作業系統：Processes

目錄

1. Process Concept

Processu 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。Process 本身涵蓋內容：

- 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。

Process 與 Program 的差異：

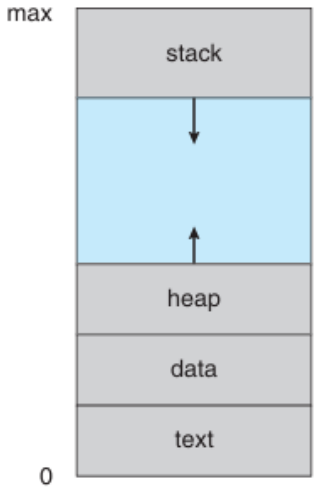
- Program 是寫好的程式，儲存在硬碟上
- Process 是執行中版本的程式，具有活躍狀態與系統資源。
- 一個程式可以對應多個 process(例如你開了兩個 Google Chrome 視窗，每個就是一個 process)。

2. Process 結構

Process 結構可以分為：

- Text：儲存程式的機器碼，固定不變。
- Data：全域變數，有些已初始化、有些未初始化。
- Heap：用 malloc、new 等語法動態配置的記憶體，可擴張縮小。
- Stack：呼叫函式時會壓入資料(例如參數、return 位置)，函式結束後再彈出。

Figure 1：process 架構



3. Process 狀態(Process States)

每個 process 執行期間都會經歷不同「狀態」:

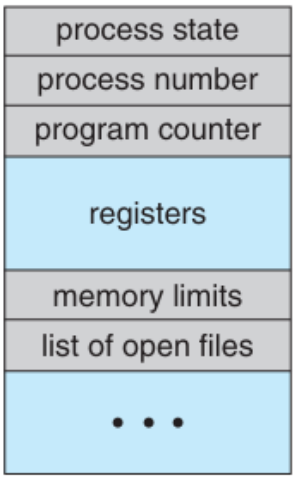
- New：剛被建立，尚未執行。
- Ready：等著 CPU 來執行。
- Running：目前正在 CPU 上執行。
- Waiting：暫時被擱置，等待某事件(如 I/O)完成。
- Terminated：執行完畢，被結束。

4. 行程控制區塊(Process Control Block, PCB)

系統中每個 process 都會用一個資料結構紀錄它的所有資訊，稱為 PCB，像是 process 的身份證。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)

行程排程的目標：

- 多工(Multiprogramming)：確保 CPU 永遠有事做，提高 CPU 使用率。
- 分時系統(Time Sharing)：讓多個使用者看起來像是同時操作電腦，靠頻繁地切換 process 來實現。

而核心任務為：

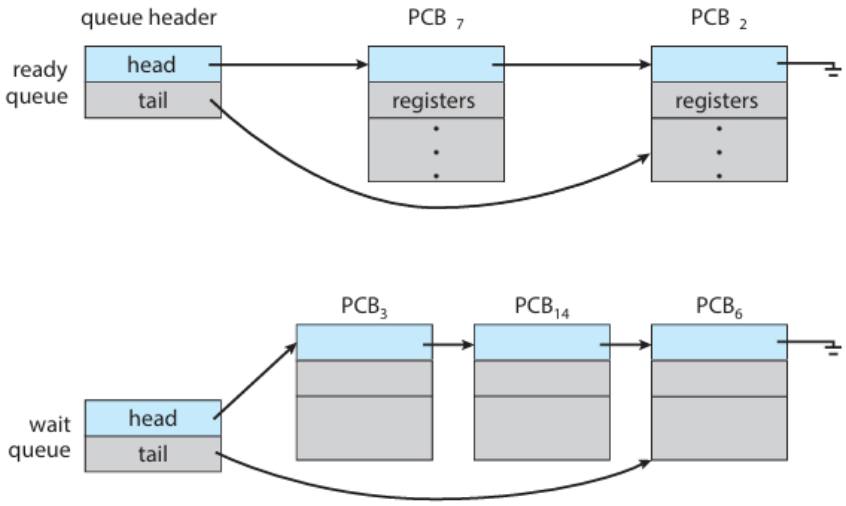
- 使用排程器(scheduler) 從 ready queue(就緒佇列) 選出一個 process，分配給 CPU 執行。
- 每個 CPU 核心同一時間只能執行一個 process。
- 在多核心系統中，每核心可執行一個 process；其餘的就排隊等著。

排程佇列(Scheduling Queues)可以分為以下三種：

Table 1：排程佇列(Scheduling Queues)

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)

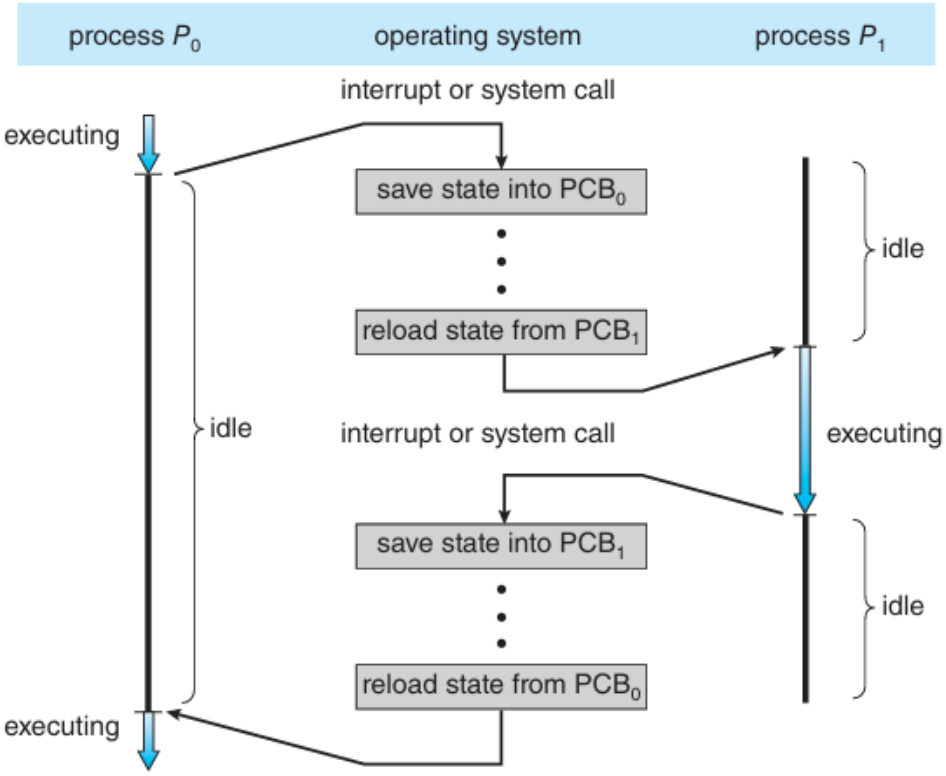
當 process 被中斷或 CPU 要切給別的 process 時，會：

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

這邊是 Context switch。

Context switch 是「額外成本」，因為切換時系統沒有實際做正事，切換速度會受硬體影響(記憶體速度、register 數量、是否有專門指令支援)。

Figure 4：Context switch 的過程示意圖

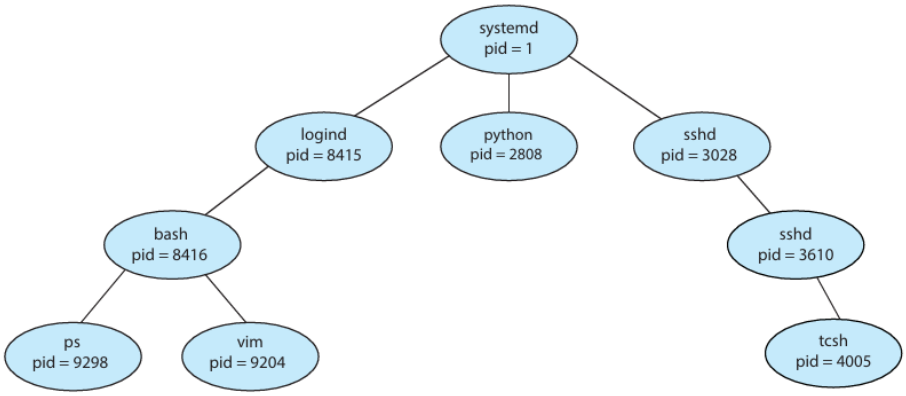


7. 行程操作(Operations on Processes)

7.1 行程建立(Process Creation)

父行程(Parent Process)可以建立子行程(Child Process)。子行程也可以建立其他行程，這樣就會形成「行程樹(Process Tree)」。每個行程都有一個 pid(process ID)

Figure 5：Process Tree



7.2 fork() 與 exec()(UNIX 系統)

- fork()：複製一個行程(父行程被複製，變成一模一樣的子行程)
- exec()：把目前行程的程式碼，換成別的程式(例如 ls)

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)

在作業系統中，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
```

父行程如何得知子行程已終止？

作業系統提供 wait() 系統呼叫，讓父行程可以主動等待其子行程的結束狀態。

使用 wait() 的行程會暫停執行，直到某個子行程終止為止，並能取得其結束碼(exit code)作為後續判斷依據。

7.4 殭屍行程(Zombie Process)

在 UNIX / Linux 系統中，當一個子行程執行完畢後，會呼叫 exit() 結束自己。但此時，作業系統不會馬上完全刪除這個行程的資訊，而是會在 process table(行程表)中暫存該行程的結束狀態(如 exit code)，以便父行程(parent)稍後使用 wait() 系統呼叫來查詢與回收資源。如果父行程沒有呼叫 wait() 來回收這些資訊，那麼這個已經結束的子行程就會變成所謂的殭屍行程(Zombie Process)。

殭屍行程的特徵：程式已結束，不再佔用記憶體或執行資源，但其 PID(Process ID)與結束狀態仍暫存在 process table 中。

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

在一個系統中，有很多行程(process)同時執行，這些行程可能需要：

- 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

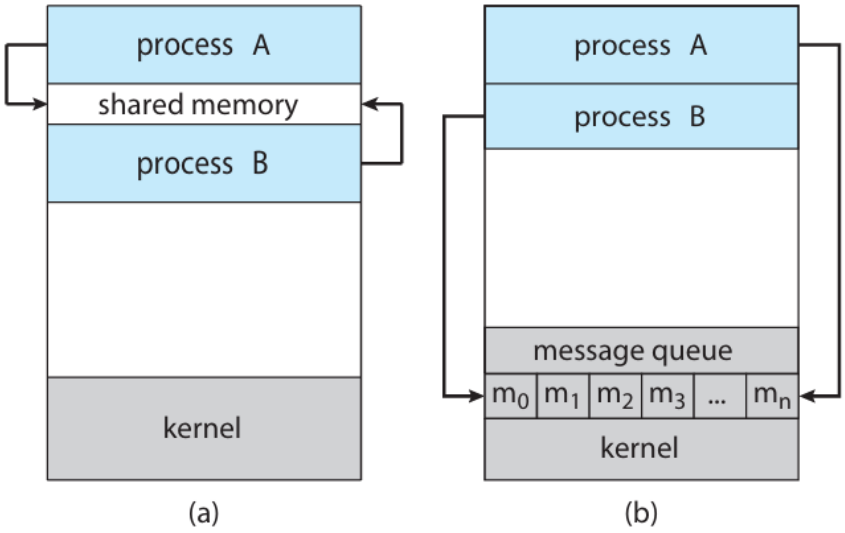
其動機，是為了：

- 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

而 IPC 的方式，可以分為

- 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)



8.1 訊息傳遞(Message Passing)

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)