Chapter 3

行程觀念

章節目標

- 識別行程的各個組成部份，並說明它們如何在作業系統中表示和排班
- 描述如何在作業系統中創建和終止行程，包括使用的適當系統呼叫執行這些操作來開發應用程式
- 描述並對比使用共用記憶體行程間通信和訊息傳遞
- 設計程式使用管道和 POSIX 共用記憶體以執行行程間通信
- 描述使用插座和遠程程序呼叫的客戶端—伺服器通信
- 與 Linux 作業系統交互的設計的核心模組

行程觀念

- 早期的電腦系統在一段時間內只允許一個程式執行。這個程式對於系統有完全的控制權，因此可以使用所有的系統資源
  - 相反地，今日的電腦系統允許許多程式載入到記憶體中，並且同時地執行
- 這種演進使得系統必須對不同的程式做更嚴謹的控制以及更精確的區分
  - 這項需求產生**行程** (process) 的觀念，行程就是一個執行的程式
- 在現在的計算系統中，行程是基本的工作單元

行程觀念

- 作業系統越複雜，越期望能站在使用者的立場
- 雖然作業系統主要的考慮是執行使用者的程式，但它同時也需要執行部份不屬於核心的系統工作，這些工作最好在使用者空間中完成，而不是在核心中
- 因此，作業系統是由一組行程所組成：
  - 執行系統程式碼的作業系統行程，和執行使用者程式碼的使用者行程
- 這些行程都可以並行執行，而 CPU 則在它們之間以多工的方式執行
- 在本章中，你將會讀到什麼是行程以及它們是如何工作

3.1 行程的觀念

- 討論作業系統的一個問題就是該如何稱呼 CPU 所有的運作項目
  - 早期的電腦是整批式系統執行工作 (job)，接著出現分時系統執行**使用者程式** (user program)
    - 或稱為**任務** (task)
  - 即使在單一使用者系統，使用者仍可同時執行數個程式：
    - 一個文書處理程式、網頁瀏覽器及 e-mail 套件程式

3.1 行程的觀念

- 就算是使用者只能一次執行一個程式
  - 例如在不支援多功的嵌入式系統上，作業系統仍須支援其內部一些工作，比方說是記憶體管理
- 從許多方面看來，這些所有的活動都類似，所以我們就稱之為**行程** (process)

3.1.1 行　程

- 行程指的是正在執行的程式
  - 一個行程當前活動狀態代表目前運作的**程式計數器** (program counter) 數值和處理器的暫存器內容
  - 行程的記憶體配置通常分為多個部份，如圖 3.1 所示。這些部份包括：
    - **文本區** (text section)—可執行程式碼
    - **資料區** (data section)—全域變數
    - **堆積區** (heap section)—記憶體在程式運行時動態分配
    - **堆疊區** (stack section)—當呼叫函數時臨時的資料儲存 (例如函數參數、返回位址，以及區域變數)

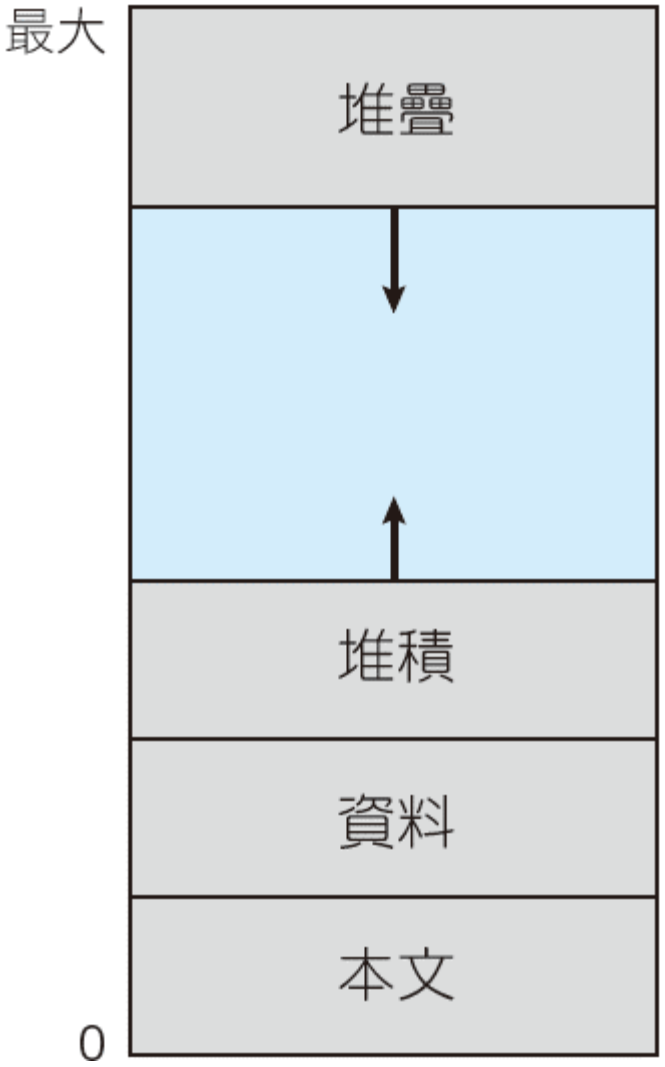
3.1.1 行　程

- 程式是一項被動 (passive) 的個體，就像儲存在磁碟內包含一系列指令的檔案
  - 通常稱為**可執行檔案** (executable file)
  - 然而行程卻是一項主動 (active) 的個體，它具備程式計數器來指明下一個執行的指令，以及一組相關的資源
    - 當可執行檔案載入記憶體時，程式變成行程
    - 載入可執行檔案的兩種方法是快按兩下可執行檔案的圖示，以及在命令列輸入可執行檔案檔名 (如 prog.exe 或 a.out)

3.1.1 行　程

- 雖然兩項行程可能是執行一個相同的程式，但它們絕不能視為兩組不相關的執行順序
  - 譬如，有幾個使用者在執行同一個郵寄程式，或是同一個使用者可能執行許多份網頁瀏覽程式
  - 以上的每一個都是一個單獨的行程，而雖然這些行程的本文區域相同，但是資料堆積、堆疊區域卻不相同
  - 一個行程在執行時複製出許多份和自己相同的行程也是常見的
  - 這些問題在 3.4 節將進一步地討論

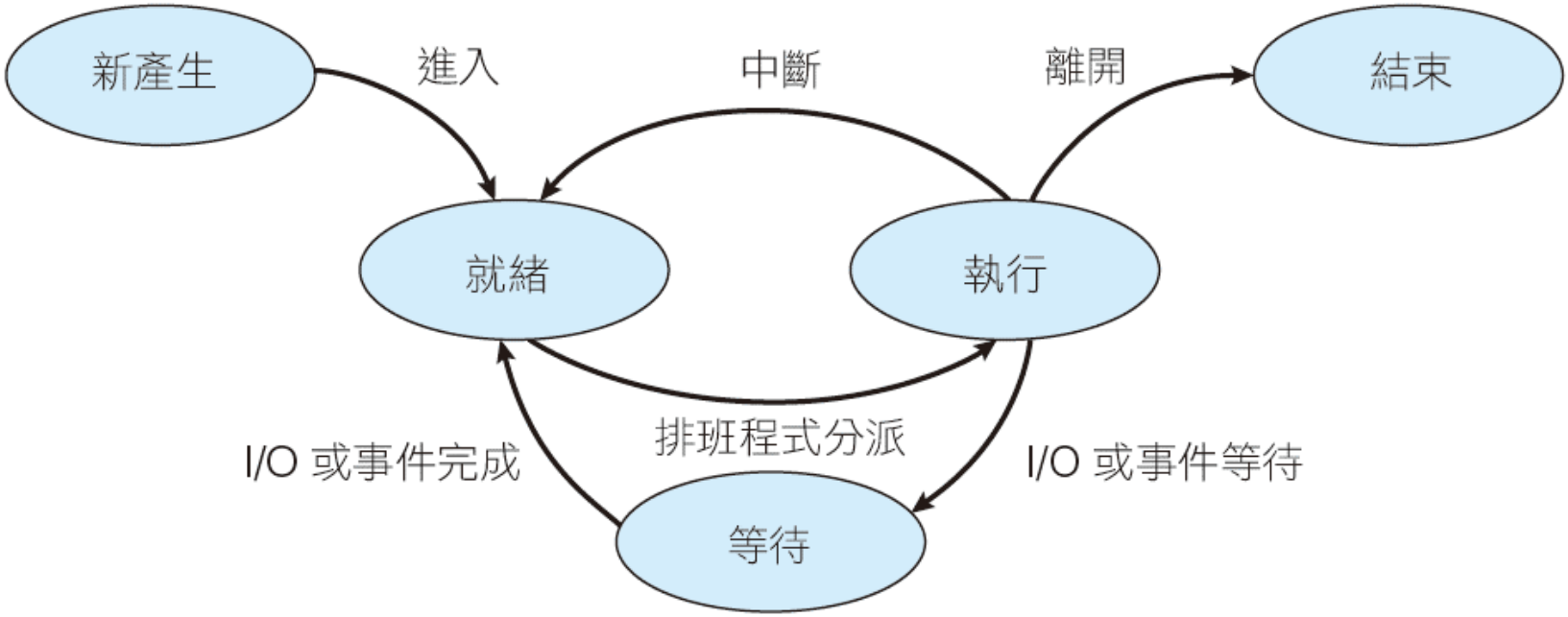
圖 3.1　行程在記憶體中的配置



3.1.2 行程狀態

- 當行程執行時，它會改變其狀態。行程的**狀態** (state) 部份是指該行程目前的動作，每一個行程可能會處於以下數種狀態之一：
  - 新產生 (new)：該行程正在產生中
  - 執行 (running)：指令正在執行
  - 等待 (waiting)：等待某件事件的發生 (譬如輸入/輸出完成或接收到一個信號)
  - 就緒 (ready)：該行程正等待指定一個處理器
  - 結束 (terminated)：該行程完成執行

圖 3.2　行程狀態圖



3.1.3 行程控制表

- 每一個行程在作業系統之中都對應著一個**行程控制表** (process control block, PCB)
  - 或稱**任務控制表** (task control block)
- 一個行程控制表 (PCB) 如圖 3.3 所示

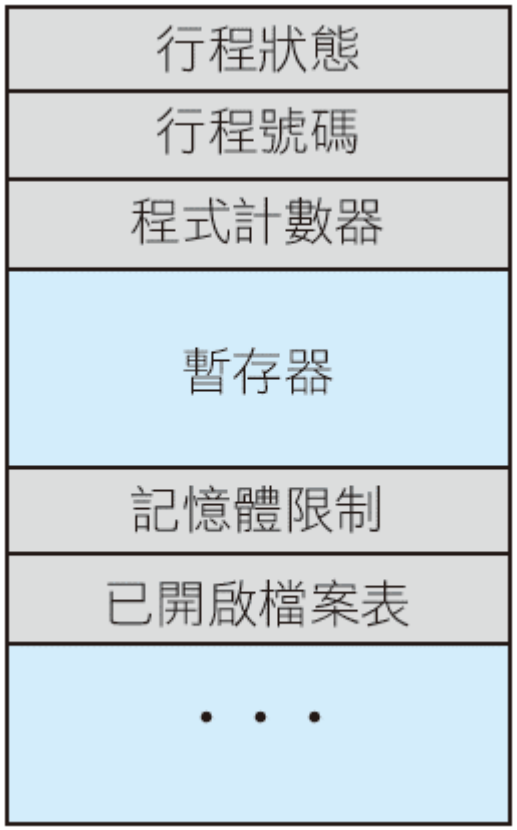


圖 3.3 行程控制表 (PCB)

3.1.3 行程控制表

- 它記載所代表的行程之相關資訊包括：
  - 行程狀態：
    - 可以是 new、ready、running、waiting 或 halted 等
  - 程式計數器：
    - 指明該行程下一個要執行的指令位址
  - CPU 暫存器：
    - 其數量和類別，完全因電腦架構而異
      - 包括累加器 (accumulator)、索引暫存器 (index register)、堆疊指標 (stack pointer) 以及一般用途暫存器 (generalpurpose register) 等

3.1.3 行程控制表

- 還有一些狀況代碼 (condition-code)

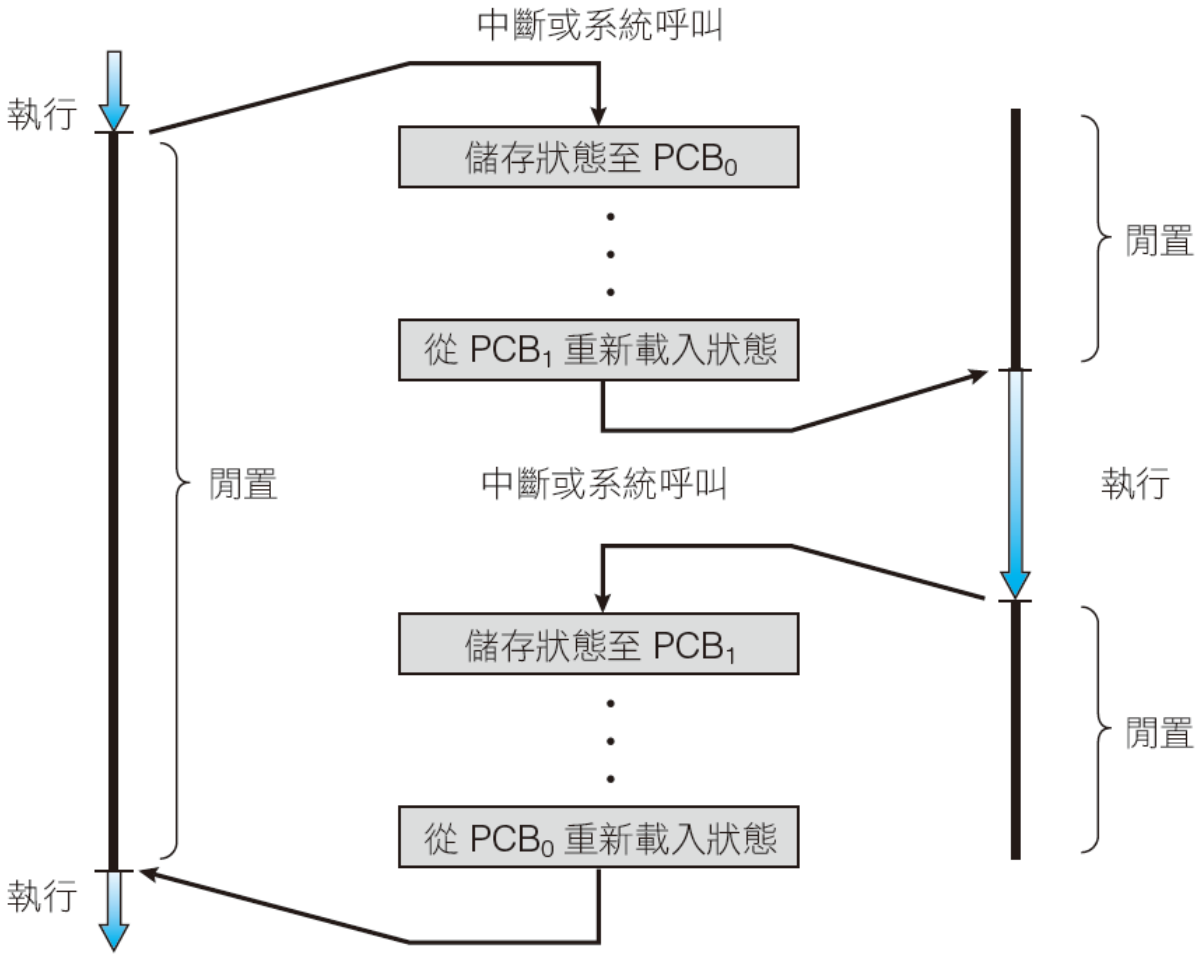
- 當中斷發生時，這些狀態資訊以及程式執行計數器必須儲存起來，以便稍後利用這些儲存的資訊，使程式能於中斷之後順利地繼續執行

- CPU 排班法則相關資訊：
  - 包括行程的優先順序 (priority)、排班佇列 (scheduling queue) 的指標，以及其它的排班參數
- 記憶體管理資訊：
  - 這些資訊包括如基底暫存器 (base register)、限制暫存 (limit register) 和分頁表 (page table) 數值的資訊，或根據作業系統所使用的記憶系統區段表 (segment table)

3.1.3 行程控制表

- 會計資訊：
  - 包括 CPU 和實際時間的使用數量、時限、帳號、工作或行程號碼等等
- 輸入/輸出狀態資訊：
  - 包括配置給行程的輸入/輸出裝置，包括開啟檔案的串列等等

圖 3.6　CPU 在行程之間內容轉換



3.1.4 執行緒

- 到目前為止，討論過的行程模式代表一個行程為執行單一**執行緒** (thread) 的程式
  - 例如，如果一個行程正在執行一個文書處理程式，則有一個單一執行緒的指令被執行
  - 這種單一執行緒的控制只允許行程一次執行一個任務
  - 因此，使用者無法在相同的行程同時打字及進行拼字檢查
  - 許多近代作業系統已擴展行程觀念，允許行程執行多個執行緒，因此允許行程一次完成一個以上的任務緒

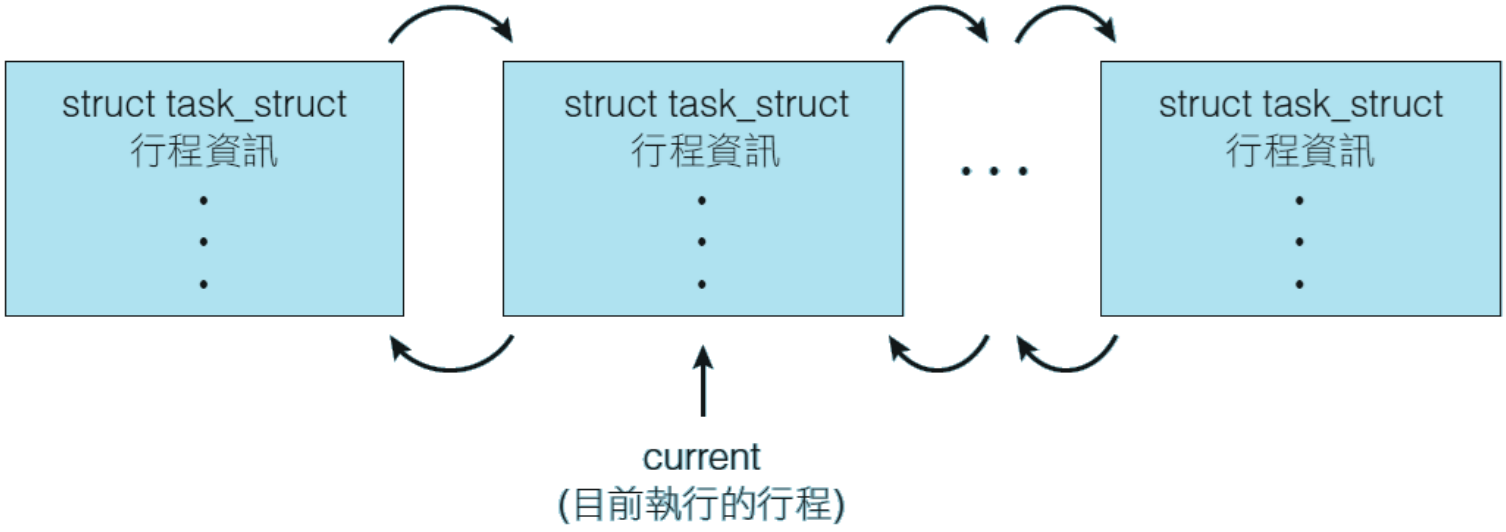
3.1.4 執行緒

- 此功能在多核心系統特別有利，因為多執行緒可以並行地執行
  - 例如，多執行緒文字處理器可以分配一個行程來管理使用者輸入，而另一個行程運行拼字檢查
  - 在支援執行緒的系統上，PCB 擴展到包含每一執行緒的資訊
  - 整個系統中也需要其它的變更以支援執行

LINUX 的行程表示

- Linux 作業系統的行程控制區塊是以 C 結構 task_struct 表示

```
long state;                     /* state of the process */
struct sched_entity se;         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space */
```



3.2 行程排班

- 多元程式規劃的目的，是隨時保有一些行程在執行，藉以最大化 CPU 的使用率
- 分時系統的目的是將 CPU 核心在不同行程之間不斷地轉換，以便讓使用者可以在行程執行時和每個程式交談
- 為了達到這些目的，**行程排班程式** (process scheduler) 為 CPU 核心上執行程式選擇一個可用的行程 (可能是從一些可用的行程中)
- 每個 CPU 核心一次可以運行一個行程

3.2 行程排班

- 當行程進入系統時，它們是放在**就緒佇列** (ready queue) 之中，且在 CPU 的核心上就緒等待執行
  - 這個佇列一般都是用鏈結串列的方式儲存
  - 在就緒佇列前端保存著指向這個串列的第一個 PCB 的指標，而每個 PCB 中都有一個指向就緒佇列中下一個行程的指標

3.2 行程排班

- 作業系統中還有其它的佇列
  - 當某行程配置到 CPU 核心，它會執行一段時間，並且最後會停下來、被中斷或等待某一特殊事件發生
    - 譬如一項 I/O 要求的完成
  - 假設行程對共用裝置提出 I/O 要求，譬如磁碟機
    - 因為裝置的運行速度明顯慢於處理器，則該行程必須等待 I/O 可用
    - 等待某個事件發生的行程
      - 例如 I/O 的完成，都會放置在等待佇列 (wait queue) 中 (圖 3.4)

圖 3.4　就緒佇列和等待佇列

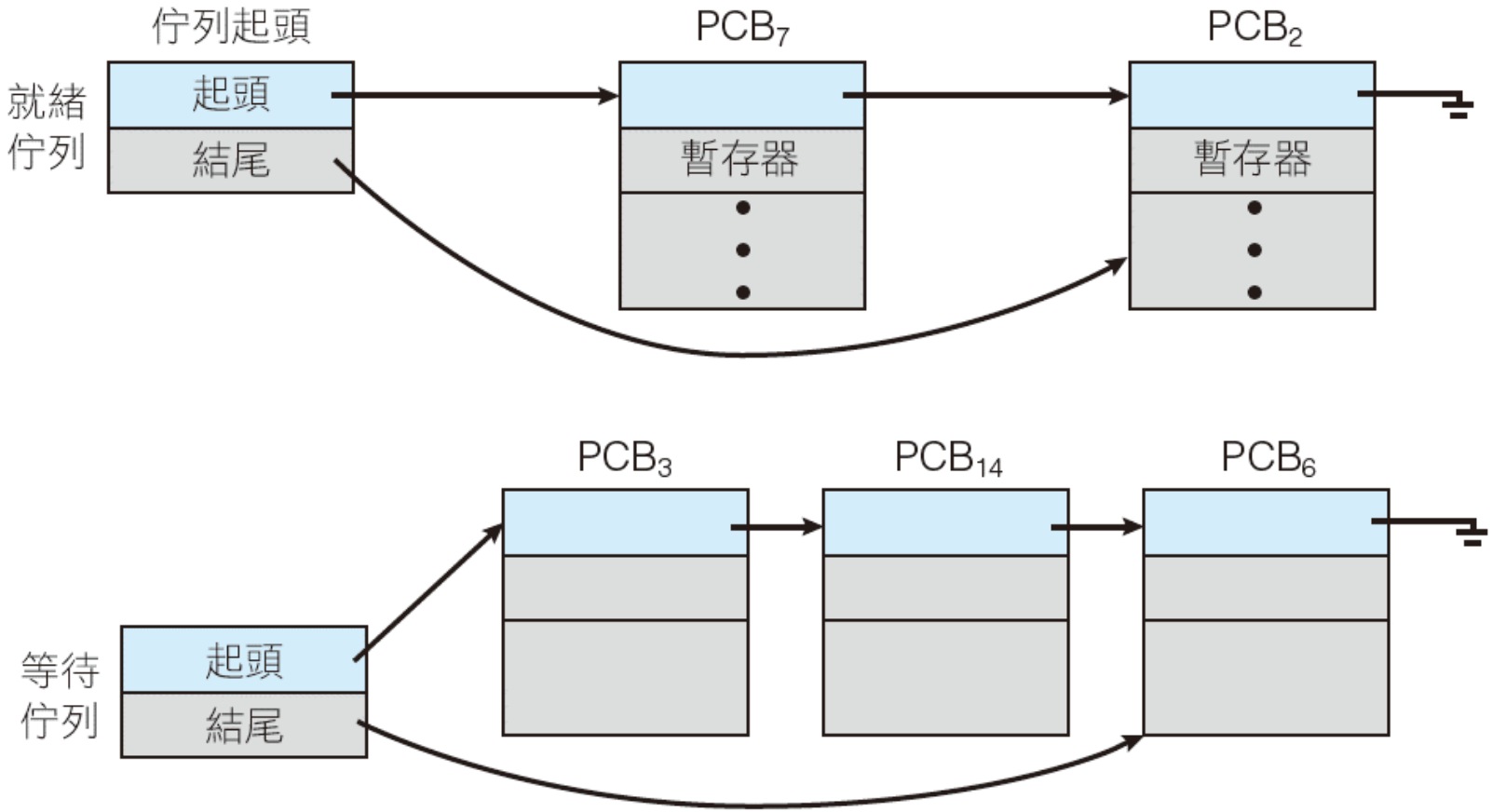
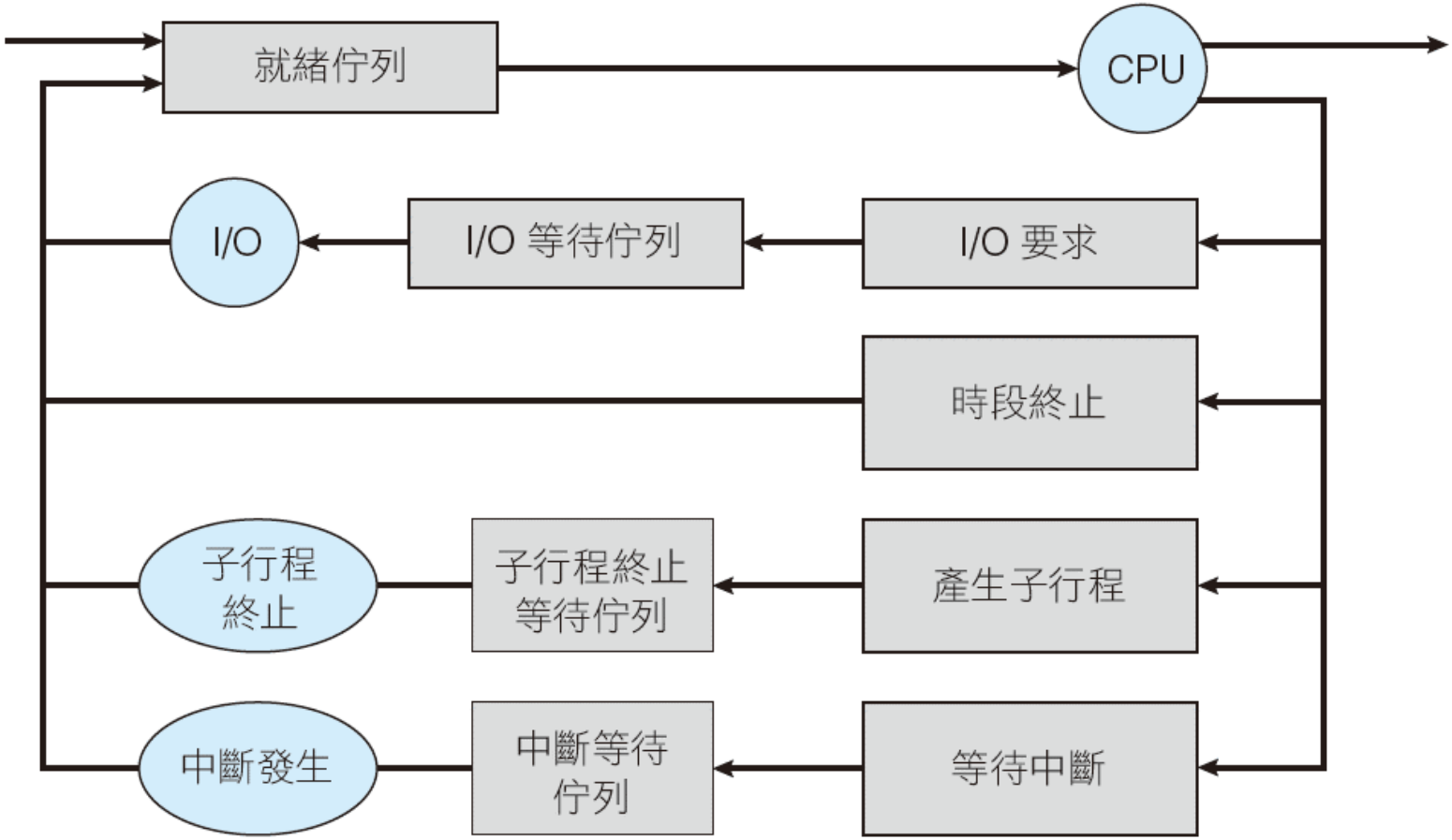


圖 3.5　行程排班的佇列圖表示



3.2 行程排班

- 在單一 CPU 核心系統裡，不可能有一個以上的行程同時執行
  - 多核心系統一次可執行多個行程
  - 如果有多個行程超過核心數量，剩餘的行程將會等待核心空閒並可以重新排程
  - 當前在記憶體中的行程數稱為**多元程式規劃的程度** (degree of multiprogramming)

3.2 行程排班

- 平衡多元程式和時間共享的目標還需要考慮行程的一般行為
  - 大多數行程可以描述為 I/O 傾向或 CPU 傾向
  - **I/O 傾向的行程** (I/O-bound process) 是花費在執行 I/O 上的時間多於在計算上花費的時間
  - 相較之下，**CPU 傾向的行程** (CPU-bound process) 使用更多時間在進行計算，因此很少發生 I/O 的要求

3.2 行程排班

- 一個行程在它整個生命期裡將在就緒佇列和不同的等待佇列間遷移
  - **CPU 排班器** (CPU scheduler) 的作用是從就緒佇列中的行程中進行選擇，並將 CPU 核心分配給它們的其中一個
  - CPU 排班器必須頻繁為 CPU 配置新行程
  - 等待 I/O 請求之前，I/O 傾向的行程可能會執行幾毫秒
  - 儘管受 CPU 傾向的行程將需要更長的 CPU 核心時間，但排班器不太可能將核心給予該行程更長的時間

3.2 行程排班

- 取而代之的是，它可能將行程從 CPU 中強制刪除並安排另一個行程執行
- 因此，CPU 排班器每 100 毫秒至少執行一次，儘管通常更為頻繁

3.2 行程排班

- 有些系統採用一種額外的、間接方式，稱為**置換** (swapping) 來排班
  - 它的主要觀念就是有時候可以將行程從記憶體中有效地移開 (並且從對 CPU 的競爭中移開)，藉此降低多元程式規劃的程度
  - 稍後，再把該行程放回記憶體中，並且放在它移開之前的位置上繼續執行
  - 這種方法稱為置換，因為行程可以從記憶體到磁碟 (它目前的狀態儲存處) 被“置換出去”，並且稍後從磁碟到記憶體 (它的狀態還原處) 再“置換進來”
  - 只有當記憶體被過量使用且必須被釋放掉時，置換才有需要

移動系統的多工

- 因為加在行動裝置的限制，早期版本的 iOS 沒有提供使用者應用程式的多工；只有一個應用程式在前台執行，其它的使用者應用程式被暫停。作業系統的任務是多工，因為它們是 Apple 公司寫的，且表現良好
- iOS 4 開始，Apple 為使用者應用程式提供有限程度的多工，因此允許一個單一前台應用程式和多個背景應用程式同時執行
  - 在行動裝置上，**前台** (foreground) 應用程式是目前開啟和顯示在螢幕的應用程式。**背景** (background) 應用程式保留在記憶體中，但不會占據顯示螢幕
- iOS 4 程式 API 提供多工的支援，因此允許行程在背景執行而不會被暫停
- 這是受到限制，並且只有有限的應用程式型態是可行的

移動系統的多工

- 作為硬體用於行動設備開始提供更大的儲存容量、多個處理核心，和更大的電池壽命，iOS 的後續版本開始支持更豐富的功能
  - 例如，iPad 平板電腦上的較大螢幕允許同時執行兩個前台應用程式，這就是一種稱為**分割畫面** (split-screen) 的技術
- 從最初開始，Android 就支援多工處理，也未對應用程式可以在背景中執行的型態加以限制
- 如果應用程式在背景下需要處理時，就必須使用一個**服務** (service)，服務是一個代表背景行程執行單獨的應用元件
- 考慮一個串流的音樂應用程式：
  - 如果應用程式移到背景，此服務繼續代表背景應用程式送出音樂檔案到語音裝置驅動程式

移動系統的多工

- 事實上，即使背景應用程式被暫停此服務也會繼續執行
- 服務沒有使用者介面，只有少量記憶體，因此對行動環境的多工提供有效技巧

3.2.3 內容轉換

- 如 1.2.1 節所提，中斷使作業系統改變 CPU 目前的工作而執行核心常式，這樣的作業常發生在一般用途系統上
  - 當中斷發生時，系統需要儲存目前在 CPU 上執行行程的**內容** (context)，所以當作業完成時，它可以還原內容，本質就是暫停行程，再取回行程
  - 行程內容以行程 PCB 表示，包含 CPU 暫存器的數值，行程狀態 (如圖 3.2) 和記憶體管理訊息
  - 一般而言，無論在核心模式或使用者模式，我們執行目前 CPU 狀態的**狀態儲存** (state save)，然後**還原狀態** (state restore) 來恢復作業

3.2.3 內容轉換

- 轉換 CPU 核心至另一項行程時必須執行目前行程的狀態儲存，並執行一不同行程的狀態復原
  - 這項任務稱為**內容轉換** (context switch) 並於圖 3.6 中說明
    - 當內容轉換發生時，核心在它的 PCB 儲存舊行程的內容以及載入被排班之新行程的儲存內容來執行
  - 內容轉換所花費的時間純粹是額外的浪費，因為此時系統所做的並不是有用的工作

3.2.3 內容轉換

- 內容轉換的速度隨著電腦而有不同，因為這必須由記憶體速度、複製的暫存器數目，以及是否有特殊指令來決定
  - 譬如載入或儲存所有暫存器的單一指令
- 一般而言，它的速度在幾微秒 (microsecond)

- 內容轉換的時間長短大多取決於硬體支援的程度
  - 例如，有些處理器提供多組暫存器，因此內容轉換時，只需將指標指到目前的暫存器組
  - 當然，如果行程的數目多於暫存器組，系統仍需使用前面所述的方法，將暫存器組搬進及搬出記憶體

3.2.3 內容轉換

- 而且作業系統越複雜，在內容轉換時所做的工作就越多
- 進階的記憶體管理技術在每次做內容轉換時，必須搬移更多的資料
  - 譬如，目前行程的位址空間 (address space) 必須儲存起來，以作為下一個準備執行行程的位址空間
- 至於位址空間如何儲存以及需要多少工作量來儲存它，則取決於作業系統的記憶體管理方法

3.3 行程的操作

- 系統中的各個行程可以並行 (concurrently) 地執行，而且也要能動態地產生或刪除
  - 因此，作業系統必須提供行程產生和結束的功能
- 在一個行程的執行期間，它可以利用產生行程的系統呼叫來產生數個新的行程
  - 原先的行程就叫作父行程，而新的行程則叫作子行程
  - 每一個新產生的行程可以再產生其它的行程，這可以形成一個行程**樹** (tree)

3.3 行程的操作

- 大部份作業系統 (包括 UNIX、Linux 及 Windows) 依據唯一的**行程識別碼** [(process identifier) 或 **pid**] 來識別行程，通常行程識別碼是一個整數
  - pid 為系統每一個行程提供一個獨一無二的數字，它可以被用來作為索引值來存取核心內行程的各種屬性
- 一般而言，當一個行程產生另一個子行程時，這一個子行程將需要某些資源 (CPU 時間、記憶體、檔案、I/O 裝置) 才能完成其任務
  - 子行程或許能夠直接從作業系統取得所需要的資源，或是它可能受限於只能使用其父行程所擁有的部份資源

3.3 行程的操作

- 對於父行程而言，它可能必須將所有的資源分配給所有的子行程，或是和其子行程之間共用某些資源
  - 例如記憶體或檔案
- 若是限制子行程只能使用父行程的部份資源，則可避免一般行程在產生太多子行程時增加整個系統過重的負擔

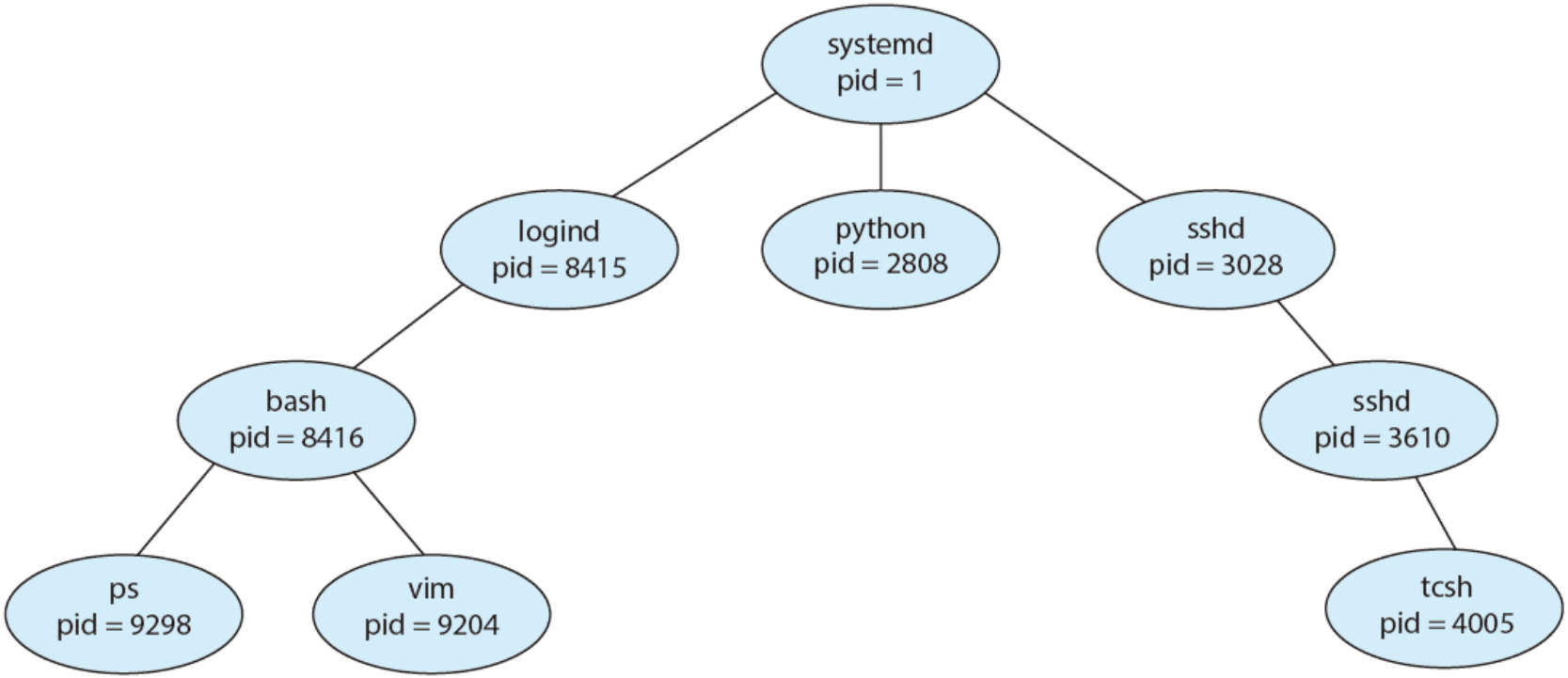
3.3 行程的操作

- 父行程除了提供不同的實體及邏輯資源，它可能要同時傳送起始資料 (輸入) 給子行程
  - 舉例來說，有一個行程的功能是把檔案 hw1.c 的內容顯示在終端機的螢幕上
  - 當這個行程產生時，它就會得到檔名 hw1.c，這就好像從它的父行程接收到輸入一樣
    - 這個行程將使用檔名、開啟檔案和輸出內容
    - 它也可能得到輸出裝置的名稱
  - 有些作業系統會傳遞資源給子行程
  - 在這種作業系統下，新產生的行程就有兩個開啟檔案：hw1.c 和終端機裝置，接下來只需要在兩者之間傳遞資料即可

3.3 行程的操作

- 當一個行程產生一個新的行程時，在執行作法上有兩種：
  1. 父行程繼續執行而子行程也同時執行
  2. 父行程等著它的所有子行程中止後才繼續執行
- 有兩種使用新行程位址空間的可能方法：
  1. 子行程是父行程的複製品
     - 子行程有父行程相同的程式和資料
  2. 子行程有一個程式載入其中

圖 3.7　典型的 Linux 系統的行程樹



3.3 行程的操作

- 為了說明這兩種不同的作法，讓我們先以 UNIX 作業系統為例
  - 在 UNIX 之中，每項行程以其獨特的整數行程識別碼來區別
  - 一項新的行程可以 fork() 系統呼叫來產生。新的行程有原行程位址空間的一份拷貝
  - 此方法使父行程可以很容易地與其子行程聯繫
  - 父行程和子行程同時從 fork() 指令的下一列繼續執行，其間只有一點差異
    - 子行程的 fork() 傳回碼為 0，而新行程的行程識別碼 (非 0) 則會傳回給原父行程

3.3 行程的操作

- 通常這兩個行程之一會在 fork() 指令之後使用系統呼叫 exec()，以便載入新的程式來替換掉原先行程的記憶體
  - 系統呼叫 exec() 載入一個二進制檔案到記憶體中並銷毀包含系統呼叫 exec() 程式的記憶體映像，然後就自動開始執行
  - 使用這種方式，這兩個行程可以互相溝通，然後各自執行
  - 父行程還可以再產生更多個子行程；或是如果在子行程執行時，父行程沒事可做，那麼它可以執行系統呼叫 wait()，把它自己由就緒佇列移出，並等到子行程結束才繼續執行

3.3 行程的操作

- 因為呼叫 exec() 後新的程式覆蓋掉行程的位址空間，而呼叫 exec() 不會歸還控制權，除非錯誤發生

圖 3.8 使用 UNIX fork() 系統呼叫產生獨立行程

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```


圖 3.10 使用 Windows API 產生個別行程

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
      fprintf(stderr, "Create Process Failed");
      return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

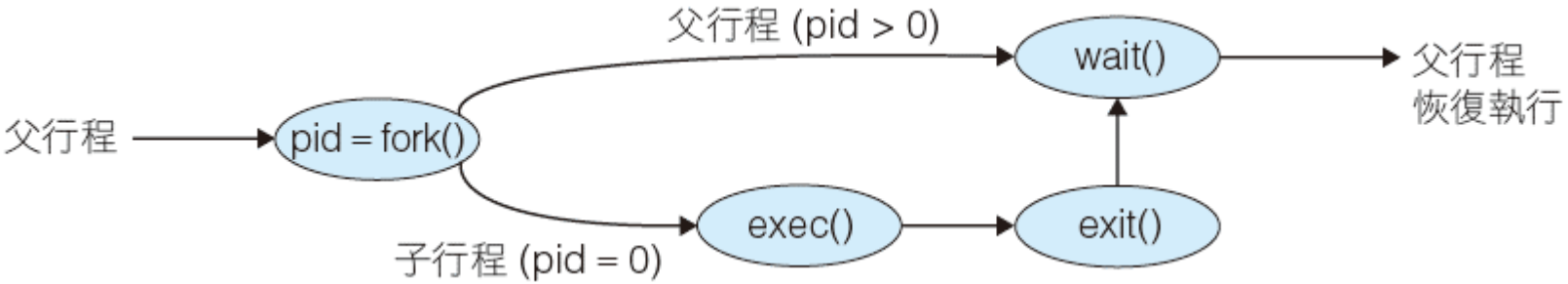
3.3 行程的操作

- 圖 3.8 所示 C 程式說明先前描述的 UNIX 系統呼叫
- 現在有兩個不同的行程正在執行相同程式的拷貝
- 唯一的差別是，子行程的 pid 值為 0，父行程則為大於 0 的整數值
  - 事實上，是子行程的實際 pid 值
- 子行程從父行程繼承特權與排班屬性，以及特定的資源
  - 例如開啟的檔案
- 子行程使用系統呼叫 execlp() 來執行 UNIX 指令 /bin/ls (用來得到目錄列) (execlp() 是 exec() 系統呼叫的一個版本) 以覆蓋它的位址空間

3.3 行程的操作

- 父行程以 wait() 系統呼叫等待子行程完成
- 當子行程完成之後 (藉由暗地或明確啟動 exit())，利用 exit() 系統呼叫，而父行程由呼叫 wait() 處再繼續，說明如圖 3.9
- 當然，無法阻止子行程不呼叫 exec()，並繼續執行父行程的拷貝
- 在此情況下，父行程和子行程就成為執行相同程式碼的並行行程
- 因為子行程是父行程的拷貝，每一個行程都有自己的資料拷貝

圖 3.9　使用 fork() 系統呼叫的行程產生



3.3.2 行程的結束

- 有些系統，如果父行程結束，系統不允許子行程繼續存在
  - 在這些系統中，如果行程結束 (不管是正常或不正常的結束)，則行程所產生的所有子行程也都要強迫它結束
  - 這種現象就是所謂的**串接式結束** (cascading termination)，通常都是由作業系統啟動
- 為了說明行程的執行和結束，考慮在 Linux 和 UNIX 系統中，我們可以利用系統呼叫 exit() 來終止一個行程，並提供一個離開的狀態作為參數：

```
/* exit with status 1 */
exit(1);
```


3.3.2 行程的結束

- 事實上，在正常結束的情況下，exit() 可以直接被呼叫或間接被呼叫，因為預設 C 執行時，函式庫 將包括對 exit() 的呼叫

- 父行程可以利用系統呼叫 wait() 等待此子行程結束。父行程向系統呼叫 wait() 傳遞一個參數，而該參數允許父行程獲取子行程的退出狀態
  - 系統呼叫 wait() 可以傳回終止之子行程的識別碼，因此父行程能分辨到底是哪一個子行程終止執行

```
pid_t pid;
int status;

pid = wait(&status);
```


3.3.2 行程的結束

- 當一個行程結束時，它的資源被作業系統重新分配
  - 它在行程表格的進入點必須保留直到父行程呼叫 wait()，因為行程表格保有此行程的離開狀態
  - 已經結束的行程，但是它的父行程還沒有呼叫 wait()，就稱為**殭屍** (zombie) 行程
  - 所有在結束時轉移到此狀態的行程，但通常它們只短暫以殭屍狀態存在
  - 一旦父行程呼叫 wait()，殭屍行程的行程識別碼和它在行程表的進入點就被釋放

3.3.2 行程的結束

- 現在考慮如果父行程沒有呼叫 wait() 而結束，讓它的子行程留下來當孤兒 (orphan) 會發生什麼事呢？
  - 傳統 Linux 系統針對這種情況，藉由設定 init 行程為孤兒行程的新父 行程
    - 在 UNIX 系統層次結構中 init 行程是行程樹的根
  - init 行程週期性地呼叫 wait()，從而允許任何孤兒行程的離開狀態可以被收集，並釋放孤兒行程的識別碼和行程表的進入點

多行程架構—CHROME 瀏覽器

- 許多包含類似 JavaScript、Flash 和 HTML5 的主動內容，提供豐富和動態的網頁瀏覽經驗。很不幸地，這些網頁應用程式可能也包含軟體錯誤，這可能造成遲緩的反應時間，甚至造成網頁瀏覽器的毀損。這對於只顯示一個網站內容的網頁瀏覽器不是一個大問題。但大部份現代的網頁瀏覽器提供分頁瀏覽，允許單一個網頁瀏覽器應用程式同時開啟數個網站，且每一個網站在單獨的分頁。在不同的網站間切換時，使用者只要按一下相對的分頁即可。這個安排描述如下：

多行程架構—CHROME 瀏覽器

- 這種作法有一個問題，如果一個分頁的網頁應用程式毀損，整個行程——包含顯示其它網站的分頁——都會毀損
- Google 的 Chrome 網頁瀏覽器被設計成使用多行程架構來解決這個問題。Chrome 辨認三種不同型態的行程：瀏覽器、渲染器、和插件：
  - **瀏覽器** (browser) 行程負責管理使用者介面、磁碟和網路 I/O。當 Chrome 開始執行時，一個新的瀏覽器行程就會產生。只有一個瀏覽器行程產生
  - **渲染器** (renderer) 行程包含呈現網頁的邏輯。因此，它們包含處理 HTML、Javascript、影像等的邏輯。一般的原則是，在每一個新的分頁開啟的每一個網站都有一個渲染器行程產生，所以同一時間可能有一些渲染器行程在動作

多行程架構—CHROME 瀏覽器

- **插件** (plug-in) 行程對每一個型態的使用插件 (例如，Flash 或 QuickTime) 都會產生。插件行程包含插件的程式碼，並允許插件和相關的渲染器行程和瀏覽器行程溝通

- 多行程作法的優點是網站是以個別隔離的方式執行。如果一個網頁毀損，只有它的渲染器行程受影響；其它的行程沒受到傷害。除此之外，渲染器行程在**沙箱** (sandbox) 中執行，這表示存取磁碟和網路 I/O 是受限的，減少了任何安全漏洞的影響

3.4 行程間通信

- 作業系統之所以要提供環境，以供行程之間合作，有以下幾點理由：
  - 資訊共享：因為數個使用者可能對相同的一項資訊(例如，複製和貼上) 有興趣，因此我們必須提供一個環境允許使用者能同時使用這些資訊
  - 加速運算：如果我們希望某一特定工作執行快一點，就必須將它分成一些子工作，每一個子工作都可以和其它子工作平行地執行
    - 請注意，只有在電腦擁有多個處理核心時，才有可能達到加速的目的
  - 模組化：我們可能希望以模組的方式來建立系統，把系統功能分配到數個行程

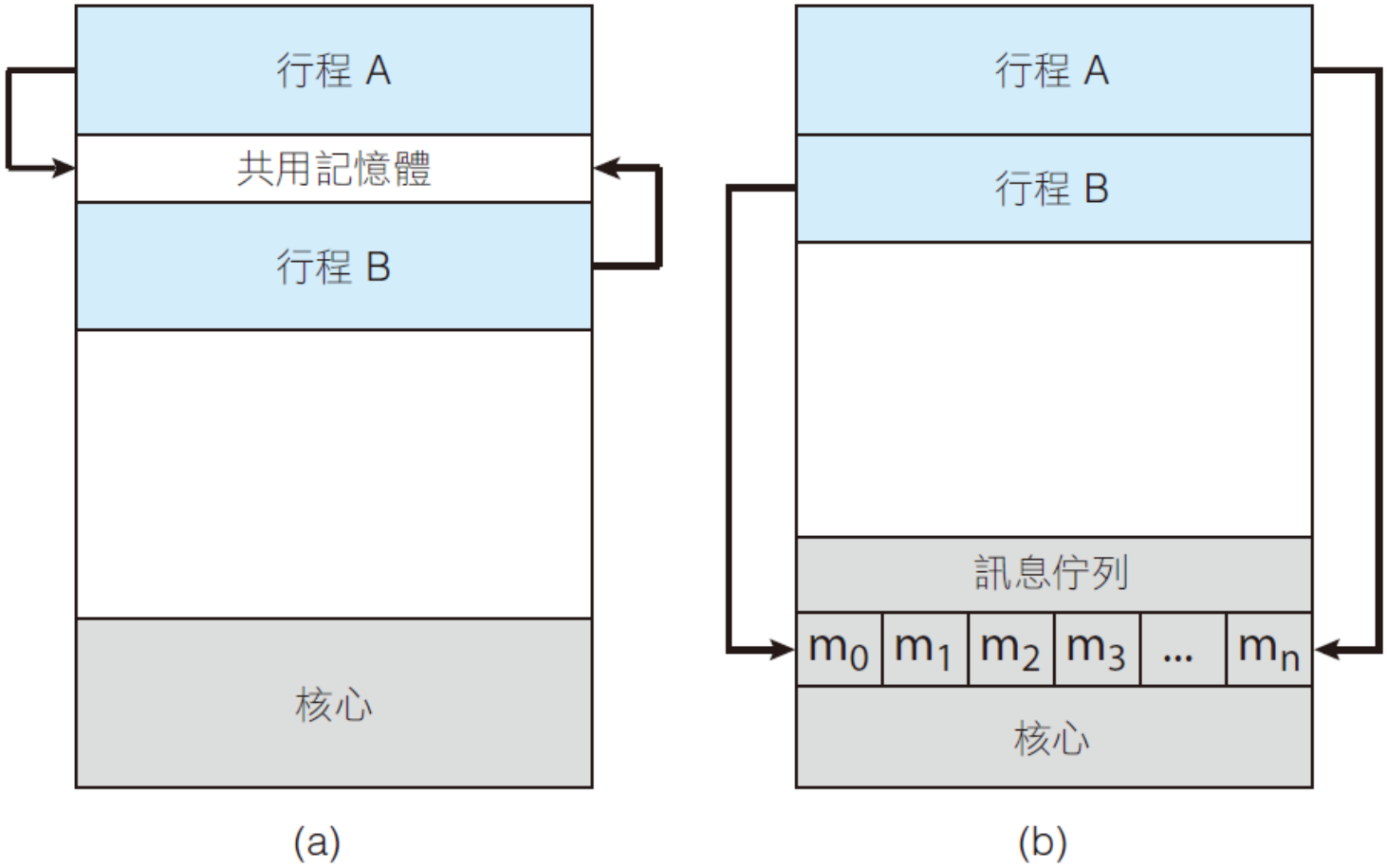
3.4 行程間通信

- 在作業系統中同時執行的行程可分為獨立行程和合作行程兩大類
  - 如果一個行程無法與在系統中正在執行的其它行程共用資料的話，它就是獨立行程 (independent process)
  - 如果一個行程能夠影響其它行程，或是受到其它行程所影響，它就是合作行程 (cooperating process)
  - 明顯地，任何和其它行程共用資料的行程，就是合作行程

3.4 行程間通信

- 合作行程需要有**行程間通信** (interprocess communication, IPC) 的機制讓彼此間交換資料
  - 也就是，彼此間可寄出和接收資料
- 行程間通信有兩個基本模式：
  - 共用記憶體(shared memory)
  - 訊息傳遞 (message passing)
- 在共用記憶體模式中，記憶體的一個區域被合作行程共用，行程藉由讀和寫資料到共用區域來交換資訊
  - 在訊息傳遞模式中，通信發生在訊息交換的合作行程之間。這兩種通信模式的比較如圖 3.11

圖 3.11　通信模式



(a) 共用記憶體；(b) 訊息傳遞

3.5 IPC 共用記憶體系統

- 緩衝有兩種類型：
  - **無限緩衝區** (unbounded buffer) 對於緩衝區的大小沒有限制
    - 消費者可能必須等待新的欄位，但是生產者卻可以不斷地產生新的欄位
  - **有限緩衝區** (bounded buffer) 假設緩衝區的大小固定
    - 在這種情況下
      - 如果緩衝區空了，消費者必須等待
      - 如果緩衝區滿了，生產者必須等待

3.5 IPC 共用記憶體系統

- 行程共用的記憶體區域：

```
#define BUFFER_SIZE10

typedef struct {
        .   .   .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- 這種技巧最多允許 BUFFER_SIZE-1 項資料在緩衝區內

圖 3.12　使用共用記憶體的生產者行程

```
item next_produced;

while (true) {
      /* produce an item in next_produced */

      while (((in + 1) % BUFFER_SIZE) == out)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```


圖 3.13　使用共用記憶體的消費者行程

```
item next_consumed;

while (true) {
      while (in == out)
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next_consumed */
}
```


3.6 訊息傳遞系統中的 IPC

- 3.5 節中，我們已說明合作行程如何在共用記憶體的環境下互相溝通
- 這項技巧需要合作的行程分享一塊共用的緩衝區，而且必須由應用程式設計者撰寫使用此記憶體區的程式碼
- 另外一種達到相同效果的方法，是由作業系統提供行程間通信的設施，以做為合作行程彼此互相溝通的方法

3.6 訊息傳遞系統中的 IPC

- 訊息傳遞提供允許行程互相溝通和彼此同步，而不需要共享相同的位址空間
  - 在分散式的環境 (通信行程放在用網路連接的不同電腦上) 下，訊息傳遞特別有用
  - 例如，全球資訊網所使用的 chat 程式的設計，以便聊天參加者藉由交換訊息彼此溝通
- 訊息傳遞訊息設備提供至少兩種操作：

  send(message)

  和

  receive(message)

  - 一個行程所傳送的訊息大小可能是固定或可變的

3.6 訊息傳遞系統中的 IPC

- 通訊鏈 (communication link) 可用許多的方式製作
  - 鏈結的實體實作
    - 共用記憶體
    - 硬體匯流排
    - 網路
  - 邏輯上製作一個鏈與 send()/receive() 操作的方法
    - 直接或間接聯繫
    - 同步或非同步的聯繫
    - 自動或外在緩衝作用

直接聯繫

- 每一個要傳送或接收訊息的行程必須先確定聯繫接收者或傳送者的名稱
- send() 與 receive() 的基本運算定義如下：
  - send (*P*, message) ——傳送一個訊息 message 至行程 *P*
  - receive (*Q*, message)——自行程 *Q* 接收一個訊息 message

直接聯繫

- 在這個方法中的聯繫鏈具有下列性質：
  - 在每一對要互相聯繫的行程之間的鏈是自動產生，因此這個行程只需知道要互相聯繫行程的身分
  - 一個鏈恰能與兩個行程結合
  - 在每對互相聯繫的行程之間，必存在一個鏈

間接式聯繫

- 需藉著信箱 (mailbox)，也叫作埠 (port) 來傳送與接收訊息
  - 信箱可視為一個抽象的物件，讓一個行程能由這個物件取得訊息，或將訊息置於其中
  - 每一個信箱都有一個識別字，以便區分它們的身分
    - 例如，POSIX 訊息佇列利用一個整數來識別信箱
  - 一個行程可以藉由一些不同的信箱與某些其它的行程互相聯繫，但只有在兩個行程有共用的信箱時，它們才可互相聯繫

間接式聯繫

- 在這個方法中，通訊鏈具有下列的性質：
  - 只有在一對具有共用信箱的行程間才能建立通訊鏈
  - 一個鏈可以和兩個以上的行程相結合
  - 在每對互相通信的行程之間，可能存在數個鏈，而且每個鏈對應一個信箱

間接式聯繫

- send() 與 receive() 的基本運算之定義如下：
  - send(A, message)——將一個訊息 message 傳送至信箱 A
  - receive(A, message)——自信箱 A 接收一個訊息 message
- 一個信箱是作業系統擁有時就是自己獨立存在，與其它特定行程無關，並不附屬於任何行程，作業系統必須提供一個方法，允許一個行程執行其事項：
  - 產生一個新的信箱
  - 經由信箱傳送並接收訊息
  - 刪除一個信箱

間接式聯繫

- 共用信箱
  - P_1、P_2 與 P_3 共用信箱 A
  - 行程 P_1 傳送訊息至 A，P_2 與 P_3 執行 receive()
  - 到底哪一個行程可以接收到自 P_1 傳送來的訊息？
- 解答取決於我們所選擇的方法：
  - 允許一個鏈最多只能與兩行程相結合
  - 允許一個行程每次最多只能執行一個 receive() 操作
  - 允許系統能任意選取接收訊息的。傳送者確認接收者是誰
    - 運算法則：輪詢，行程輪流接收訊息

3.6.2 同步化

- 行程間通信藉由呼叫 send() 和 receive() 基本操作來完成
  - **等待** (blocking) 也稱為**同步** (synchronous)
    - 等待傳送 ：傳送行程等待著，直到接收行程或信箱接收訊息
    - 等待接收：接收者等待，直到有訊息出現
  - **非等待** (nonblocking) 也稱為**非同步** (asynchronous)
    - 非等待傳送：傳送行程送出訊息及重新操作
    - 非等待接收：接收者取回有效訊息或無效資料
- 可能有不同的組合
  - 當傳送 send() 與接收 receive() 兩者都在等待時，則傳送者與接收者之間就有**約會** (rendezvous)

生產者–消費者行程

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

圖3.14 使用訊息傳遞的生產者行程

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

圖 3.15　使用訊息傳遞的消費者行程

3.6.3 緩衝器

- 不論是直接或間接聯繫，經由通信行程交換的訊息是放在一個暫時的佇列
- 三種製作這種佇列的方式：
  - 零容量 (zero capacity)：鏈中將不含有任何等候的訊息，傳送者必須等候接收者接收到資料
  - 有限的容量 (bounded capacity)：鏈是有限長度，如果鏈已經填滿，傳送者必須等候佇列空間
  - 無限制的容量 (unbounded capacity)：具有無限長度的潛力，任何個數的訊息能在佇列中等候，傳送者從不阻塞

3.7 IPC 系統的範例

- POSIX 共用記憶體使用記憶體對映檔案的方法，此方法將共用記憶體與檔案做關聯
- 行程必須先使用 shm_open() 系統呼叫產生共用記憶體物件：

```
fd = shm_open(name, O_CREAT | O RDWR, 0666);
```


3.7 IPC 系統的範例

- 第一個參數指定共用記憶體物件的名字，希望存取共用記憶體的行程必須使用此名字來參考這個物件
  - 後續的參數指定如果要被產生的共用記憶體物件不存在 (O_CREAT)，而且此物件是為了讀和寫 (O_RDWR) 而開啟，則此物件將被產生
  - 最後一個參數建立共用記憶體物件的目錄允許權
  - 呼叫 shm_open() 成功會傳回一個共用記憶體物件的檔案描述器 (file descriptor) 整數值

3.7 IPC 系統的範例

- 一旦此物件被建立，函數 ftruncate() 被用來設定物件以位元組為單位的大小
  - 例如，呼叫

```
ftruncate(smh_fd, 4096);
```

  - 設定物件的大小為 4,096 位元組
- 函數 mmap() 建立一個包含共用記憶體物件的記憶體對映檔案
  - 它也傳回一個指到記憶體對映檔案的指標，被用來存取共用記憶體物件

3.7 IPC 系統的範例

- 生產者建立共用記憶體物件，並寫入共用記憶體，而消費者從共用記憶體讀取
  - 圖 3.16 的生產者產生一個名稱為 OS 的共用記憶體物件，寫入字串 “Hello World!" 到共用記憶體
    - 程式記憶體對映一個指定大小的共用記憶體物件，並允許寫入到此物件
    - 旗標 MAP_SHARED 設定對共用記憶體的改變將可讓所有共用此物件的行程看到
    - 藉由呼叫函數 sprintf() 來寫入共用記憶體物件，並寫入格式化字串到指標 ptr
    - 每次寫入，必須將指標曾寫入的位元組數目

圖 3.16 說明 POSIX 共用記憶體 API 的生產者行程

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* create the shared memory object */
    fd = shm_open(name,O_CREAT | O_RDWR,0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```


圖 3.17 說明 POSIX 共用記憶體 API 的消費者行程

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```


3.7.2 Mach 訊息傳遞

- Mach 中大部份的通信都是由**訊息** (message) 完成
  - 訊息都是由信箱來傳送及接收
  - 埠的數量是有限的，並且是單向的
  - 對於雙向通信，將訊息發送到一個埠，並將回應傳送到個別的應答埠
  - 每個埠可能具有多個發送者，但只會只有一個接收者
  - Mach 使用埠來代表資源，其中作為任務、執行緒、記憶體和處理器，而訊息傳遞則提供物件導向的方法與這些系統資源和服務互動
  - 訊息傳遞可能發生在同一主機上的任何兩個埠之間或在分散式系統上的單一主機上

3.7.2 Mach 訊息傳遞

- 與每個埠相關的是一個**埠的權限** (port rights)，這些埠的權限標示了任務交互所必需的功能
  - 例如，對於從埠接收訊息的任務，它必須具有該埠的 MACH_PORT_RIGHT_RECEIVE 能力
  - 建立埠的任務是該埠的擁有者，而擁有者的任務是唯一允許從該埠接收訊息
  - 埠的擁有者還具有操作該埠的功能，這通常是在建立回覆埠的時候來完成
    - 例如，假設任務 *T*1 擁有埠 *P*1，並且向埠 *P*2 (為任務 *T*2 所擁有) 發送一則訊息

3.7.2 Mach 訊息傳遞

- 任務 *T1* 希望收到 *T2* 的回覆，則必須授予 *T2* 的埠 *P1* 之正確 MACH_PORT_RIGHT_SEND 訊息
- 埠權限的所有權跟任務的級別有關，這意味著所有行程屬於同一任務的共享相同的埠權限
- 屬於同一任務的兩個執行緒可以透過與每個執行緒相關聯的每個執行緒埠交換訊息來進行通信

3.7.2 Mach 訊息傳遞

- 當建立任務時，還將建立兩個特殊埠—*Task Self* 埠和 *Notify* 埠
  - 核心具有 Task Self 埠的接收權限，該埠允許任務將訊息發送到核心
  - 核心可以將事件發生的通知發送到任務的 Notify 埠
- mach_port_allocate() 函數建立一個新的埠，並為其訊息佇列分配空間及標識埠權限
  - 每個埠的權限代表該埠的名稱，並且只能藉由該埠權限進行存取
  - 埠名稱是簡單的整數值，其行為與 UNIX 檔案描述符非常相似

3.7.2 Mach 訊息傳遞

- 送出 (send) 和接收 (receive) 的操作本身頗有彈性
  - 例如，當一個訊息送到一個埠時，它的佇列可能是滿的
  - 如果這個佇列不是滿的，訊息便會複製到佇列之中，而且繼續傳送任務
  - 如果埠的佇列是滿的，這個傳送者有四種選擇：
    1. 不確定的等待直到佇列中有空間
    2. 最多等待 n 毫秒
    3. 根本不等待，馬上返回
    4. 暫時地保留訊息，可以送一個訊息給作業系統以便保持，甚至當被傳送訊息的佇列是滿的

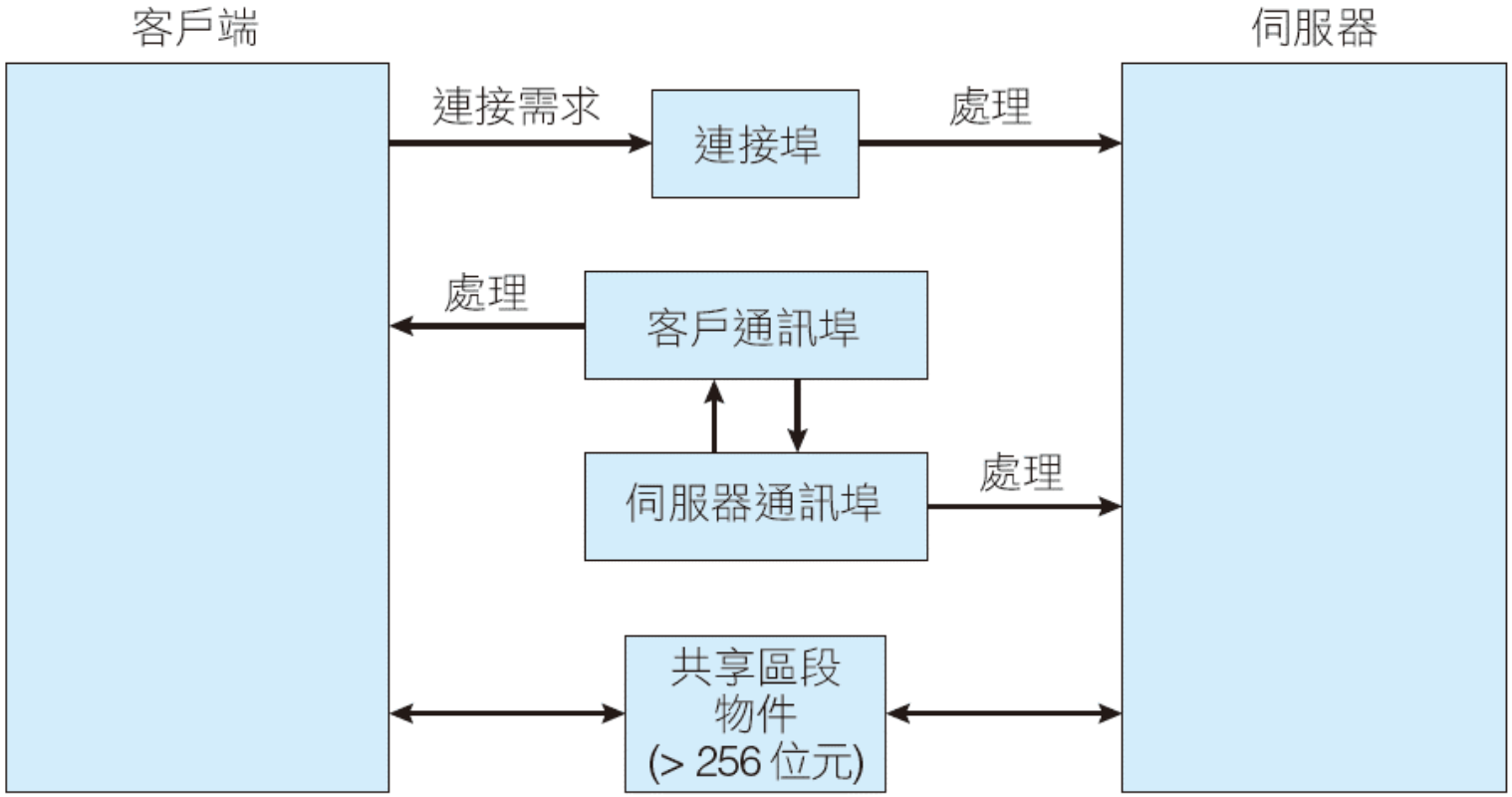
3.7.3 Windows

- 在 Windows 中的訊息傳遞設備稱為**進階區域程序呼叫** (advanced local procedure call, ALPC)
  - 被用來讓同一台機器中的兩個行程之間做通信
  - 相似於標準的遠端程序呼叫 (remote procedure call, RPC) 功能
  - 廣泛地使用，但特別是在 Windows 有最佳的結果
  - Windows 如同 Mach 一樣，它使用一個埠物件來建立和維持一條在兩個行程之間的連接
  - Windows 使用兩種形式的埠：
    - **連接埠** (connection ports)
    - **通訊埠** (communication ports)

3.7.3 Windows

- 伺服器行程發布連接埠物件讓所有行程看得見
  - 當客戶端希望從子系統獲得服務，它開啟一個處理器 (handle) 給此伺服器的連接埠物件，並傳送連接要求給該連接埠
  - 伺服器產生一個通道，並傳回處理器給客戶端。此通道由一組私有的通訊埠組成：
    - 一個給客戶端—伺服器訊息
    - 一個給伺服器—客戶端訊息
  - 通信通道支援回呼 (callback) 機制，此機制允許客戶端和伺服器期待一個回覆時接受要求

圖 3.19 Windows 中的進階區域程序呼叫



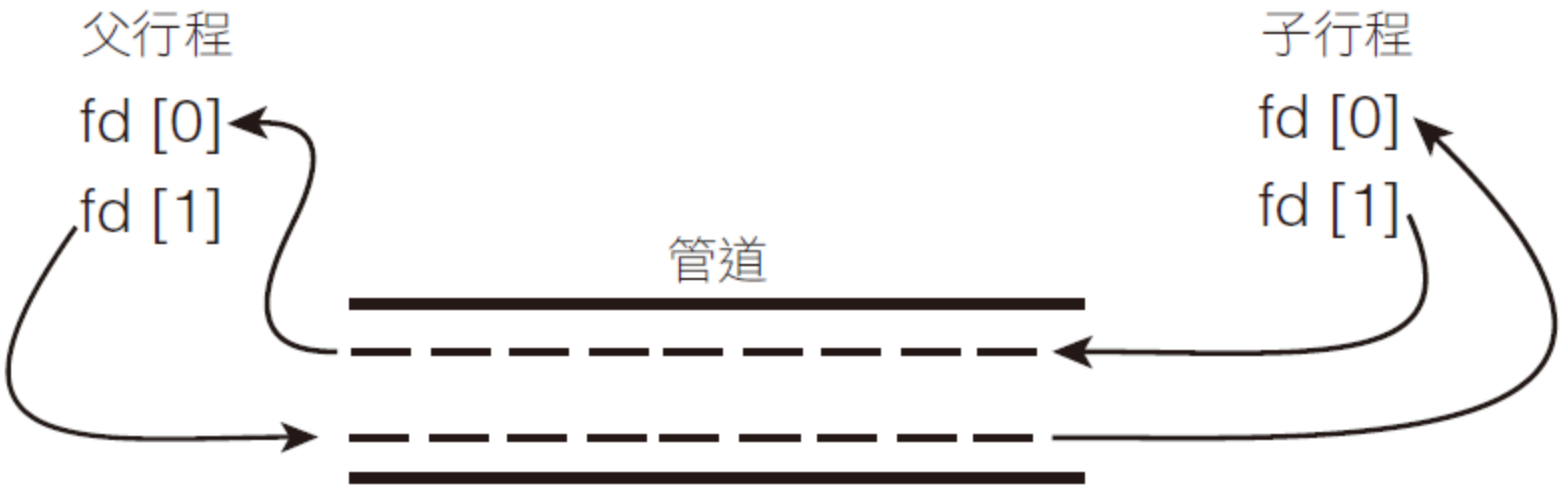
3.7.4 管道

- **管道** (pipe) 如同一個允許兩個行程通信的導管
  - 早期 UNIX 系統中，管道是最先IPC 的機制之一
  - 管道是一種提供行程與其它行程間通信的更簡單方法，雖然它們也有一些限制
  - 實作管道時，有四個議題必須考慮：
    1. 管道是否允許單向通信或雙向通信？
    2. 若雙向通信是被允許的，那麼是半雙工或是全雙工？
    3. 通信的兩個行程間是否必須存在如父－子關係？
    4. 管道通信是否可透過網路，或是通信的行程必須在同一台機器上？

3.7.4.1 普通的管道

- 普通的管道允許兩個行程在標準生產者－消費者方式下進行通信：
  - 生產者從管道的寫入端 (write end) 寫入，消費者從另一**讀取端** (read end) 讀取
  - 普通的管道是單向的，只有允許單向通信
  - 若是有雙向通信的需求，兩個管道就必須使用，每個管道傳送不同方向的資料
  - 在這兩個的程式範例中，一個行程編寫訊息 Greetings 至管道中，而另一個行程從管道讀取這個訊息

圖 3.20　普通的管道的檔案描述符



3.7.4.2 命名管道

- 命名管道提供更有力量的通信工具
  - 通信可以雙向
  - 不需要父－子關係
- 一旦命名管道被建立，幾個行程可以使用它以進行通信
  - 在典型的方案中，命名管道有許多編寫者
  - 命名管道在行程完成通信後仍持續存在
  - UNIX 和 Windows 系統支援命名管道，雖然實作的細節差別很大

3.7.4.2 命名管道

- 命名管道在 UNIX 系統中被稱為 FIFO
  - 一旦被產生後，它們就好像檔案系統中的典型檔案
  - FIFO 由系統呼叫 mkfifo() 產生並且由普通的 open()、read()、write() 和 close() 系統呼叫操作
  - 持續存在直到明確地從檔案系統中刪除
  - FIFO 允許雙向通信，但只有允許半雙工傳送
  - 如果資料必須在兩個方向中傳送，則通常會使用兩個 FIFO
  - 通信行程必須常駐於同一台機器
    - 機器間通信被要求時，則必須使用插座

3.8 客戶端–伺服器的通信

- 在客戶端—伺服器系統通信的其它兩個策略：
  - 插座
  - 遠程程序呼叫 (RPC)

3.8.1 插　座

- **插座** (socket) 定義成通信的終端
  - 一組行程使用一對插座
    - 每個行程一個
    - 在網路上通信
  - 一個插座是由一個 IP 位址和一個埠號碼所組成
  - 通常插座使用了客戶端—伺服器的架構
  - 伺服器藉由傾聽某一特定埠來等待進入的客戶要求
  - 一旦要求被接受之後，伺服器就接受從客戶端插座的連接來完成連接

3.8.1 插 座

- 製作特定服務的伺服器傾聽眾所皆知的埠
  - SSH 伺服器傾聽埠 22
  - FTP 伺服器傾聽埠21
  - 網頁伺服器 (HTTP) 傾聽埠 80
- 1024 以下的埠都是眾所周知的；我們只能用它們來製作標準的服務

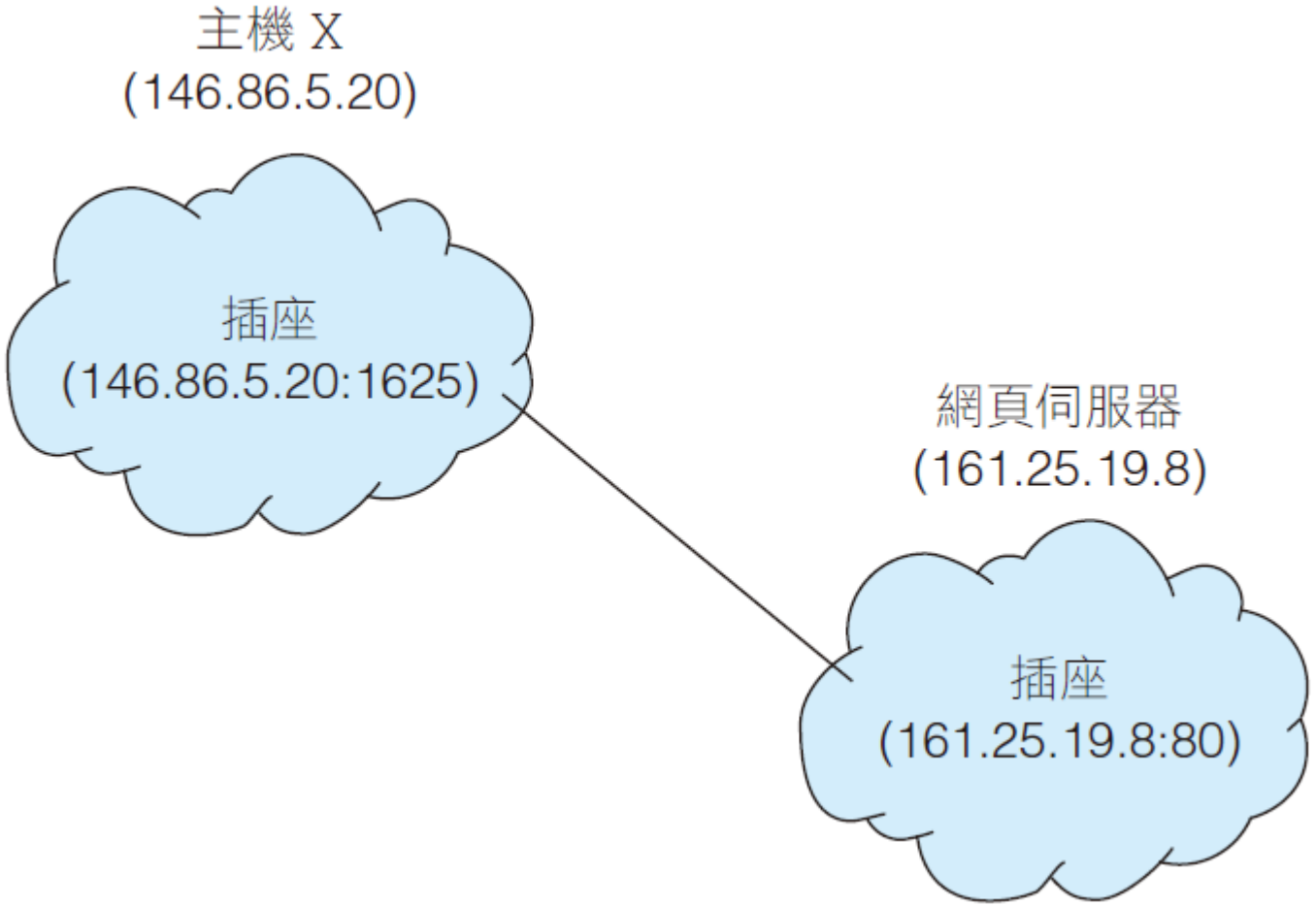
3.8.1 插 座

- 當一個客戶端的行程啟動一項連接要求時，它會被主電腦設定一個埠，大於1024 的任意數字
  - 譬如，如果主機 X 的客戶端 (IP 是 146.86.5.20) 希望和位於 161.25.19.8的網頁伺服器建立連線，主機 X 被設定成使用埠 1625
  - 此連接將由一組插座組成：
    - 主機 X 的 (146.86.5.20:1625) 和網頁伺服器的 (161.25.19.8:80)
    - 此狀況描繪在圖 3.26
    - 封包根據目的地埠號碼在主機間旅遊，並傳送到適當的行程

3.8.1 插 座

- IP 位址 127.0.0.1 是一個特殊位址，也就是**內部迴圈網路** (loopback)
  - 當一台電腦使用到 127.0.0.1 的 IP 位址時，它就連到自己
  - 這種功能允許客戶端和伺服器在同一台電腦上使用 TCP/IP 協定通信
  - IP 位址 127.0.0.1 可以換成其它執行日期時間伺服器的 IP 位址
  - 真實主機名也可以使用，如 www.westminstercollege.edu

圖 3.26　使用插座的通信



3.8.1 插 座

- Java 提供三種不同型態的插座
  - **連接傾向插座** [connection-oriented (TCP) socket]
    - 要用 Socket 類別製作
  - **無連接傾向插座** [connectionless (UDP) sockets]
    - 必須使用DatagramSocket 類別
  - 廣播插座允許資料被送到許多接收者
    - MulticastSocket 類別，它是 DatagramSocket 類別的子類別

圖 3.27 日期伺服器

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```


3.8.2 遠程程序呼叫

- RPC 被設計成一種使用在以網路連接之系統間的程序呼叫機制
  - 它在許多方面和 3.4 節所討論過的 IPC 功能很相似，而且它通常被建立在一個系統的上層
  - 因為我們所處理的是各個行程在個別系統上執行的環境，我們必須使用一個以訊息為基礎的通信技巧以提供遠程服務

3.8.2 遠程程序呼叫

- 相對於 IPC 訊息，RPC 通信所交換的訊息有良好的結構，因此不再只是封包形式的資料
  - 這些訊息被送往一個 RPC 守護程序
    - 此 RPC 守護程序正在傾聽遠方系統的一個埠，訊息中包含被執行函數的識別碼和傳送給該函數的參數
  - 這個函數會根據要求被執行，而任何輸出會以另一個獨立訊息送回給要求者

3.8.2 遠程程序呼叫

- 一個**埠**在這裡是一個包含在訊息封包開頭的簡單數字
  - 雖然系統通常有一個網路位址，但每一個網路位址可以有許多埠在其中，以區別它所支援的許多網路服務
  - 如果一個遠端行程需要一項服務時，它會將它的訊息送到正確的埠
    - 例如，如果系統希望允許其它系統能夠列出目前其上的使用者時，它將有一個守護程序支援連接到某一個埠 (例如埠 3027) 的 RPC
    - 任何一個送出 RPC 訊息到伺服器上埠 3027 的遠端系統可以獲得所需要的資訊：
      - ✓ 資料會以回發資訊的格式接收

3.8.2 遠程程序呼叫

- RPC 的語法允許客戶端對遠端主機呼喚一個程序時就如同它呼叫本地的程序一樣
  - RPC 系統隱藏允許通信發生時所需要的細節
  - RPC 系統藉由在客戶端提供的**存根** (stub)完成這項工作
  - 對於每一個獨立的遠程程序都存在一個個別的存根
  - 當客戶呼喚一個遠程程序時，RPC 的系統呼叫適當的存根，並傳遞給它要提供給遠程程序的參數
  - 此存根找到伺服器上的埠，並且**重排** (marshal) 參數
  - 參數的重排牽涉到將參數封裝成可以在網路上傳遞的格式

3.8.2 遠程程序呼叫

- 然後存根使用訊息傳遞的方法傳遞信號給伺服器
- 伺服器端另一個相類似的存根接收此訊息，而且在伺服器端呼叫程序
- 如果有必要時，傳回值也使用相同的技巧傳回到客戶端
- 在 Windows 系統，存根程式碼是由使用**微軟介面定義語言**(Microsoft Interface Definition Language, MIDL) 寫的規格編譯而成，MIDL 被用來定義客戶端和伺服器間的介面

3.8.2 遠程程序呼叫

- 參數編碼解決了有關客戶端和伺服器端上資料表示差異的問題
  - 考慮 32 位整數的位元表示順序
    - **大在前排列法** (big-endian) 是指資料放進記憶體中的時候，最高位的位元組會放在最低的記憶體位址上，
    - **小在前排列法** (little-endian) 則是剛好相反，它會把最高位的位元組放在最高的記憶體位址上
  - 兩種位元順序都有被採用並沒有任一個是“較佳的”
  - 為了解決這種差異，許多 RPC 系統都定義了與機器無關的資料表示形式

3.8.2 遠程程序呼叫

- 一種這樣的表示稱為**外部資料表示** (external data representation, XDR)
- 在客戶端，參數重排包括將與機器相關的資料轉換為XDR，然後再將其發送到伺服器端
- 在伺服器端對 XDR 資料進行封送處理並將其轉換為伺服器的機器相關表示

3.8.2 遠程程序呼叫

- 有兩種常見的作法
  - 第一種是連接的資訊可以用固定埠位址的格式預先決定
    - 在編譯時，RPC 系統呼叫有一個和它相關的固定埠號碼
    - 一旦程式被編譯好之後，伺服器不能改變所要求之服務的埠號碼
  - 第二種方法是結合可以藉由約會的機能動態地完成
    - 通常作業系統會在一個固定的 RPC 埠提供一個約會守護程序
    - 也稱為**媒人** (matchmaker)

圖 3.29 遠端程序呼叫 (RPC) 的執行