的資源
    - 資源使用效率低
- Circular wait
  - 照順序拿資源

## Deadlock Avoidance

- 需要事先就宣告系統執行時所需要的資源
- 動態監控系統中的資源，Ex：還剩多少、已使用多少...
- 確保系統不會進入不安全狀態
- 如果系統目前在不安全狀態，則讓目前發送要求的 process 先 wait
- 其他 process 如果能讓系統在安全狀態就先執行

文

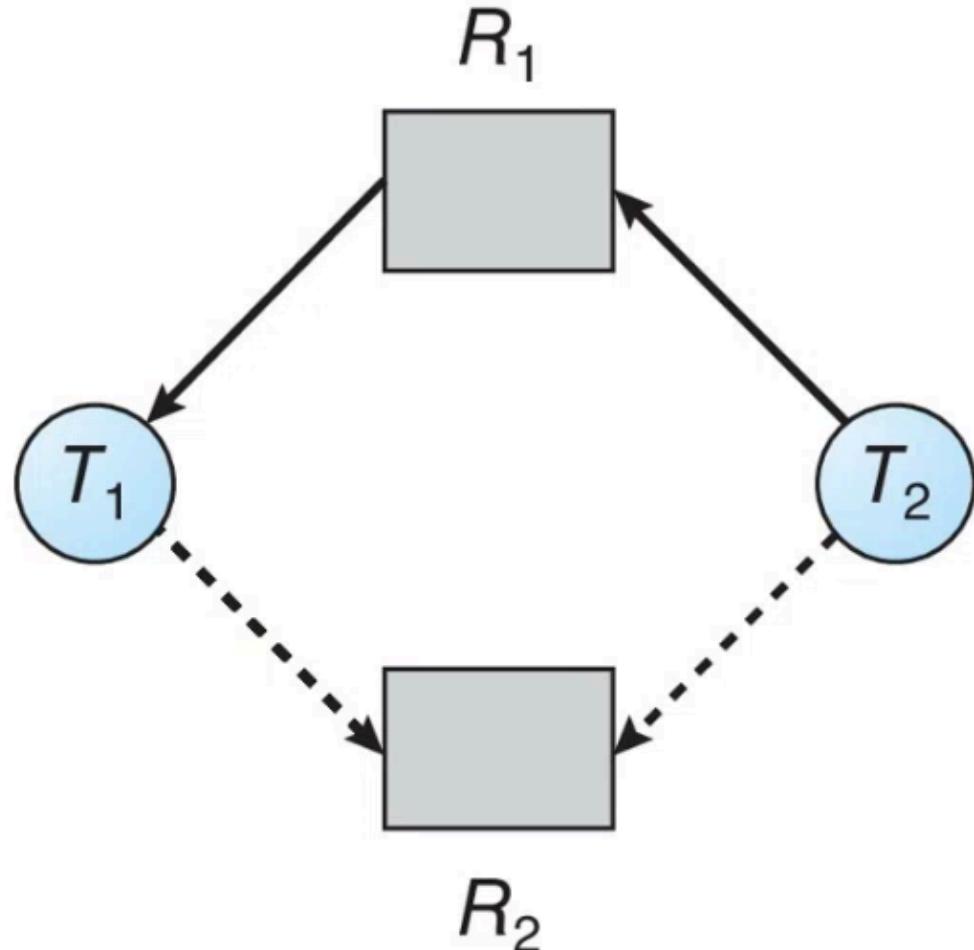
### What is Safe State?

- 如果現在系統剩下的資源能夠讓接下來所有的 process 都執行完

### How to inspect Safe State?

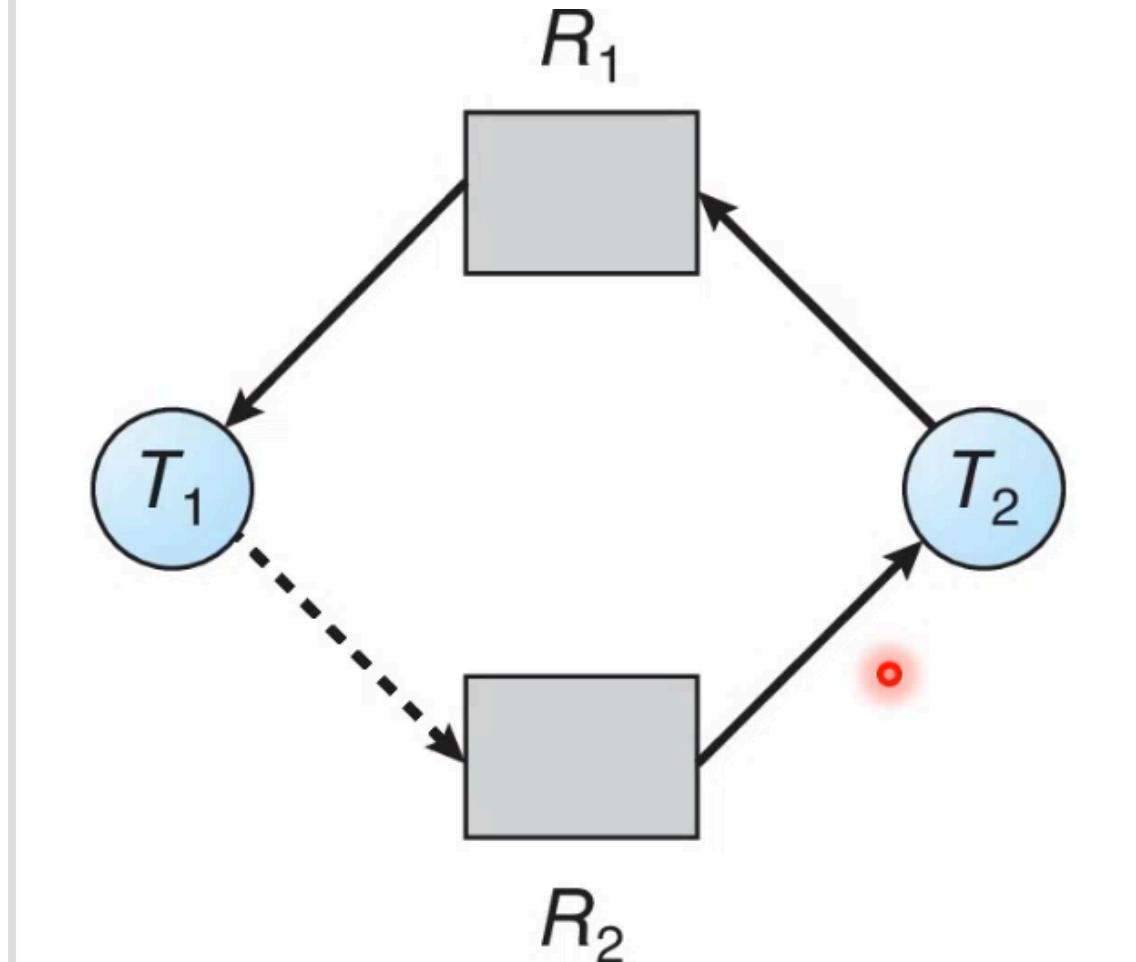
- Single instance
  - Use a resource-allocation graph
    - Claim edge (通常用虛線表示)：當 process 需要資源時，會需要先發送宣告，確保不會不安全，之後才會正式發送 request。

Example1 T1, T2 需要 R2 正在發送宣告 · T2 已發出 request 給 R1 · R1 再給 T1 資源



文A

Example2 T2 發送 request 後，R2 給 T2 資源但如果 T1 發送 request 給 R2 的話就會進入不安全狀態 所以 T1 發送宣告的時候就會被擋下來了



- Multiple instances
  - Use the Banker's Algorithm
    - Available : 表示系統目前各種資源有多少是可用的
    - Max : 每個 process 最多會用到多少的資源
    - Allocation : 目前配置多少的資源給每個 process 了
    - Need : 目前的 process 還需要多少資源

- o Safety Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:

**Work = Available**

**Finish [i] = false, for  $i = 0, 1, \dots, n-1$**

2. Find an  $i$  such that both:

(a) **Finish [i] = false**

(b) **Need<sub>i</sub> ≤ Work**

If no such  $i$  exists, go to step 4

3. **Work = Work + Allocation<sub>i</sub>**

**Finish[i] = true**

go to step 2

4. If **Finish [i] == true** for all  $i$ , then the system is in a safe state

- o Resource-Request Algorithm

- 判斷是否可以配置資源給 process
- 假設如果真的配置給 process 的話會不會不安全
- 如果不會的話再真的配置給他

**Request<sub>i</sub>** = request vector for process  $P_i$ . If **Request<sub>i</sub>[j] = k** then process  $P_i$  wants  $k$  instances of resource type  $R_j$

- 1. If **Request<sub>i</sub> ≤ Need<sub>i</sub>** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
- 2. If **Request<sub>i</sub> ≤ Available**, go to step 3. Otherwise  $P_i$  must wait, since resources are not available
- 3. **Pretend** to allocate requested resources to  $P_i$  by modifying the state as follows:

**Available = Available – Request<sub>i</sub>**;

**Allocation<sub>i</sub> = Allocation<sub>i</sub> + Request<sub>i</sub>**;

**Need<sub>i</sub> = Need<sub>i</sub> – Request<sub>i</sub>**;

- If safe  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored

- o Example

- Banker's Algorithm



執行順序如果是  $P_1, P_3, P_4, P_2 \dots$  則會保持 Safe  
 $P_0$  的話會不安全

- 5 processes:  $P_0$  through  $P_4$

3 resource types:

$A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3
$P_1$	2	0	0	3	2	2
$P_2$	3	0	2	9	0	2
$P_3$	2	1	1	2	2	2
$P_4$	0	0	2	4	3	3

(Need 可以從 Max - Allocation 獲得)

## Need

	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	7	4	3
$P_1$	1	2	2
$P_2$	6	0	0
$P_3$	0	1	1
$P_4$	4	3	1

文A

- Resource-Request Algorithm

### 比較 Available, Need

如果  $\text{Available}[i] > \text{Need}[i]$ ,  $\text{Available}[i] + \text{Allocation}[i]$

- Check that  $\text{Request} \leq \text{Available}$ : (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$ )

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

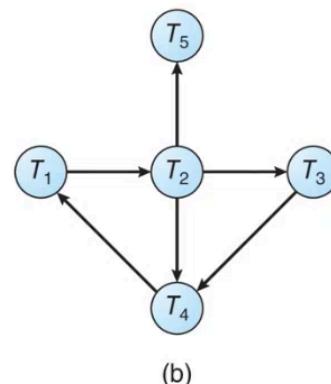
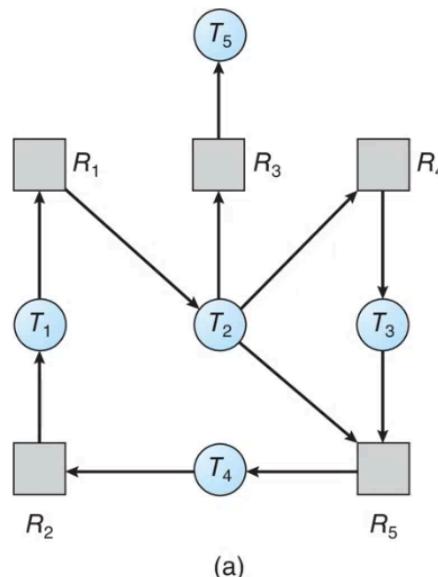
- Executing safety algorithm shows that sequence  $< P_1, P_3, P_4, P_0, P_2 >$  satisfies safety requirement

## Deadlock Detection

- Single instance
  - Wait-for Graph



Resource-Allocation Graph, Wait-for Graph 比較圖



Resource-Allocation Graph

Corresponding wait-for graph

- 演算法檢查是否 cycle · 最基本要  $O(n^2)$  的時間
- Multiple instances

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:
  - a) **Work = Available**
  - b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_i \neq 0$ , then  $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$

2. Find an index  $i$  such that both:

- a)  $\text{Finish}[i] == \text{false}$
  - b)  $\text{Request}_i \leq \text{Work}$

If no such  $i$  exists, go to step 4

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$   
 $\text{Finish}[i] = \text{true}$   
go to step 2

4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked

XA

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state

- o Example

- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2	1	1	1
$P_2$	3	0	3	0	0	0	1	1	1
$P_3$	2	1	1	1	0	0	1	0	0
$P_4$	0	0	2	0	0	2	1	1	1

- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = true$  for all  $i$
- $P_2$  requests an additional instance of type C

Request

	A	B	C
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

- State of system?

- Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes' requests
- Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$

- Usage
  - 偵測頻率等等參數需要是系統情況而定
- 發生 Deadlock 後該如何解開?
  - 停止全部 process 執行並重新啟動
  - 將發生 deadlock 的 cycle 一個一個終止
    - 先終止哪個 process ?
      - Priority
      - 執行過久的
      - 用太多資源的
      - 還需要很多資源的
      - 哪幾個 process 最快可以解開 deadlock
- Resource Preemption
  - 允許優先權較高的 process 先使用 => Selecting a victim

- 把原本暫停的 process 回復 => Rollback
- 如果 process 一直被暫停 => Starvation

# Chapter 9 : Main Memory

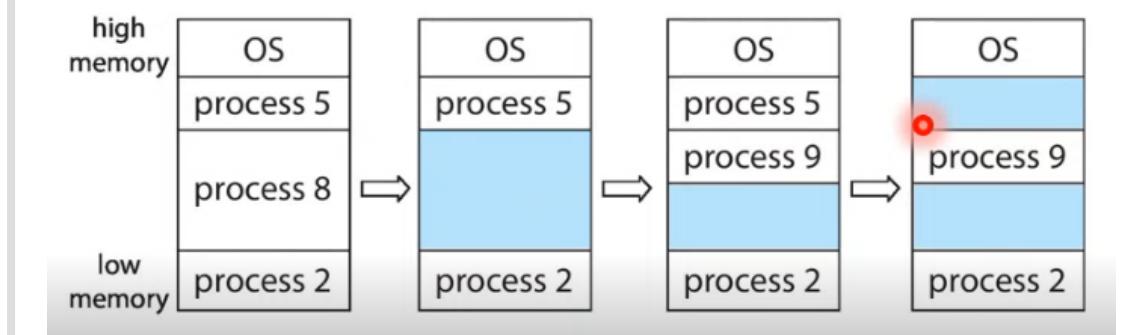
- Logical address : process 看到的 memory 位置 => 相對位置
- Physical address : Memory 中實際的位置
- Ex :
  - process 上面寫 memory 從 0 開始，但實際上的 Memory 不是從 0 開始，而是那個 process 的 0 開始
- Memory-Management Unit (MMU)
  - 將 Logical address 轉換成 Physical address 的東西

## Contiguous Memory Allocation

文

- 給每一個 process 一個範圍，讓他只能使用那個範圍
- 位置計算就直接相加就好
- 問題：
  - 限制了 process 使用記憶體的空間
  - 如果不用這麼大的空間會導致浪費
- 解決：
  - Variable Partition
    - 要多少空間就給多少空間
  - 延伸問題

記憶體可能會變得零碎，導致不連續 => Hole



- Fragmentation
  - External Fragmentation : 所有空間加起來很大，但是不連續
  - Internal Fragmentation : 配置很多空間，但實際上用的不多
- 解決
  - Dynamic Storage-Allocation Problem
    - First-fit
      - 選一個 Hole

- Best-fit
  - 比 process 大的所有 Hole 選最小的
- Worst-fit
  - 選最大的 hole，這樣切完的 hole 還是會很大
- compaction
  - 移動記憶體位置，使其合併
  - 但如果執行到一半移動位置，可能會造成 error
  - 而且會花很多時間

## Discontiguous Memory Allocation

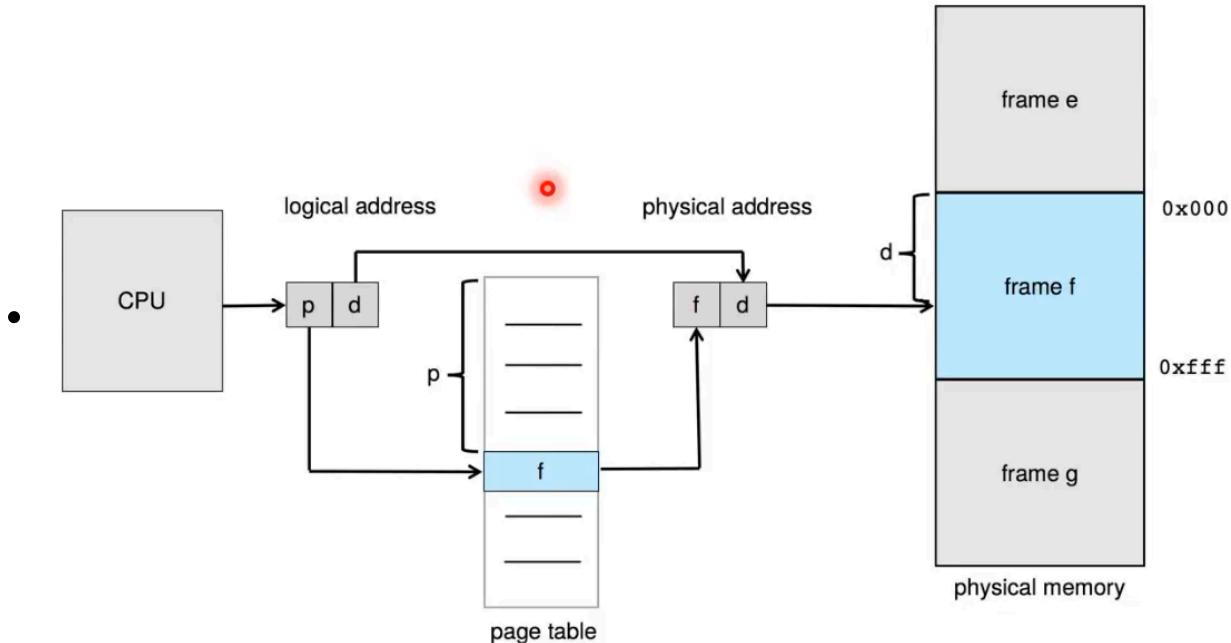
### Segmentation

- 將 process 分開放
- Data 放一塊，stack 放一塊，code 放一塊3
- 還是會有 External Fragmentation 的問題

文

### Paging

- 將 memory 切成好幾塊
  - physical memory 叫做 frames
  - logical memory 叫做 pages
- 假設 pages 是連續的，但在 physical memory 可以隨便放不用連續，只需要紀錄好 frames, pages 的對應關係
- page table : 存放 frames, pages 的對應關係
- free frames list : 紀錄 free frames (還沒有放入空間的 physical memory )
- page table 在 main memory 的表示
  - page-table base register (PTBR) : 紀錄 page-table base
  - page-table length register (PTLR) : 紀錄 page-table size
- Address Translation Scheme
  - page number : 可以透過 page number 查到對應的 frame number
  - page offset : 固定的大小，因為 physical memory 和 logical memory 是一樣大的



- 問題

- 還是有 Internal Fragmentation 的問題
  - 將 physical memory 切的更細，但 page table 所佔的空間也會變大
- 要花兩倍讀寫記憶體的時間：page table + physical memory
  - 將 page table 放在 cache (TLB)

文

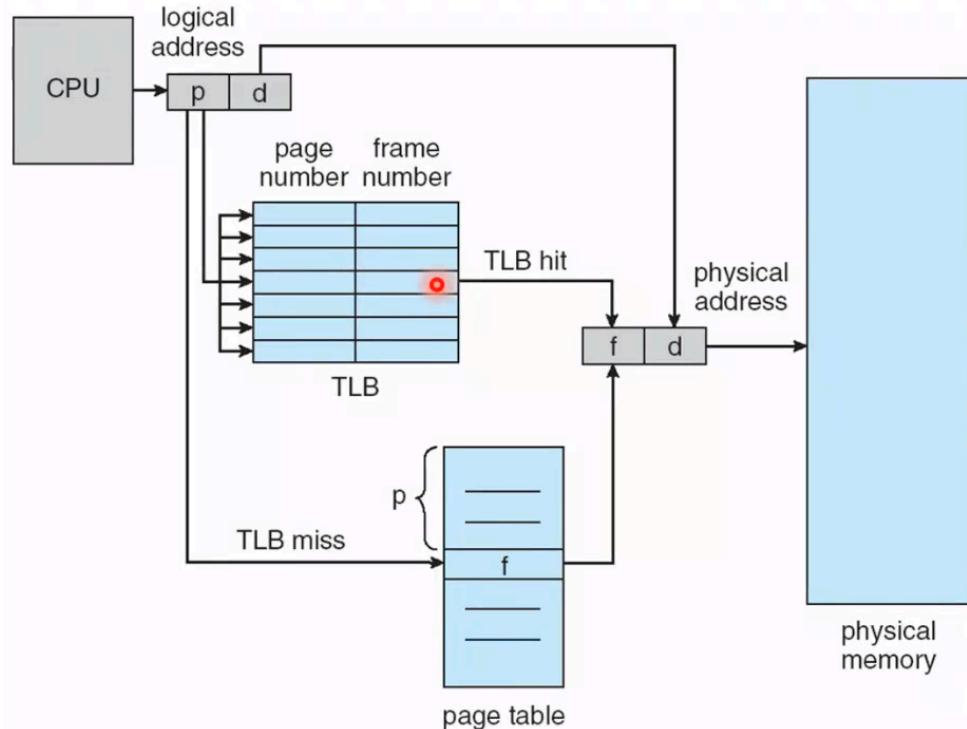
## TLB : 平行處理

因為 TLB 很小，如果 TLB miss 就還是要用 page table

Hit ratio : 80% 的 page number 都查得到

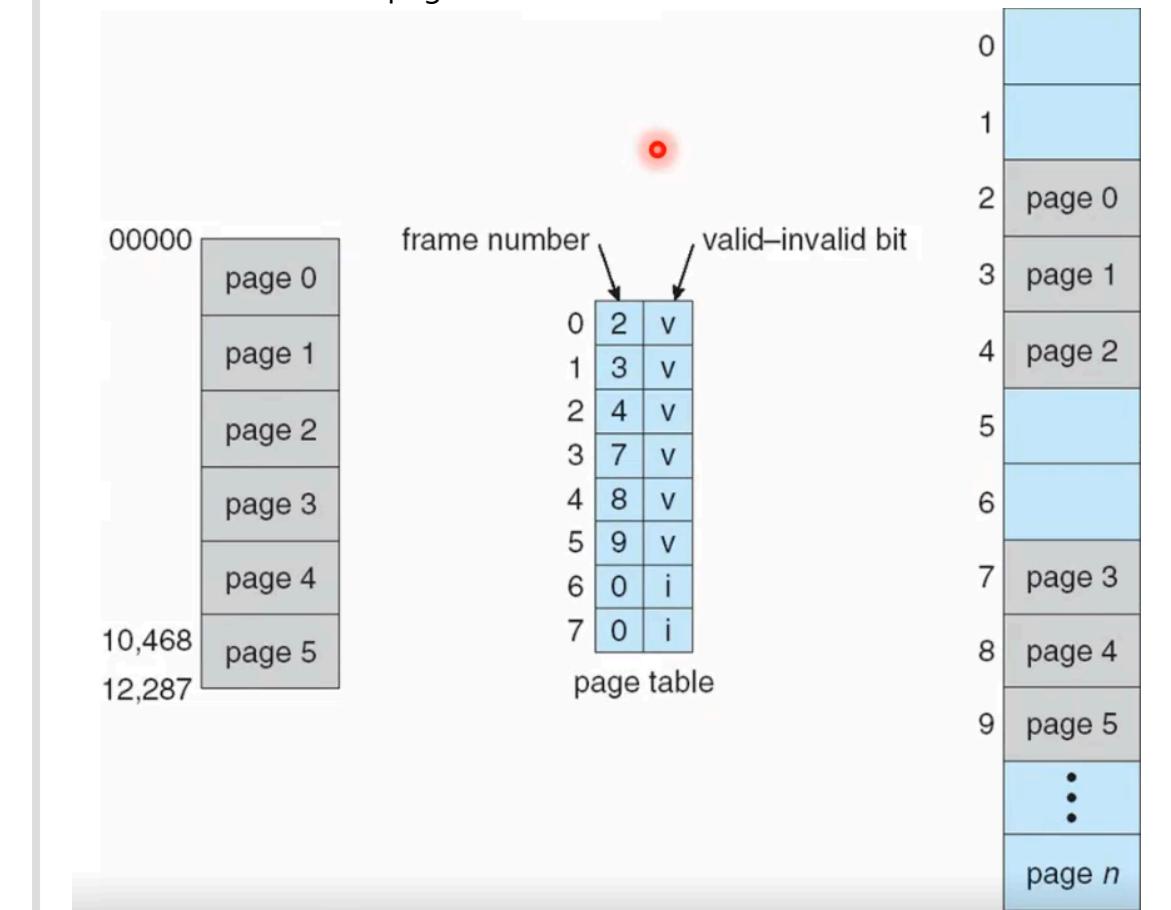
翻譯後置器 ( Translation Lookaside Buffer ) 的縮寫，是一種用於提高虛擬記憶體管理效率的硬體快取。TLB 主要用於處理虛擬地址到物理地址的轉換，以提高存取記憶體的速度。

通常與虛擬記憶體系統結合使用。當 CPU 存取虛擬記憶體時，它發出一個虛擬地址。這個虛擬地址必須經過轉換才能找到對應的物理地址，而這個轉換過程需要額外的時間。為了加快這個轉換過程，TLB 將最近的一些虛擬地址和對應的物理地址的映射關係保存在快取中。



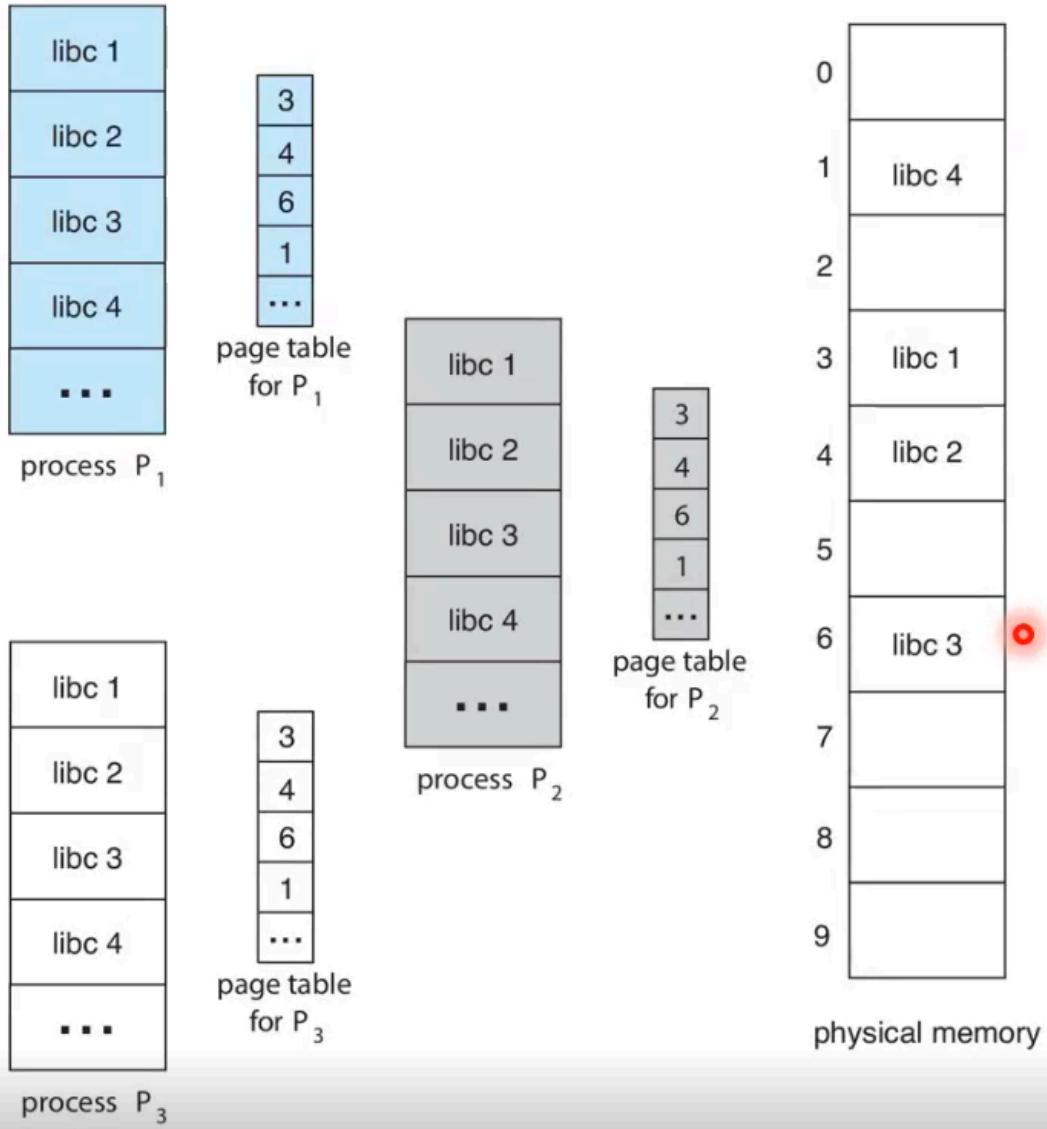
- page table 有可能會超出 physical memory 範圍

## Valid-invalid bit 紀錄 page 空間是否合法



- 有些 code 不需要每個都執行

Shared Pages : page number 對應到同樣的 frame number

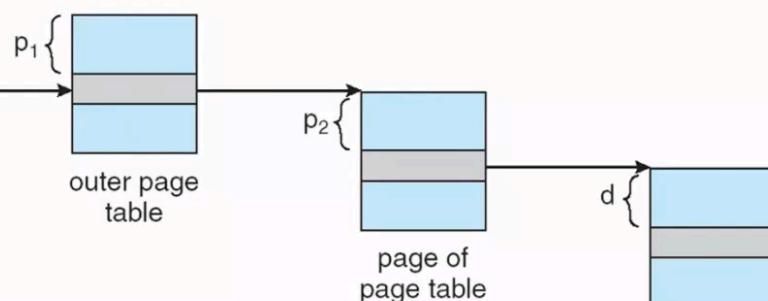
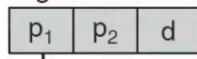


- 實際情況中 · page table 非常占空間

#### ■ Hierarchical Paging

兩層的 page table

logical address



- Hashed Page Tables
- Inverted Page Tables