Chapter 4 Process Management 與 Thread Management

- **Process && Program**

Process	**Program**
● A program in execution ● 是 OS 分配 Resource 的對象 ● Process 之建立，其主要組成 ■ Code Section ■ Data Section ■ Programming Counter ■ A set of CPU register ■ Stack, etc. ● "Active" entity	● 只是一個 file in storage space ● "Passive" entity

- Automatic & Explicit buffering

Automatic	Explicit buffering
不會定義 buffer 大小	會
確保 sender 在等待執行 copy a message 時不會被 block	Sender maybe blocked while waiting for available space in the queue
會預留 memory 空間給需要大量 memory 使用之工作	沒有這部分的限制

- Send by Copy & Send by Reference

Send by Copy	Send by Reference
不允許 receiver 更改變數的 state	允許
	允許 programmer 去撰寫一個中央應用程式的擴充版本

- PCB (Process Control Block)

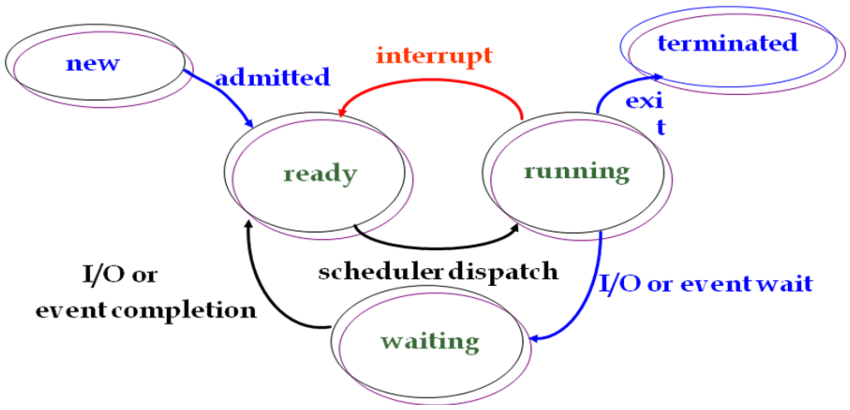
→為了管理 Process，會在 kernel space 中針對每個 process 準備一個 block (or Table)，記錄該 processes 之所有相關資訊，稱之為 PCB，其主要的組成內容有

- ✓ Process ID -> is unique
- ✓ Process State ->e.g. ready, wait, running, etc.
- ✓ Programming Counter ->e.g. 記錄下一條指令所在的位置
- ✓ CPU register
- ✓ CPU scheduling information ->e.g. Process 之優先權值 or time Quantum 大小
- ✓ Memory Management information ->e.g. Base/Limit register 或 page table
- ✓ Accounting information ->e.g. 使用了多少 CPU time, 多少 resources, etc.
- ✓ I/O status information ->e.g. I/O request 之處理狀況, etc.

NOTE: 哪些東西不在 PCB 中

➔ I/O device queue, Bitmap(OS 管 Disk Free space 用)

- Process 之 state transition Design(STD) (已鮮少學校不用恐龍版)
  - [恐] 5 states STD



  - [Moderm] 3 states STD
  - [Stalling] 5 states, 6states, 7states, Unix states

- Concurrent Process

	Multiprocessor System	Distributed System	Concurrent Process
優點 Or 原因	1. Increased Throughput 2. Increased Reliability 3. Economy of scale(硬體擴充)	1. Resource sharing -> cost節省，因為某些resource在某部machine上有提供，故machine不須自己私有，而可共享之。 2. speed up -> throughput提升 3. reliability 4. communication	● Resource sharing ● Speed up computation ● Modularize ● Convenient

- Process Control 之 operations
  - 建立 Process (Create)
  - 終止 Process (Terminate)
  - 暫停 Process (Suspend)
  - 恢復執行 (Resume, Wake up)
  - 設定/更改 Process 屬性, etc.

- Process Creation 之相關探討

(一) Process 可建立他的 Process

1. 被建立者 -> child process
2. 建立者 -> parent process

(二) 建立 child process 之目的為合?

1. 作工
2. 執行 parent 所交辦的事項
   i. 與 parent 相同的工作
   ii. A special task

(三) Child process 所需的 resource 由誰提供?

1. 由 OS 直接供應 -> OS 負擔重，往往會限制 child 的生成數
2. 由 Parent 供應、分派

(四) Child 與 parent process 之運作互動模式

1. Concurrent execution
2. Parent waits child until child terminate

- 幾個重要的 system call
  - fork()：此 system call 目的在於建立 child processes，其回傳值，依傳回對象或結果而有所不同，傳回值有三個：
    - <0：表 fork()失敗
    - ==0：此值會傳給 child process
    - >0：此值會給 parent process，且此值表是 child process ID
  - execlp()
    - 此 system call 用以載入，特定的 object code 到 memory 中，成為某個 process 之 code section，且執行此 object code
  - wait()
    - 此 system call 用來強制 process 暫停，即 running -> wait(Blocked)
  - exit()
    - 此 system call 用來終止 process
      - exit(-1)：不正常之結束
      - exit(0)：正常之結束

- What shall an OS do in order to initiate the execution of a program after loading the program into the memory ?

  1. Process creation 2. Scheduling 3. Context switching
  4. Change mode to user mode 5. Jump to the execution entry of user process

- Scheduler

	Long-term Scheduler	Medium-term Scheduler	Short-term Scheduler
Def.	又稱 Job scheduler	常用於 Time sharing system 當 memory 不足且又有其他 process 需要 memory space，則此時此 scheduler 會被啟動	又稱 CPU scheduler 或 process scheduler
目的	從 Job Queue 中，排出合適的 Jobs，載入 memory 中以便 ready for execution	是要挑選一些 process(e.g. sleep/Blocked in Memory or lower-priority)，將它們 swap out 到 Disk(or Blocking store)，以便空出 memory 給其他 process 使用，等到 memory space 足夠時，再將它們 swap in 回 memory 中之 ready for execution	從 Ready Queue 中，排出 Highest priority process，使之取得 CPU 執行
特點	1. 執行頻率最低 2. 可調控 Multiprogramming Degree 及 CPU Bound 與 I/O Bound Job 比例 3. 通常 Batch system 可用，但 real time system，time sharing system 不用	1. 執行頻率居中 2. 可調控 Multiprogramming Degree 及 CPU Bound 與 I/O Bound Job 比例 3. Time sharing system 使用，但 HW Real time 不用	1 執行頻率最高 2 所有 system 皆用 3 不能調控 Multiprogramming Degree 及 CPU Bound 與 I/O Bound Job 比例

- Context Switching
  - ➔ 當 CPU 要從執行中的 process 切換給另一個 process 執行之前，必須保存目前 process 之執行的狀態，資訊(i.e. store into its PCB)，且載入另一個 process 之狀態資訊(i.e. Load from its PCB)，此一工作稱之 Context switching 是一種負擔(因為 CPU time 來用於 process execution 上)，其時間長短大都取決 HW。
  - ➔ 如何降低 Context switching 的負擔
    - 使用 register：若 register 的數量夠多，則可以讓每一個 process 皆可擁有自己的(private) register set，將來 context switching 時，OS 只要切換指向 registers set 之"register set pointer"即可(因為不需要 Memory load/store, 所以 speed up)
    - 利用 Thread，來降低 Context switching 的負擔
      (Thread 是一種 light weighted process，同一個 process 內的 threads 彼此共享資源，相對的 thread 使用的 local register 數目較 process 少，一旦 thread 之間作 context switching, 則 store old thread data 與 load new thread data 之 effort 便較輕)
    - 讓 user process 與 system process 擁有各自的 process register set，所以當 user process 與 system process 切換時，OS 只要改變 register pointer 即可。
  - ➔ OS 如何做"Context Switching"
    OS 須將 process 使用的 CPU register 之內容及其他的 information 存入 memory 中，再將新的 process 載入 CPU register. OS 須將目前 PCB pointer 其所在之 PCB record，再將控制權交給 Next process

- Dispatcher
  - 此模組負責將 CPU 授予經由 CPU scheduler 所挑選出的 process，其主要工作有三
    - Context switching
    - Change mode to user mode(if needed)
    - Jump to the execution entry of the selected process

    上述這三個工作的"時間和"稱為"Dispatch Latency"，其越短代表 process 可以開工的時間就越提早。
- 評估 scheduling performance 之五個 criteria
  - CPU 之 utilization
    - CPU time 用在 Process execution 上 / CPU Total Time
  - Throughput
    - 單位時間內完成工作數目
  - Waiting time
    - Process 待在 ready queue 裡等待取得 CPU 之等待時間和
  - Turnaround time
    - 自 process 進入系統到它完成工作的這段時間和
  - Response time
    - 自 user command 交付給 system 到系統產生第一個回應的時間和。

  Note: Time sharing system, user interactive AP 重視此 criteria

- Starvation
  - Process 因為長期無法取得完工所須的各類資源，導致此 process 遲遲無法完工，形成"Indefinite Blocking"之現象
  - 解決方法：利用"Aging"技術

    (Aging -> 隨著 process 待在系統內的時間越久，逐步增加其優先權值，經過一段有限時間後，其值會變成最高，故此 process 可取得資源完工，所以不會 starvation)

- Non-Preemptive && Preemptive

	Non-Preemptive	Preemptive
Def.	執行中的 process，除非其自願放掉 CPU(ex. Wait for I/O complete, terminate, etc.)	執行中之 Process，可能會被迫放棄 CPU，交給其它 CPU 使用(ex. Time out, Interrupt, 高優先權 Process 到達, etc.)
優點	1. Process 完成工作的時間較可預期 2. Context switching 之次數較少	與 Non-Preemptive 優缺點
缺點	1. 排班效益較差(ex. Convey effect 發生, waiting time ↑) 2. 不適用於 Real time system, Time sharing system	
✓ 以 CPU 排班決策發生的時機點，作為區分 non-preemptive & preemptive 的依據		

Note:

Convey effect : 許多 process 均在等待一個需要很長 CPU time 才完工之 process 完成工作，此舉會造成 Average waiting time 大幅增加之不良效果

- Scheduling Algorithm

	Definition	特性	公平？	Starvation?	Preemptive?
FIFO	最早到達的 process, 優先取得 CPU 使用	1. 簡單，容易實施 2. 會有 Convey effect	V	X	X
SJF	具有最小的 CPU Burst time 的 process，優先取得 CPU 執行	1. 排班效益最佳(因為 Avg. wait, turnaround time 最小) 2. 不會有 Convoy effect 3. 不適用於 Short-term scheduler，較適用於 Long-term scheduler(因為 short-term scheduler 執行頻率太高，不易在極短之時間內，去精確預期出 Process 之 Next CPU Burst time 且選出最小值)	X	V	X
SRJF (SRTF, SRTN)	是 Preemptive SJF，即若新到達的 Process，其 CPU Burst time 小於目前之執行中的 Process 剩餘的 CPU time，則新的 Process 可以插隊執行，強迫執行中之 Process 放棄 CPU	✓ 與 SJF 相比，Avg. Waiting/Turnaround time 較小，但 Context switching 次數可能較多	X	V	V
Priority	具有 Highest priority 之 process，優先取得 CPU，若權值相同，則以 FIFO 為準	1. 可分為 Preemptive 和 Non-preemptive 2. 不同之優先權定義，可以演化出不同的法則，是一個可參數化的法則	X	V (可用 Aging 解決)	
RR (Round Robin)	常用於 Time Sharing System，OS 會規定一個 CPU time(or Slice), 若 Process 取得 CPU 後，未能於此 Quantum 內完工，則必須被迫放棄 CPU, 等待下一輪再取得 CPU 使用 註一：Virtual RR	1. 常用於 Time Sharing System 2. RR 之效能取決於 Time Quantum 大小的制定 i. Quantum = ∞, -> FIFO ii. Quantum = 極小值, -> Context switching 太頻繁，CPU 毫無產能可言 iii. 實驗顯示，"Quantum 大小最好能讓 80%的工作，可在此 Quantum 內完工"，效率較佳 3. RR 法則可參數化	V	X	V
Multilevel Queues	1. 將單一一層 ready Queue 分成許多不同優先等級的 ready Queues 2. 通常 Queue 與 Queue 之間採 Preemptive priority 法則 3. 每個 Queue 也有自己的排班法則	✓ 此種設計可增加排班設計或效能調整的彈性(因為可參數化的項目多樣化) ✓ EX. 可參數化的項目：Queue 的數目、Queue 與 Queue 之間的排班法則、各個 Queue 的排班法則、Process 英放入哪個 Queue 中	X	V	V

	4. 不允許 process 在各 Queue 之間移動				
Multilevel Feedback Queues	與 Multilevel Queue 類似(1.~4.相同) 差別：允許各 process 可以在各 Queue 中移動，如此一來可防止 starvation(註二)	■ 增加 Flexibility，因為可參數化的項目增多	X	X	V
SRRN	具有最大的 Response-Ratio 值者，優先取得 CPU Response-Ratio = $\frac{S+W}{S}$ S : service time W: Waiting time		V	X	X

註一：

Virtual RR

1. 緣由： RR scheduling 在某些層次來看，似乎對 I/O Bound Job 不是那麼公平，因為 I/O Bound Job wait for I/O complete 過程中，CPU-Bound Job 有較多時間取得 CPU 執行，而一旦 I/O Bound Job 變成 ready 時，仍需要與 CPU-Bound Job 競爭
2. 解決：除了正常的 ready Queue 之外，另增加一個 Aux. ready Queue(輔助型)，放此 Queue 中的 process 有較高的優先權可取得 CPU，所以當 I/O Bound Job 回 ready 時，因為是置入 Aux. ready Queue，以便較 CPU-Bound Job 先取得 CPU 執行

   Note: 如何在 RR scheduling 下，設計一個支援 Higher-priority process 取得較多的 CPU time，即支援 priority scheduling 的特色？
   [法一] 在 ready queue 中，針對 Higher-priority process，置入多份 PCB pointers 在 Queue 中排隊，如此在一輪迴中，此 process 有多次機會取得 CPU 執行
   [法二] Higher-priority process 之 CPU time Quantum 大於 Lower-priority process 之 Quantum 值

註二：EX. 採 Aging 技術，每隔一段時間，將 lower-priority process 往上移一層 Queue，經過有限的時間後，此 process 會被置於 Highest priority queue 中，所以不會 Starvation

■ Discuss how the following pair of scheduling criteria conflict in certain settings(恐習)
   a. CPU utilization and response time
   b. Average turnaround time and maximum waiting time
   c. I/O device utilization and CPU utilization

- a. 欲提升 CPU utilization 可以藉由降低 Context switching 的執行頻率，如此會造成 process 之 response time 下降
- b. Avg. turnaround time 下降 by executing the shortest task first. 但是這樣會導致 long time task starvation，而他們之 waiting time 上升
- c. CPU utilization is maximized by running long-running “CPU-bound” task without context switching. I/O device utilization maximized by scheduling I/O bound jobs as soon as they become ready to run, thereby increasing the overheads of context switches.

■ Multiprocessor System 與 Real-time System 支排班設計考量/原則

<table>
<tr><td colspan="3">Multiprocessor System</td></tr>
<tr><td colspan="3">一、每個 CPU 有各自的 ready Queue，只會存取自己的 ready Queue，不會存取其他人的)</td></tr>
<tr><td colspan="3">二、所有的 CPUs 均共享一個 ready Queue，分為下列兩種 model</td></tr>
<tr><td></td><td>Symmetric mode</td><td>Asymmetric mode</td></tr>
<tr><td>Def.</td><td>給個 CPU 街可以去存取此 ready queue</td><td>只有 Master CPU 才可存取此 Queue，其餘 Slave CPUs 不可存取</td></tr>
<tr><td>優點</td><td>Performance 較佳</td><td>容易從單一 CPU 的 OS 修改取得，設計較簡單(因為只有 Master CPU 才可存取 ready queue, 無互斥必要)</td></tr>
<tr><td>缺點</td><td>設計較複雜
因為需對所有 CPU 共享的 Resources 及 Kernel Data Structure 提供互斥存取的同步控制機制，以防止共享資料發生 race condition</td><td>Performance 差, Master CPU 會是 Bottle neck
可靠度低, 萬一 Master CPU 掛了</td></tr>
</table>

<table>
<tr><td colspan="4">Real-time System</td></tr>
<tr><td colspan="2">Hard Read-time System</td><td colspan="2">Soft Real-time System</td></tr>
<tr><td>一、決定是否可排程化？</td><td>Process 之所有可能執行時間延遲的加總(ex. Execution time, OS service time, etc.)，是否小於 Time constraint，若是，則 schedulable, 否則 non-schedulable</td><td>原則</td><td>1. 要能夠支援 Priority scheduling 法則
2. 不提供”Aging”技術
3. 降低 Kernel Dispatch Latency，使得 Real-time Process 可以盡早開始執行(註一)</td></tr>
<tr><td>二、schedulable 公式(背)</td><td>$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$
n : Event 的總數
Ci: 表 Event 之 CPU burst time
Pi: 表 Event 之發生週期</td><td colspan="2">註一(第三點的補充)</td></tr>
<tr><td></td><td></td><td>緣由</td><td>某些 OS 不允許 system calls 執行過程中被中斷，目的是防止 Kernel Data Structure 發生 Race Condition，資料內容會不正確。
可是，此舉對 real-time process 極為不利，因為萬一有 long-time system call 正在執行，而 real-time process 到達又不能插隊，需等 Long-time system call 做完之後，才能取得 CPU，導致 Dispatch Latency 太長</td></tr>
<tr><td></td><td></td><td colspan="2">解決</td></tr>
<tr><td></td><td></td><td>[法一]</td><td>Preemptive Point
在 system call 中適當處加入 Preemptive Point，將來 system call 若執行到此處遇到 Preemptive Point，會暫停下來檢查是否有 real-time process，若有，則讓 real-time process 先執行，system call Blocked。

缺點：通常，在 system call 中可加入的 Preempted Point 數目並不多，所以 Dispatch Latency 仍長</td></tr>
<tr><td></td><td></td><td>[法二]</td><td>允許整個 Kernel 可被隨時 Preempted
必須對 Kernel Data Structures 及共享資源提供互斥存取之 synchronization 機制，防止 Race Condition</td></tr>
</table>

		缺點：會產生"Priority Inverse"之問題(註二)

註二：Priority Inverse(優先權反轉)

Def. 高優先權的 Process 所欲存取的 Data Structure 或 Resource 恰被一些低優先權的 Process 所把持(未釋放)，造成高優先權之 Process 等待低優先權 Process 釋放，共用 Data Resource 的情況，若在加上低優先權的 Process 遲遲無法取得 CPU，完成資源的使用，則高優先權的 process 會等更久

解決方法："Priority Inheritance" Protocol(優先權繼承)

➔ 讓低優先權的 process 暫時繼承高優先權 process 之權值，以便盡快取的 CPU 執行，完成資源使用後且釋放，然後恢復其原先的權值。

- Thread
  - Def.： 又稱"Lightweight Process"是 OS 分配 CPU time 之對象單位。
    - Thread 建立後，所擁有的 private info 之主要組成: Thread ID, CPU register, Program Counter, Stack, etc.(交大)
    - 而一個 Process 內不同 Threads 彼此共享
      - Code Section
      - Data Section
      - Other OS resources (ex. Openfiles, signal, etc)
  - Benefit (背)(筆)
    - Responsiveness
    - Resource sharing
    - Economy
    - Utilization of Multi-processes Architecture (or Scalability)

- Process v.s. Thread

Process	Thread
Heavy weight process	Light weight process
OS 分配 resource 之對象	OS 分配 CPU time 之對象
不同 process 之間無共享之 memory 與 resources (除了 share memory 溝通之外)	同一個 process 內的不同 Threads 共享 memory 與 resources
Signal-Thread	Multi-Thread
Process 內的單一 Thread 若 Blocked，則整個 Process 亦 Blocked	Process 內還有 Thread 可以執行，則 Process 就不會 Blocked
Process Management Cost 貴	Thread Management Cost 便宜
不易發揮 Multiprocessors 架構之效益	並行化程度高，較可發揮 Multiprocessors 架構之效益

- Threads 種類(View1 : Thread Management 由誰負責)

	User Thread	Kernel Thread
Def.	✓ Thread Management (e.g. creation, scheduling, etc.) 是用在 user mode 端執行的 process 或 thread library 負責 ✓ 完全不須 Kernel 參與 ✓ Kernel 亦不知道 User threads 存在	Thread Management 完全由 Kernel 負責
優點	✓ Fast (creation and context switching) ✓ Thread Management 之 cost 更便宜	優缺點與 User thread 相反 ➢ How to make use of threads ? 在 client-server 的架構，我們可以建立一個 thread
缺點	● 若某一 User Thread 發出一個 Blocking System call，即使 Process 內還有其他 available	

	Threads，仍會導致整個 Process 亦 Blocked(因為會被 Kernel 認為是一個 process) ● 無法做到在同一個 process 支不同 user thread 在多顆 CPU 上平行執行，所以 Multiprocessor 架構發揮效益較差	負責監聽 clients 的 request，一旦收到 client request，則可以建立其他 thread 去處理此 request

- Threads 種類(View2 : [恐])

	Many-to-one	One-to-one	Many-to-many
Def.	This model maps many user threads to one kernel thread	This models maps each user thread to a kernel thread	This models multiplexes many user threads to a smaller or signal number of kernel threads
優點	同 user thread	同 kernel thread	1,2. 同 kernel thread 3. OS 負擔不如 one-to-one 來的重
缺點		1. 同 kernel thread 2. 因為每生出一條 user thread，系統就必須生出一條 kernel thread 與之對應，所以 kernel 負擔重 (通常限制 process 生成 user thread 的數目)	設計較為複雜，製作成本高
例子	Thread Library	Windows 2000, XP, NT, OS/2…	Sun Solaris 2 以上

- Thread Pool
  - 緣由：在 client-server 架構中，當 server 收到 client's request 後，server 才建立 thread，執行對應之服務請求，但因為 Thread Creation 仍需耗費一點時間，以至於對 client 之回應不夠迅速
  - 作法
    - Process 事先建立多條 Thread 置於 Thread pool 中
    - 一旦 server 收到 client 之請求，則到 thread pool 中找出一條 available thread 負責此項請求，thread 完成後，再回到 thread pool 中等待下一份工作。若 thread pool 無可用的 thread，則 client's request must wait.
  - 缺點：若 process 事先建立為數眾多的 thread 在 stand-by，則會造成系統負擔過重
    - ➔ 所以通常會限制 thread pool 之 size

- Copy-on-Write

  當 parent 以 fork()產生 child process 時，child 會共享相同的 page，而不需針對 child 配置全新的 page，若 child process 欲修改某 page，則此 page 會被標示成" Copy-on-Write" page，那產生一個 copy page 給 child process 專屬修改而不影響 parent process 之 page

  優點：節省 fork 出 child process 的負擔

- Zero-fill-on-demand

  為了避免一開始 initial 太多 pages，而沒被使用所造成的浪費，所以只在需要的時候才產生新的 page

- Multi-core Programming
  - Multithread programming provides a mechanism for more efficient use of multiple cores(thread can run in parallel)
  - Multi-core systems putting pressure on system designer and application
  - Multi-core Programming 的五個挑戰
    - Dividing activity
    - Balance
    - Data Splitting
    - Data dependency
    - Testing and debugging