作業系統：Processes

目錄

1. Process Concept

Process 意旨正在執行中的程式。當你點兩下 .exe 檔案或在 terminal 輸入指令時，系統會把這個程式載入到記憶體並執行，這時它就成為一個 process。這一刻起，它就不再是硬碟上靜靜躺著的檔案，而是活生生、有生命週期、有資源的執行單位。

Process 本身內容涵蓋：

- 程式計數器(Program Counter)：紀錄下一條要執行的指令位置。
- 記憶體內容：包含程式碼、資料、堆疊、堆積區等。
- 系統資源：如檔案描述器、I/O 裝置等。

此外，Process 與 Program 的差異：

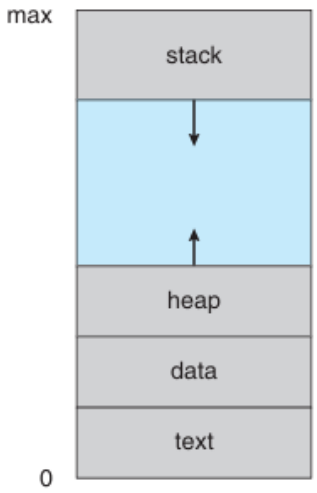
- Program 是寫好的程式，儲存在硬碟上。Process 是執行中版本的程式，具有活躍狀態與系統資源。
- 一個程式可以對應多個 process(例如你開了兩個 Google Chrome 視窗，每個就是一個 process)。

2. Process 結構

Process 結構可以分為：

- Text：放程式碼（機器碼），是執行的內容，不可更動。
- Data：放全域變數，可能有初始化值或尚未初始化。
- Heap：放動態配置的資料，例如 malloc()、new 建立的空間，可擴展或釋放。
- Stack：當函式被呼叫時，參數、區域變數、回傳位址會被壓入堆疊，函式結束時再彈出。

Figure 1：process 架構



3. Process 狀態(Process States)

一個 Process 不會永遠都在執行，它會根據當前情況切換狀態。以下是常見的五種狀態：

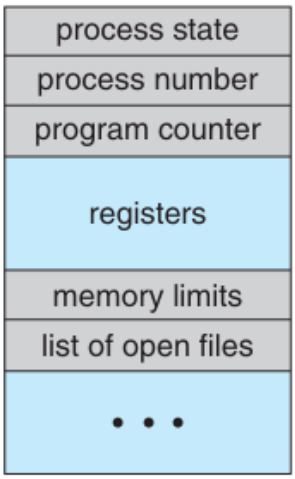
- New：剛被建立，尚未執行。
- Ready：等著 CPU 來執行。
- Running：目前正在 CPU 上執行。
- Waiting：暫時被擱置，等待某事件(如 I/O)完成。
- Terminated：執行完畢，被結束。

4. 行程控制區塊(Process Control Block, PCB)

作業系統需要記錄每個 Process 的詳細資訊，才能進行管理。這些資訊就存在一份名為 PCB（Process Control Block） 的資料結構中。內容包含：

- Process 狀態(例如 Running、Waiting)
- 程式計數器(下一步要執行哪行程式)
- CPU 註冊內容(中斷時需要保存)
- 記憶體相關資訊(如頁表、基底/界限)
- I/O 狀態(哪些檔案開著、使用哪些裝置)
- CPU 排程資訊(優先權、排隊指標等)

Figure 2：PCB



5. 行程排程(Process Scheduling)

CPU 是多個 Process 共用的，為了讓每個 Process 都有機會執行，作業系統會進行排程（Scheduling）。其目標有兩個：

- 多工（Multiprogramming）：讓 CPU 不要閒著，提高效能。
- 分時系統（Time Sharing）：讓使用者感覺每個程式都在「同時」運作。

排程的運作方式：

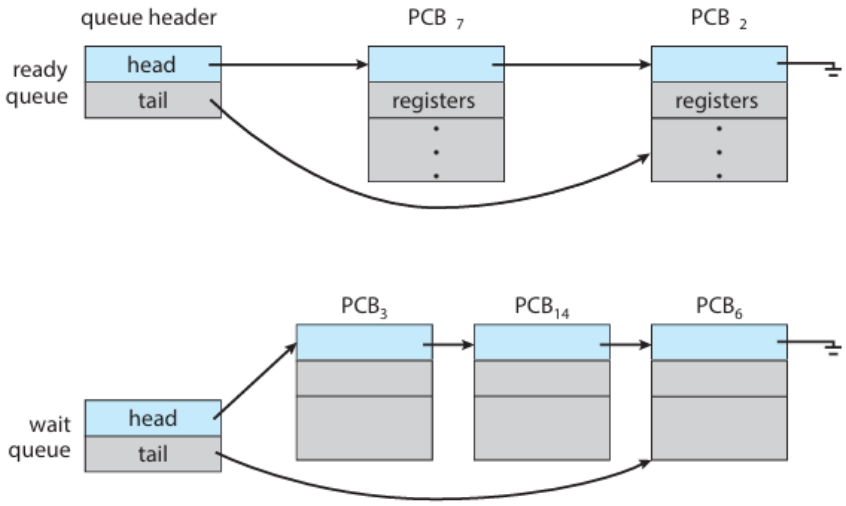
- 使用一個 Scheduler（排程器），從就緒佇列中選出下一個要執行的 Process。
- 每次 CPU 執行完一段時間（或發生 I/O 等事件）後會切換 Process，讓別的 Process 上場。
- 在多核心系統中，每顆核心可同時執行一個 Process，其餘的 Process 排隊等候。

5.1 行程排程佇列（Scheduling Queues）

排程時，作業系統會根據 Process 所在狀態，將它們放入不同的佇列中管理：

- 就緒佇列(Ready Queue)：儲存「準備好要執行」的 processes。實作為鏈結串列，佇列頭指向第一個 PCB
- 等待佇列(Wait Queues)：當 process 執行中呼叫 I/O 等操作時，會進入等待狀態。如 I/O wait queue、child termination wait queue 等
- I/O Queue：Process 等待硬碟、滑鼠、鍵盤等設備

Figure 3：The ready queue and wait queues



6. 上下文切換(Context Switch)

電腦裡的 CPU 數量是有限的，但同時想被執行的 process 卻有很多。當作業系統要從一個行程（process）切換到另一個行程時，會發生一件非常重要的事：保存與恢復行程的狀態。這個過程就叫做「上下文切換（Context Switch）」。

當 process 被中斷或 CPU 要切給別的 process 時，會：

1. 要保存目前 process 的狀態 到 PCB(state、registers、memory info)
2. 再載入新 process 的狀態，恢復執行

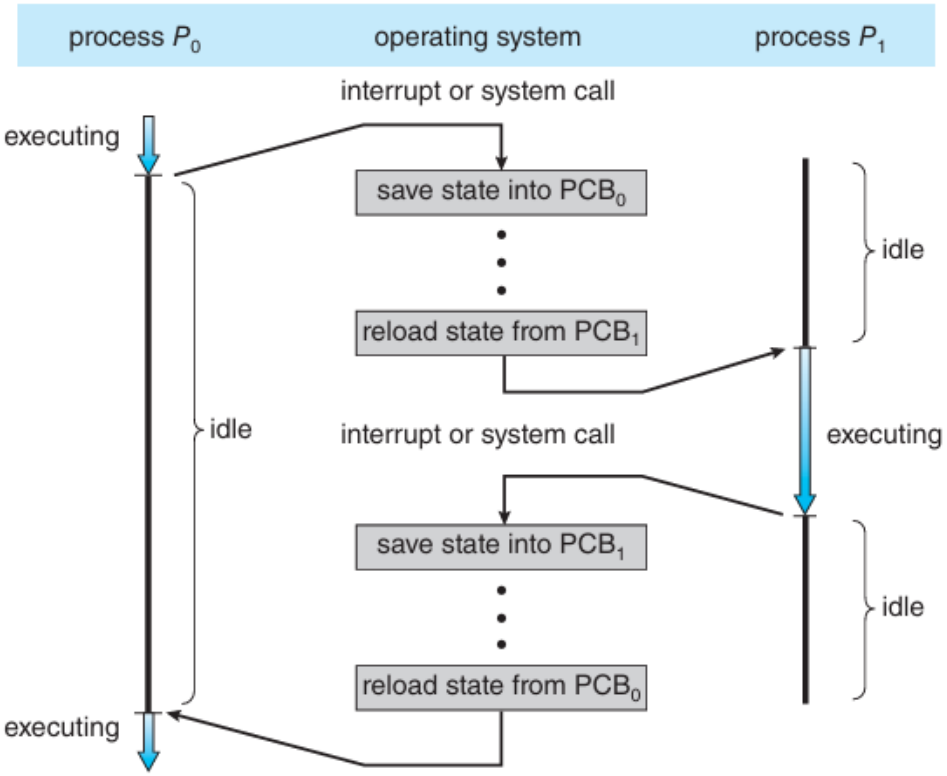
Context Switch 是有代價的。雖然 Context Switch 是必須的，但它本身 不做任何實際運算工作，只是「搬資料、切換角色」，所以：

- 它是一種系統開銷（overhead）
- 切換越頻繁，CPU 用在「準備演出」的時間就越多，真正執行程式的效率反而下降

哪些因素會影響 Context Switch 的速度？

- 暫存器數量：要保存/還原的資料越多，切換越慢
- 記憶體存取速度：讀寫 PCB 的速度會影響整體切換時間
- 是否有硬體支援：有些 CPU 提供快速切換指令，能加速上下文保存與恢復
- 作業系統實作方式：有些 OS 設計更精簡，能減少不必要的切換開銷

Figure 4：Context switch 的過程示意圖

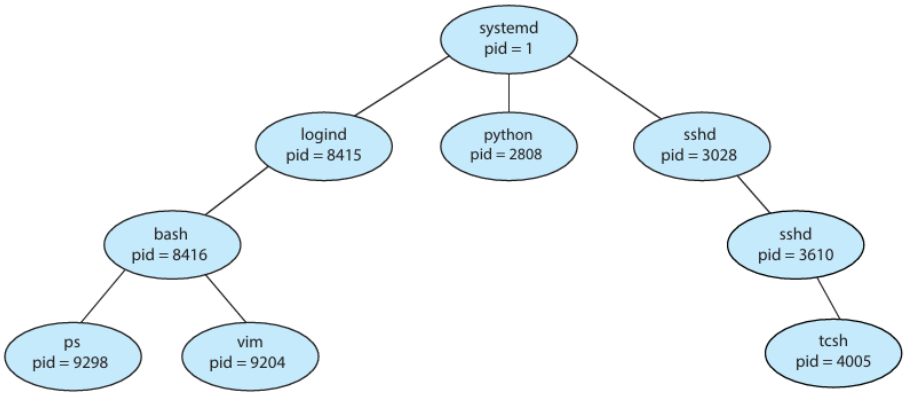


7. 行程操作(Operations on Processes)

7.1 行程建立(Process Creation)

在作業系統中，行程是可以產生其他行程的。我們把產生新行程的行程叫做父行程（Parent Process），被建立出來的叫做子行程（Child Process）。每個行程都會被分配一個獨立的 PID（Process ID），而這些父子行程之間的關係會形成一顆 行程樹（Process Tree）。

Figure 5：Process Tree



7.2 fork() 與 exec()(UNIX 系統)

- fork()：複製自己

當一個行程呼叫 fork()，它會產生一個幾乎一模一樣的子行程。這個新行程會有自己的 PID，但內容（程式碼、記憶體）最初與父行程相同。父子行程會從 fork() 後面那一行繼續各自執行。

- exec()：換程式內容

  呼叫 exec() 的行程，會把目前的程式內容整個替換成另一個程式，但 PID 不會變。也就是說，「身體還在，但靈魂被換了」。

Figure 6：fork() , exec() 結合使用

```c
pid = fork();  // 建立子行程

// 子行程，pid 回傳為 0
if (pid == 0) {
    execlp("/bin/ls", "ls", NULL);  // 執行 ls 程式
} else {
// 父行程，pid 是子行程的 pid
    wait(NULL);                     // 等子行程結束
    printf("Child Complete\n");
}
```

7.3 行程終止(Process Termination)

當作業系統中的每一個行程(process)在完成其工作後，都必須正確終止，以釋放系統資源並維持系統穩定性。當行程完成執行時，會主動或被動地呼叫系統呼叫 exit()。exit()具有以下功能：

- 通知作業系統該行程已結束執行
- 釋放該行程所使用的資源
- 傳回結束狀態(exit status)給其父行程(parent process)

```
int status;
pid_t pid = wait(&status);   // status 儲存子行程的結束狀態
```

7.4 父行程如何得知子行程已結束：wait()

作業系統提供 wait() 系統呼叫，讓父行程可以等待子行程結束。一個行程呼叫 wait() 後會暫停執行，直到：

1. 某個子行程終止
2. 系統把該子行程的 exit code 傳給父行程

這種設計可以讓父行程知道子行程是否順利完成工作，或是否出錯。

8. 殭屍行程(Zombie Process)

在 UNIX / Linux 系統中，一個子行程執行完畢後，通常會呼叫 exit() 表示「任務完成」，並將控制權交還給作業系統。理論上，它應該從系統中完全消失，但實際上並不是立刻清除。

8.1 為什麼會出現殭屍行程

當子行程呼叫 exit() 結束時，作業系統會：

1. 先暫時保留該行程的資訊（如 PID、exit code）在 process table 中。
2. 等待父行程使用 wait() 系統呼叫來查詢與回收這些資訊。

如果父行程有正常呼叫 wait()，子行程的資料就會被清除，這稱為「回收」子行程。但如果父行程從頭到尾都沒呼叫 wait()，那麼這個子行程就會「身體不見了，但在系統中還留有一點殘留資料」，變成所謂的 殭屍行程(Zombie Process)。

8.2 如何避免殭屍行程？

- 設計良好的父行程邏輯：父行程在適當時機呼叫 wait() 或 waitpid()，確保回收所有子行程。
- 讓 init（或 systemd）接管孤兒行程：若父行程提早終止，子行程會被系統接管。這些「孤兒行程」由 init 接手後，會負責正確回收，避免留下殭屍。
- 使用非同步處理機制：在某些情況下，可以使用 signal(SIGCHLD) 搭配處理函數，自動在子行程結束時執行回收動作。

9. 行程間通訊(Interprocess Communication, IPC)

電腦系統中，絕大多數情況下不是只有一個程式在跑。多個行程（process）同時運作，它們可能各做各的，也可能需要彼此配合、傳遞資料。這就產生了「行程間通訊（IPC）」的需求。我們可以將行程分成兩類：

- 獨立行程(Independent)：各做各的，互不干涉
- 合作行程(Cooperating)：互相合作，分享資料

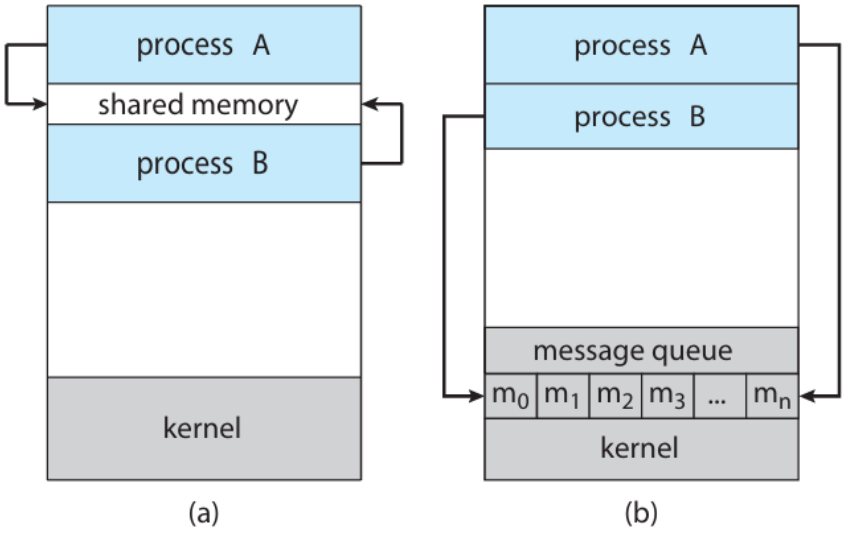
需要行程溝通的原因如下：

- 資訊共享：多個應用程式可能要共用資料(例如：剪貼簿內容)
- 加速計算：把大任務分成小任務，讓多核心同時執行(平行處理)
- 模組化設計：系統設計時，把功能拆分成多個行程或模組

9.1 IPC 的兩大方式

- Shared Memory(共享記憶體)：建立一塊記憶體區塊，讓多個行程能同時存取那塊記憶體
- Message Passing(訊息傳遞)：行程之間用「傳送訊息」的方式來溝通，就像傳紙條

Figure 7：Shared Memory(左)與 Message Passing(右)



9.2 訊息傳遞(Message Passing)

Message Passing 是透過 send() 和 receive() 這兩個動作來傳遞資料適合分散式系統 (例如不同電腦之間的行程)。訊息可以是固定大小(簡單實作但限制多)或變動大小(程式好寫但系統要多處理)。

```
send(message);      // 傳送訊息
receive(message);   // 接收訊息
```

在 Message Passing 中有三大設計層面要考慮：

第一， **命名方式(Naming)**：行程之間怎麼「知道要傳給誰」。依據 Naming 可以區分為：

1. 直接命名(Direct Communication)：一對一通訊(每一個 link 只給一對行程)，傳送和接收者必須知道對方的名字

```
send(P, msg)：傳訊息給 P
receive(Q, msg)：從 Q 收訊息
```

2. 間接命名(Indirect Communication)：使用「Mailbox / Port(信箱)」作為中介物件
   - 多個行程可以共用一個 mailbox
   - Mailbox 可以由 process 擁有(會隨 process 終止)或由作業系統管理(獨立存在)

```
send(A, msg)：傳送給信箱 A
receive(A, msg)：從信箱 A 收訊息
```

第二，**同步方式(Synchronization)**：當 send() 和 receive() 呼叫時，會不會「等對方」？

當 send() 和 receive() 都是 blocking，稱為 rendezvous(會合點) → 雙方等彼此，成功才繼續。

	Blocking	Non-blocking
Send	傳送後卡住，等對方收到才繼續	傳送後就繼續跑，不管對方有沒有收到

Receive	沒收到就卡住，一直等	嘗試接收，有的話就收，沒有就回傳 null

第三，**緩衝區設計(Buffering)：**即使是 message passing，訊息在送達之前也要有地方暫存，即為 buffer(緩衝區)

類型	說明
0 容量(Zero Capacity)	緩衝區不能存訊息 → `send()` 會卡住，直到 `receive()` 把訊息收走
有限容量(Bounded Capacity)	緩衝區最多可存 `n` 筆訊息，如果滿了 → `send()` 會等
無限容量(Unbounded Capacity)	想送幾筆就送幾筆，`send()` 永遠不會卡住(理論上)