第四章 處理元

4.1 處理元的觀念(Process Concept)

- 正在執行的程式（Program）稱為處理元（Process）
- A program in execution

- 通常電腦使用者所說的程式，當它執行後，作業系統將它看成是處理元。

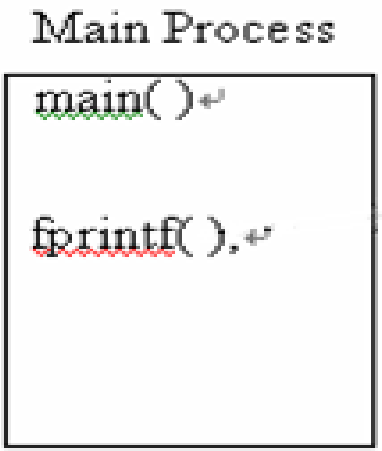


圖 3.1　計算處理元及輸出 / 輸入處理元

處理元(Process)(1)

- 程式是一個被動的實體（Passive Entity），它必須由人或系統要求執行，
- 而處理元是一個活動的實體（Active Entity），一旦它被建立之後，便能獨立自主於作業系統內活動。
- A process will need certain resources: CPU time, memory, files, I/O devices to accomplish its task.

處理元(Process)(2)

- 工作是為完成一個任務所須的一連串動作，一個工作是由多個工作步驟（Job Step）組成，此處的工作步驟可以視為程式，而一個程式執行後，會建立多個處理元。

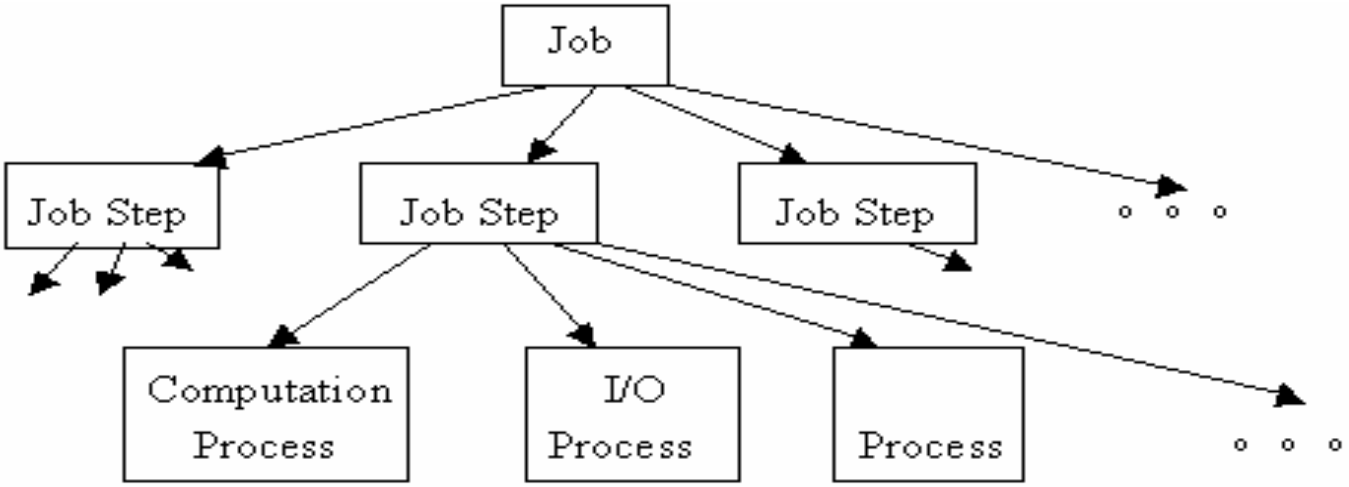


圖 3.2　工作、程式、及處理元之關係

處理元(Process)(3)

- 處理元又可稱為工作元（Task），由於在一個分時多重程式（Time Sharing Multiprogramming）系統中，每個使用者可同時執行出許多個處理元，這些處理元依中央處理器排程輪流使用中央處理器。

- 分時多重程式系統又可稱為多重工作元（Multitasking）系統。

處理元(Process)(4)

- 在只有一個中央處理器的電腦，處理元的執行是虛擬平行（Pseudo Parallelism），也就是許多處理元輪流使用中央處理器。

- 若電腦內有多顆中央處理器，則可以同時有多個處理元使用中央處理器，它們是真正平行（True Parallelism）。

處理元包含的元件

- 被執行的機器碼。
- 執行時所需要使用的資料。
- 執行時所需要使用的資源，例如記憶體、檔案、輸出 / 輸入設備、中央處理器...等。
- 堆疊。
- 處理元的狀態（Status）。

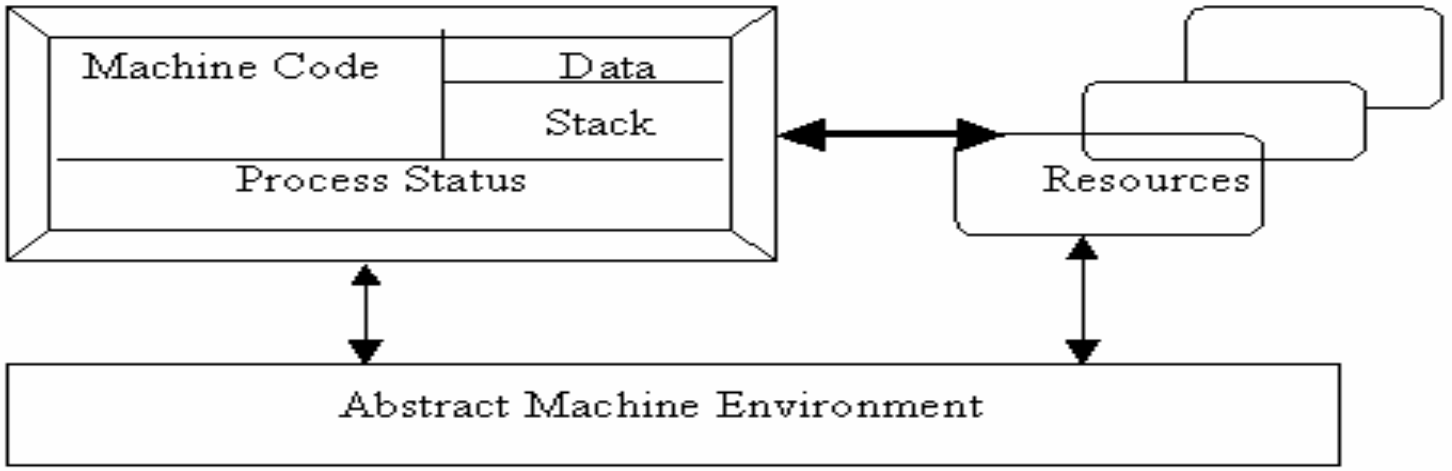


圖 3.3　處理元包含的元件

處理元的狀態(1)

- Only one process can be running on any processor at any instant.

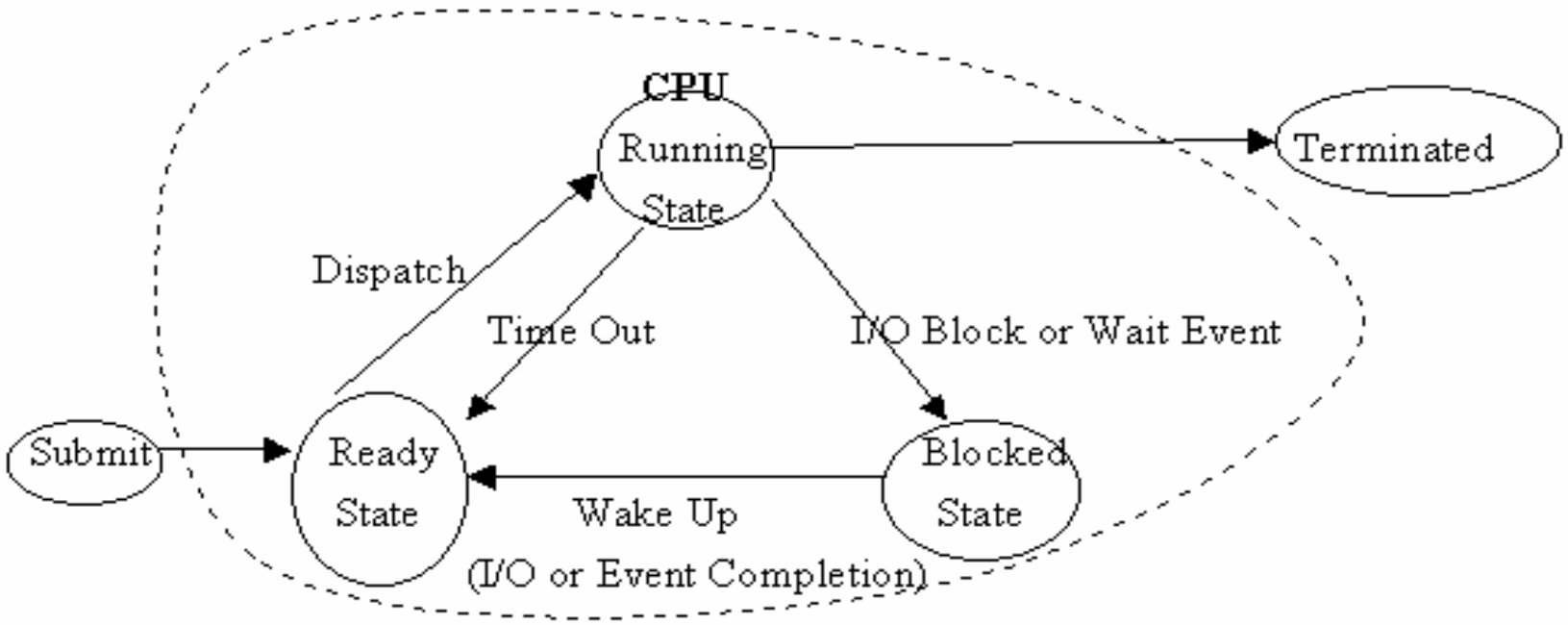


圖 3.4　處理元的狀態

處理元的狀態(2)

- 備妥狀態（Ready State）
  - 由於有許多處理元等待執行，所以系統內會有一個備妥佇列，讓等待的處理元依序排列，我們說在備妥佇列排隊的處理元，是在備妥狀態中。

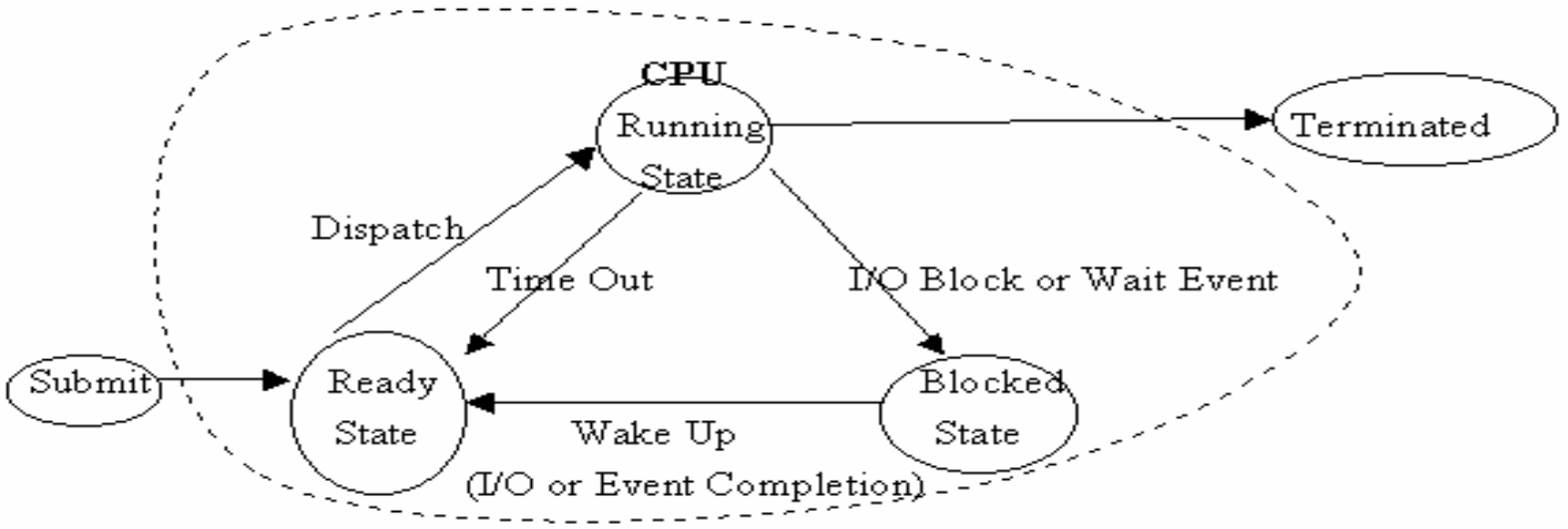


圖 3.4　處理元的狀態

處理元的狀態(3)

- 執行狀態（Running State）
  - 假設中央處理器只有一個，則在每一瞬間僅會有一個處理元使用中央處理器執行工作，正在執行的處理元便是在執行狀態。

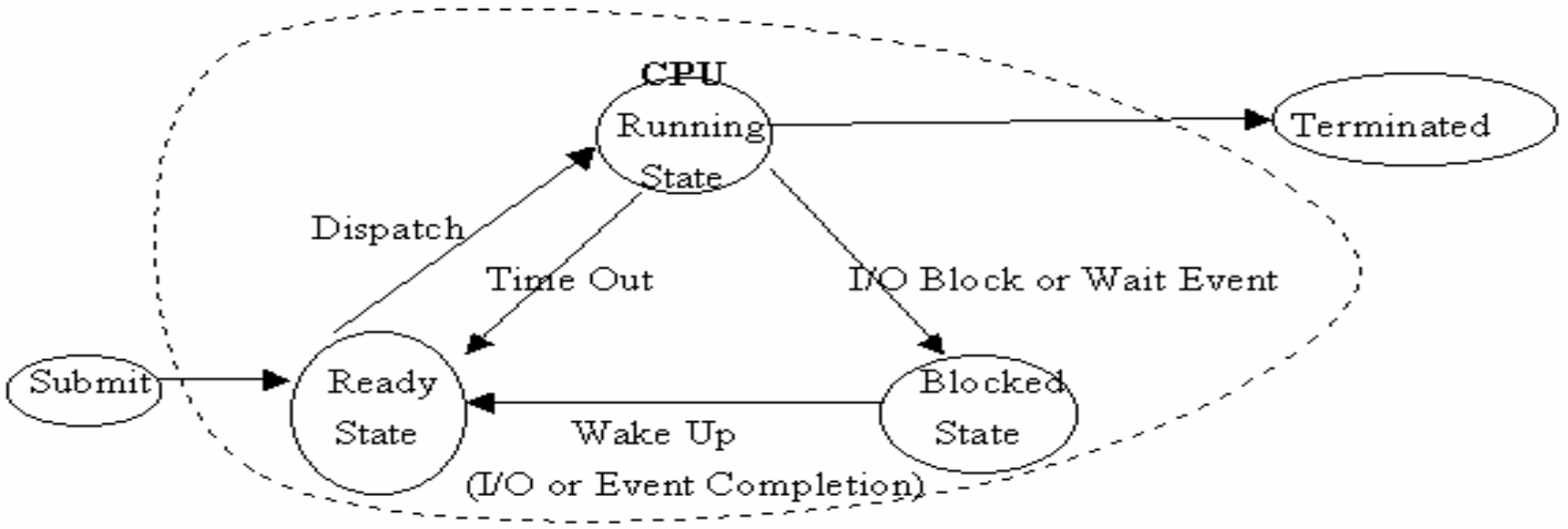


圖 3.4　處理元的狀態

處理元的狀態(4)

- 懸置狀態（Blocked State）
  - 懸置狀態又稱為等待狀態（Waiting State）。當在執行狀態的處理元欲進行輸出 / 輸入時，它不必佔有中央處理器，而將中央處理器讓出給別的處理元，而自己就進入懸置狀況，等待輸出 / 輸入作完。同樣必須等待事件（Event）發生之處理元，亦必需須進入懸置狀態等待。

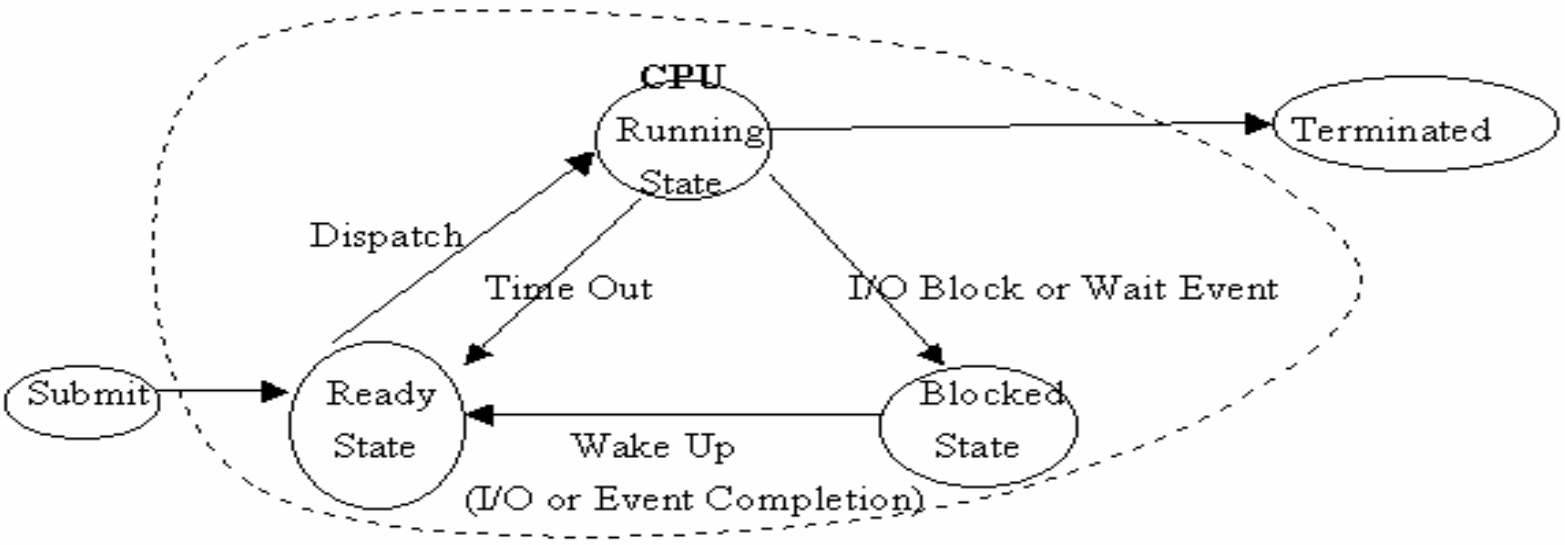


圖 3.4　處理元的狀態

處理元的狀態(5)

- Dispatch (Dispatcher)
  - Assign CPU to the process.

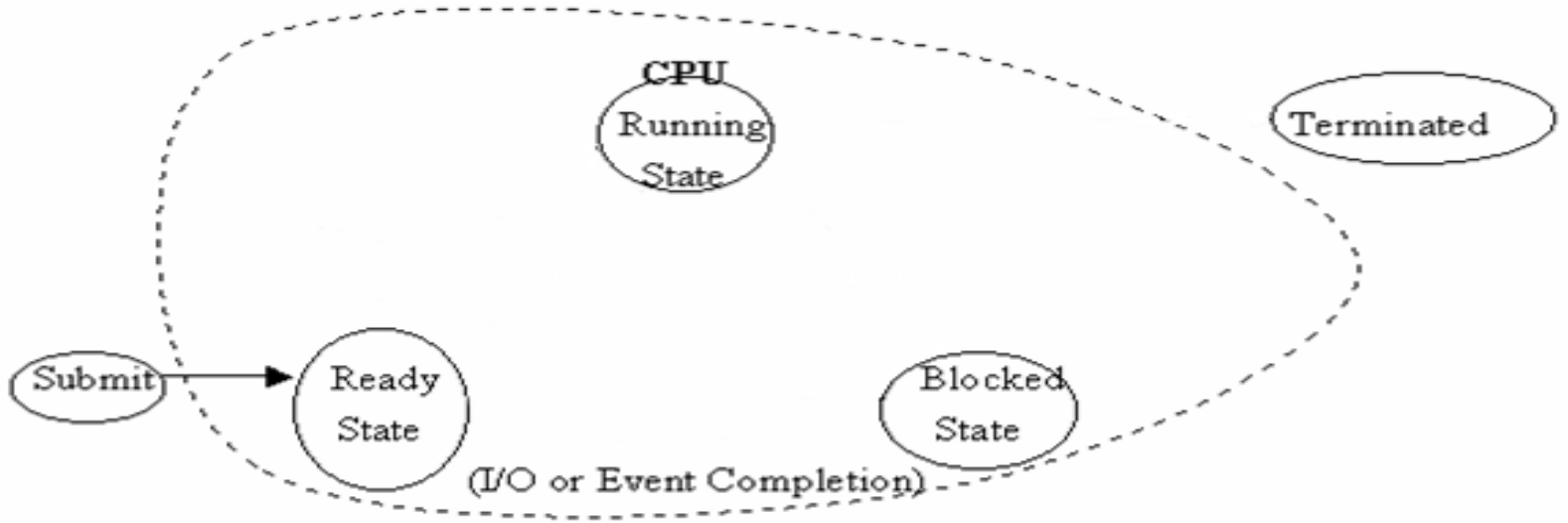


圖 3.4　處理元的狀態

環境切換（Context Switch）(1)

- 當時間片段用罄之後，中央處理器排程器會執行時間逾時，將處理元變換為備妥狀態，並至備妥狀態中分派另一個處理元進入執行狀態，這個動作我們稱為環境切換（Context Switch）。
- Switching the CPU to another process.

- 環境切換是一種中斷，它的處理方式與發生中斷後處理方式相同 。

環境切換（Context Switch）(2)

- 環境切換之處理方式如下：
  - 作業系統取得控制權，並將處理元狀態儲存至處理元控制區塊內。
  - 執行中央處理器排程工作，並至備妥佇列內選取一個處理元
  - 將被選取之處理元之處理元控制區塊載入系統內。
  - 執行此被選取之處理元。
- When an interrupt occurs, the OS save the status of interrupted process, routine control to appropriate interrupt handler, then loading the saved state for the new process to execute.

處理元控制區塊(Process Control Block)(1)

- 處理元隨時會變換狀態，因此每個處理元一旦建立之後，它就必須有處理元控制區塊（Process Control Block），簡稱PCB。

- 處理元控制區塊有時又稱為處理元描述者（Process Descriptor）。

處理元控制區塊(Process Control Block)(2)

- 處理元控制區塊會記錄這個處理元的內容，包含
  - 處理元編號（Process Identifier），簡稱PID
  - 處理元狀態（Process State）
  - 程式計數器
  - 中央處理器之暫存器內容
  - 處理元優先等級
  - 處理元地址空間
  - 輸出 / 輸入狀態、使用資源情形
  - 帳號資訊...等。

- 處理元編號是唯一的，它不與其他處理元重復，在作業系統內均以處理元編號來辨識處理元。

處理元之排程(1)

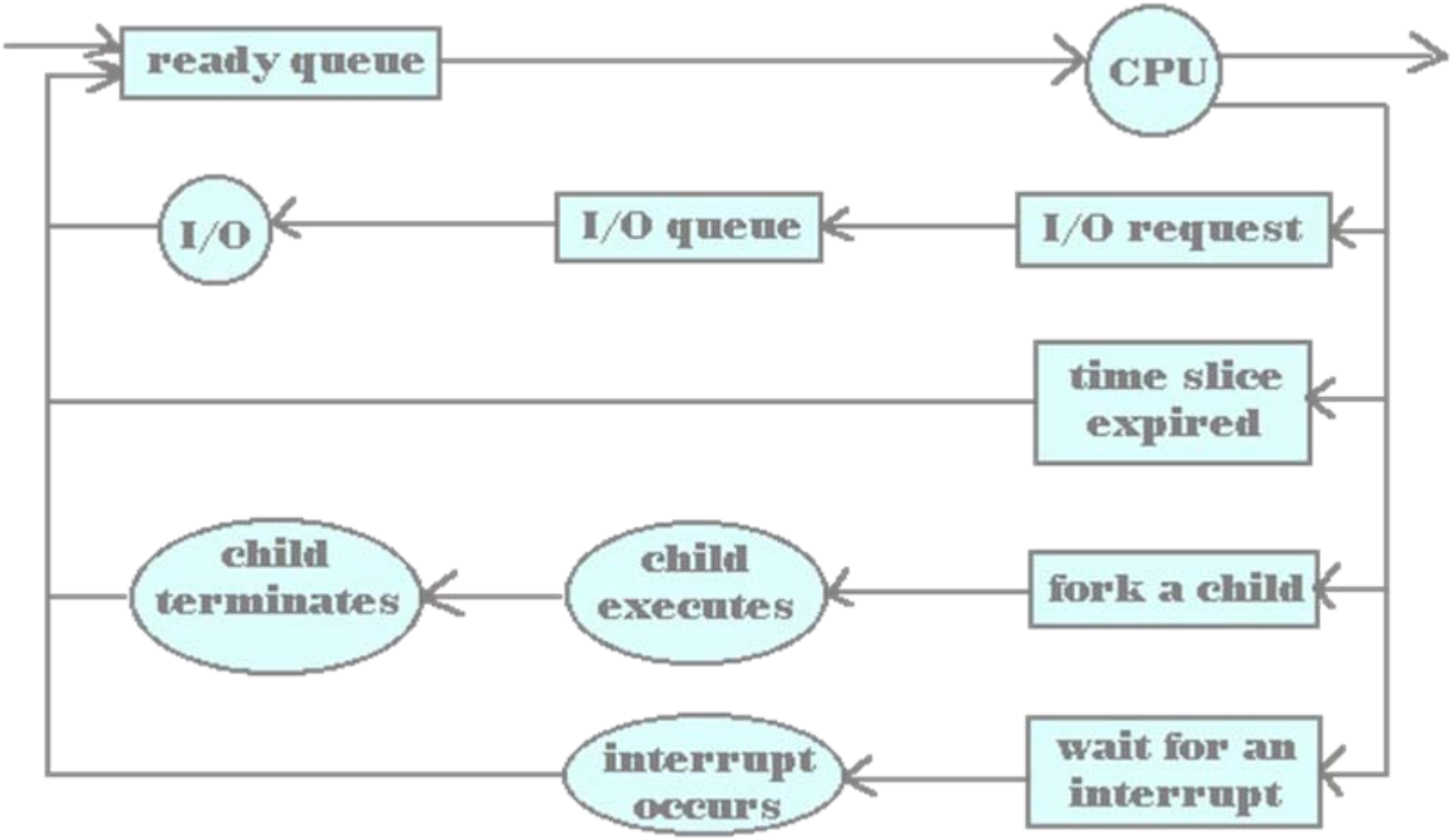
4.2 處理元之排程

- 在作業系統內，處理元幾乎都是隨時在排隊等待，它們要等待系統給予資源、等待事件發生、等待中央處理器、等待輸出 / 輸入...等。

- 中央處理器排程器（CPU Scheduler）是一種作業系統軟體，它會到備妥佇列內選取一個處理元，並將此處理元分派（Dispatch）給中央處理器。

- Selects from among the processes that are ready to execute, and dispatch one of them.
-

處理元之排程(2)


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child terminates
child executes
fork a child
interrupt occurs
wait for an interrupt

處理元之排程(3)

- 中央處理器排程器有時又被稱為短期排程器（Short Term Scheduler），它僅對已經擁有完整資源之處理元進行佔有中央處理器之排程工作 。

- 工作排程器（Job Scheduler），又稱為長程排程器（Long Term Scheduler）。

- 中程排程器（Medium Term Scheduler）。

長程排程器

- 長程排程器之目的是將處理元由磁碟內搬至主記憶體內，以便進入備妥佇列，等待被執行。
- Selects processes from disk pool and loads them into memory for execution.

- 長程排程器可以用來控制多重程式之程度（Degree of Multiprogramming），所謂多重程式之程度，是指在主記憶體內處理元之個數。
- Long term scheduler control the degree of multiprogramming.

中程排程器

- 中程排程器是用來將某些處理元由主記憶體內移出至磁碟內，它減少多重程式之程度。
- Medium term scheduler reduce the degree of multiprogramming.

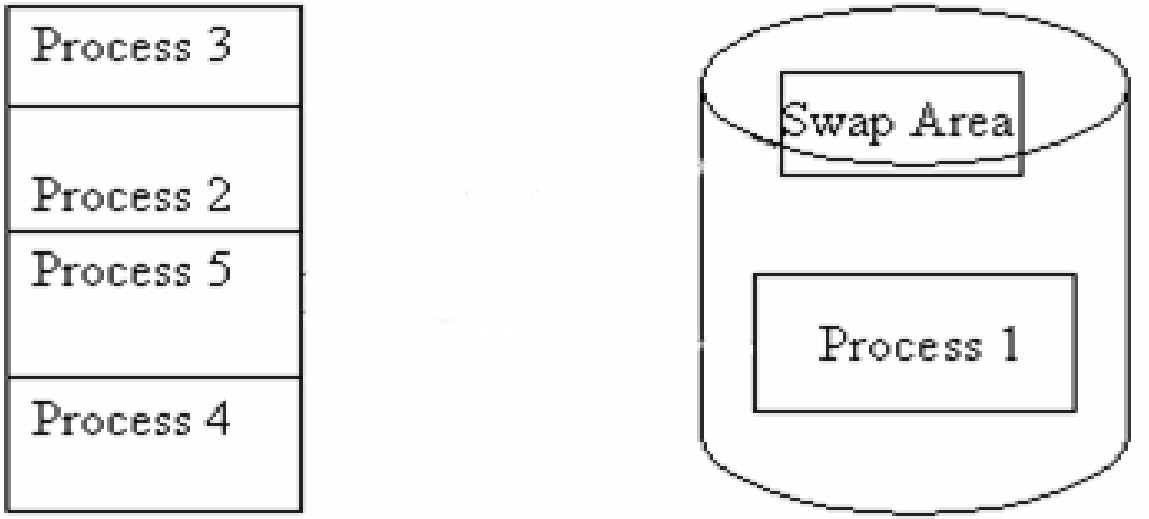


圖 3.5　中程排程器之置換出/置換入

I/O Bound Process and CPU Bound Process(1)

- 輸出 / 輸入限制處理元（I/O Bound Process）
  - 處理元在執行的過程當中，執行輸出 / 輸入的時間比計算時間多很多的處理元。
  - Spend more of its time doing I/O than computations.

- 中央處理器限制處理元（CPU Bound Process）
  - 執行計算的時間比輸出 / 輸入的時間多很多的處理元。
  - Use more of its time doing computations than I/O.

I/O Bound Process and CPU Bound Process(2)

- All processes are I/O bound
  - CPU scheduler will have little to do.

- All processes are CPU bound
  - Device will go unused.
  - 使用者會感覺電腦系統執行的異常緩慢，這是因為每個中央處理器限制處理元進入執行狀態後，幾乎都佔滿時間片段才離開的原因。

處理元之運作（Process Operating）(1)

4.3　處理元之運作（Process Operating）

- 使用者在外殼（Shell）中下一個ls命令，外殼便衍生（Fork）另一個外殼，被生出之外殼以系統呼叫叫出ls這個處理元來執行，此時被生出之外殼結束執行，而原來之外殼與ls這個處理元同時在工作，亦即原來之外殼繼續等待命令　。

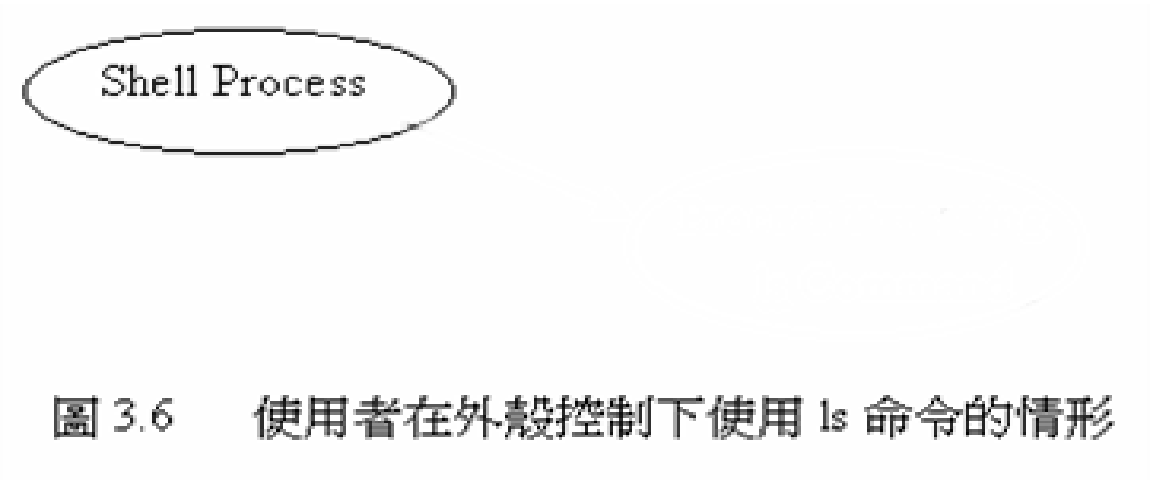


圖 3.6　使用者在外殼控制下使用 ls 命令的情形

處理元之運作（Process Operating）(2)

- 處理元之運算功能包含
  - 衍生（fork）
  - 執行（execve）
  - 等待（wait）
  - 離開（exit）
  - 放棄（abort）

- 它們均為系統呼叫。

衍生（fork）(1)

- 任何處理元呼叫fork( )，將使得系統衍生一個新的處理元。
- 被生出的處理元稱為子處理元（Children Process）。
- 生出處理元的處理元稱為父處理元（Parent Process）。
- 子處理元與父處理元長相一樣，有相同的程式碼並繼承父處理元的資源。

衍生（fork）(2)

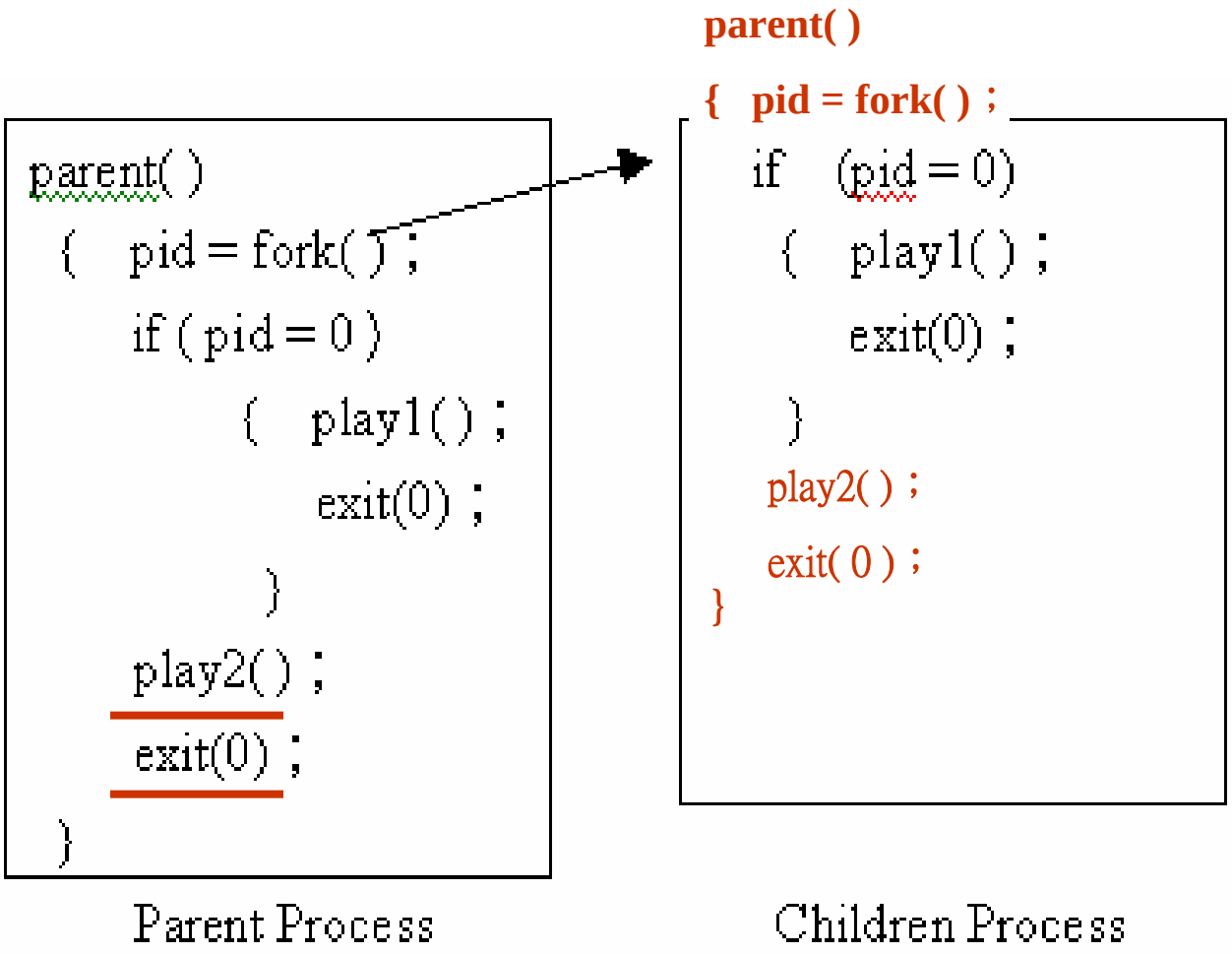


圖 3.7　父處理元衍生子處理元範例

衍生（fork）(3)

在系統內被fork( )的處理元之pid與父處理元之pid均不為0，但在子處理元內看到自己的pid為0，故子處理元與父處理元會執行不同程式碼。

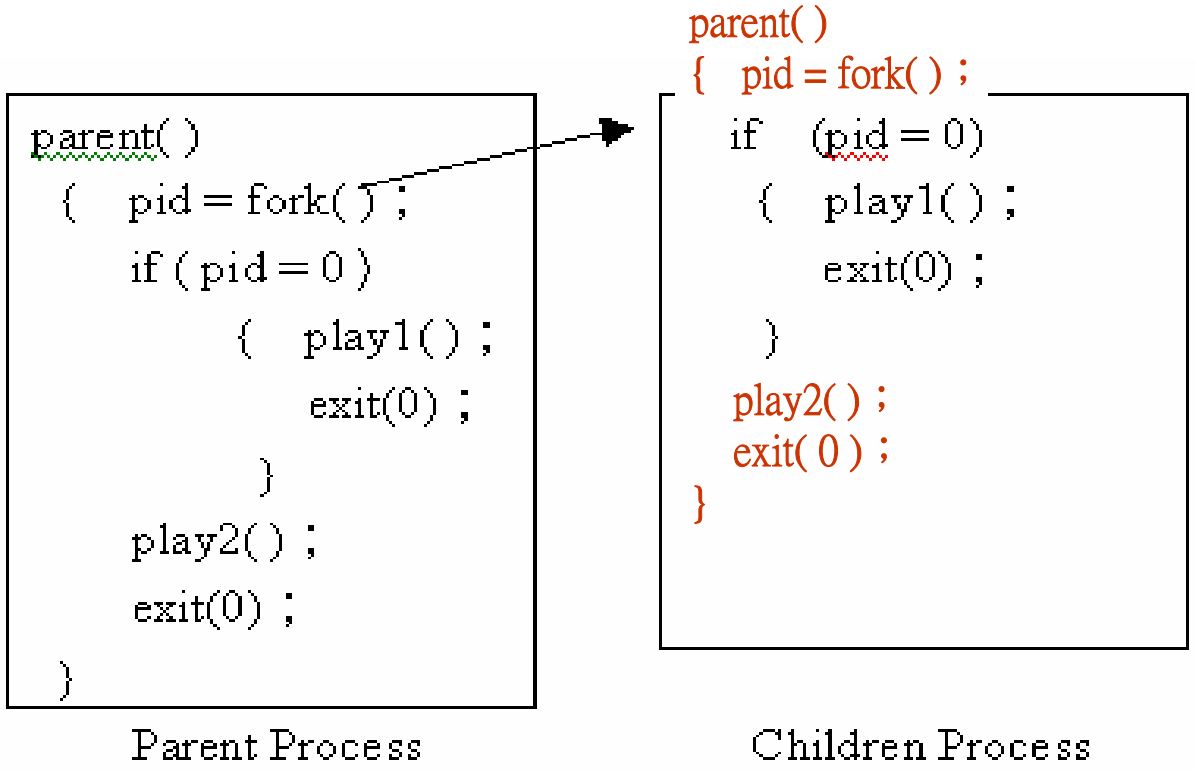


圖 3.7　父處理元衍生子處理元範例

衍生（fork）(4)

- 父處理元可以衍生許多子處理元，而子處理元亦可再衍生許多子處理元，所以它們父子之間的關係可能

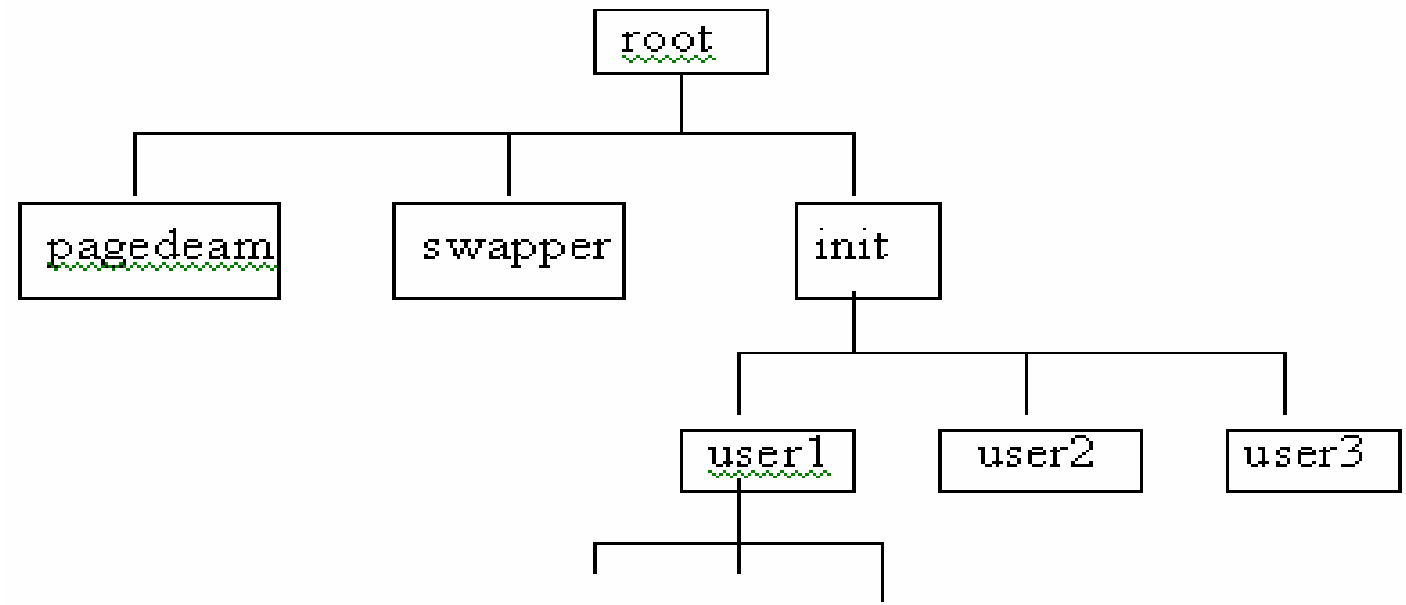
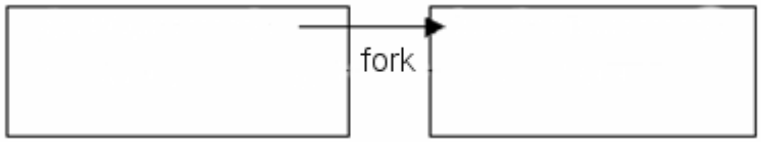


圖 3.8　處理元衍生之樹狀結構

執行（execve）

- 當一個處理元呼叫execve(“ls, null, null”)時，系統會衍生ls這個處理元，並把ls這個處理元之程式碼載入主記憶體內，且蓋掉呼叫execve( )的這個處理元；亦即用來呼叫execve( )這個處理元會自行結束執行，並將它的控制權交至ls這個被呼叫的處理元。

- execve( )這個系統呼叫與fork( )這個系統呼叫最大不同在於，執行fork( )系統呼叫後，父處理元與子處理元均存在系統內，而執行execve( )系統呼叫後，僅有被生出之處理元存在，原處理元結束執行。



等待（wait）及離開（exit）

- 等待（wait）
  - 若父處理元執行wait( )呼叫，則父處理元進入懸置狀態，當它的所有子處理元結束後，父處理元才由懸置狀態進入備妥狀態。

- 離開（exit）
  - 處理元欲正常結束，必須呼叫exit( )以通知作業系統正常結束，並將控制權交回作業系統。

放棄（abort）

- 作業系統或處理元可以呼叫abort( )，結束子處理元之執行。

- 通常在作業系統內，若父處理元結束工作，則不論子處理元是否繼續工作，作業系統均會放棄子處理元，亦即停止子處理元工作，這就是所謂連帶終止（Cascading Termination）。
- If a process terminates then all its children must also be terminated.

處理元之運作（Process Operating）(3)

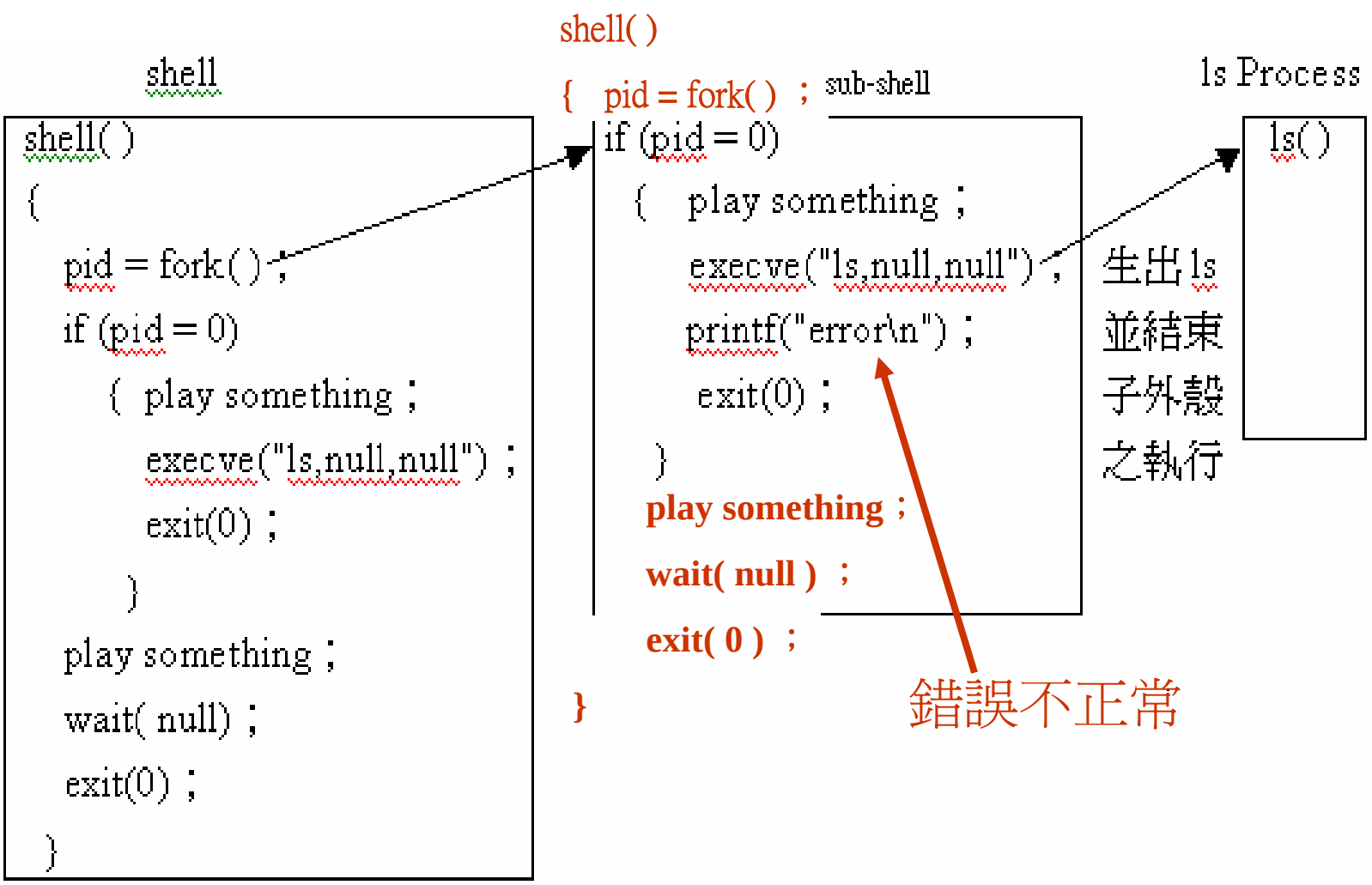


圖 3.9　外殼接受使用者命令，並執行使用者命令的程式碼

合作處理元(Cooperating Processes)(1)

4.4 合作處理元(Cooperating Processes)

- 在電腦系統內的處理元彼此是合作處理元（Cooperating Processes），所謂合作處理元是指處理元會影響別的處理元或被別的處理元影響 。

- The process can affect or be affected by the other processes executing in the system.

- 並行處理元(Concurrent Processes)
  - Processes execute concurrently.

- Independent Process

為什麼處理元要互助合作呢？

- 資訊共享（Information Sharing）
- 增快計算速度（Computation Speedup）
  - CPU process, I/O process
- 模組化（Modularity）
  - 因為OS是這樣設計的
- 方便性（Convenience）
  - User可以同時editing, printing, compiling

合作處理元(Cooperating Processes)(2)

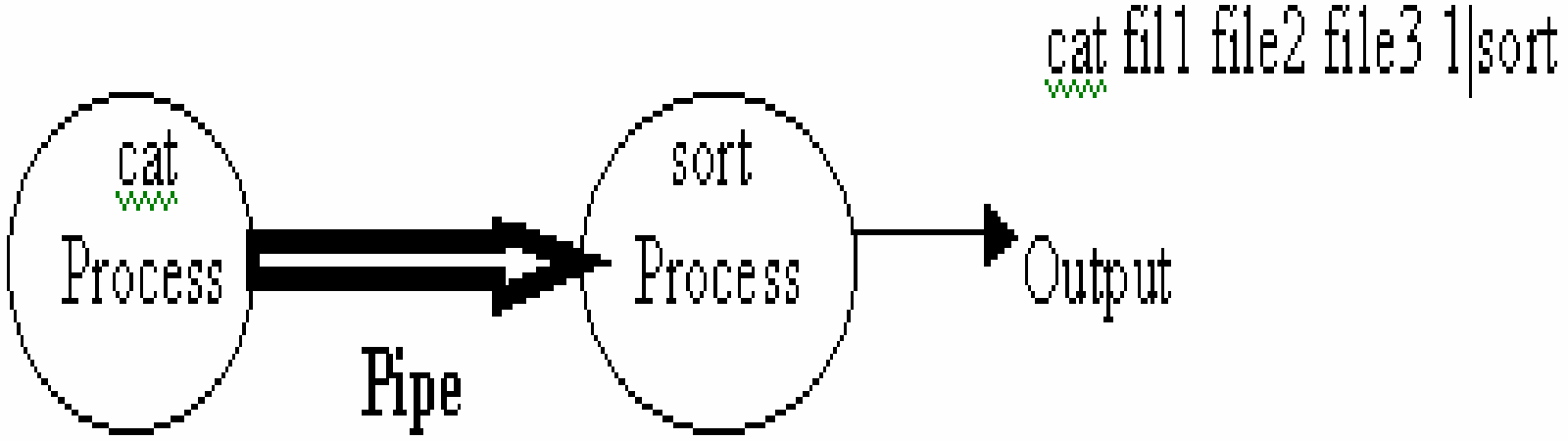


圖 3.10　使用管道之合作處理元

生產者與消費者（Producer and Consumer）模式(1)

- 合作處理元之合作方式，可以是一種生產者與消費者（Producer and Consumer）模式 。

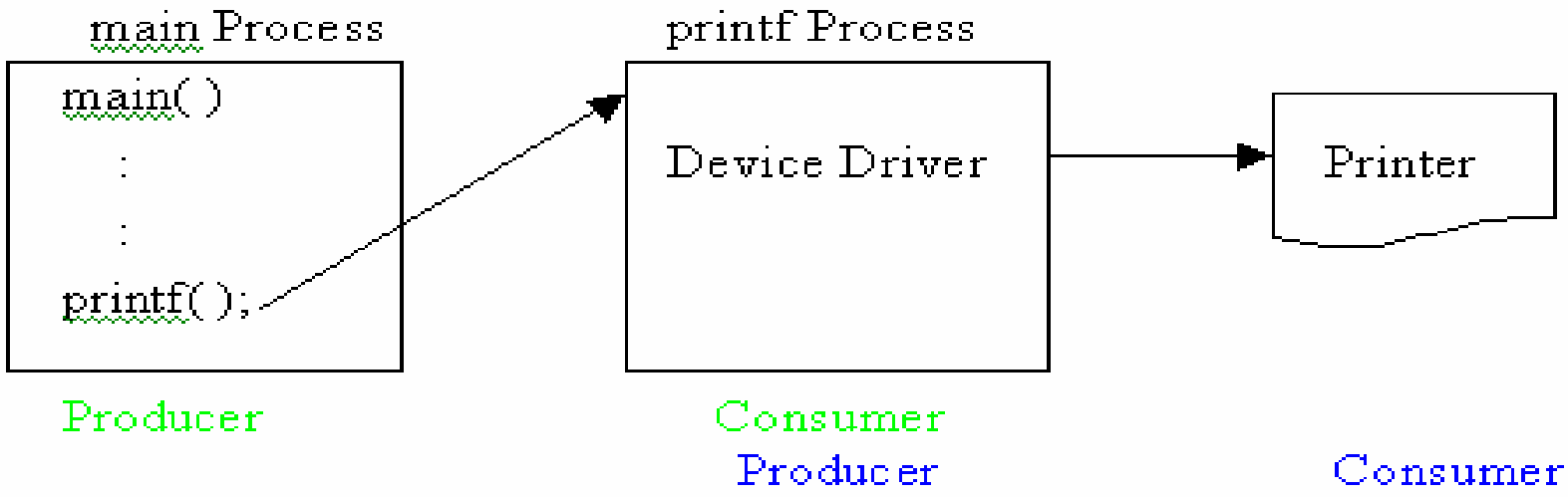


圖 3.11　生產者與消費者模式

生產者與消費者（Producer and Consumer）模式(2)

- 處理生產者與消費者處理元之通訊時，必須於程式內判斷是否會發生溢位（Overflow）或向下溢位（Underflow），以便適時處理問題。

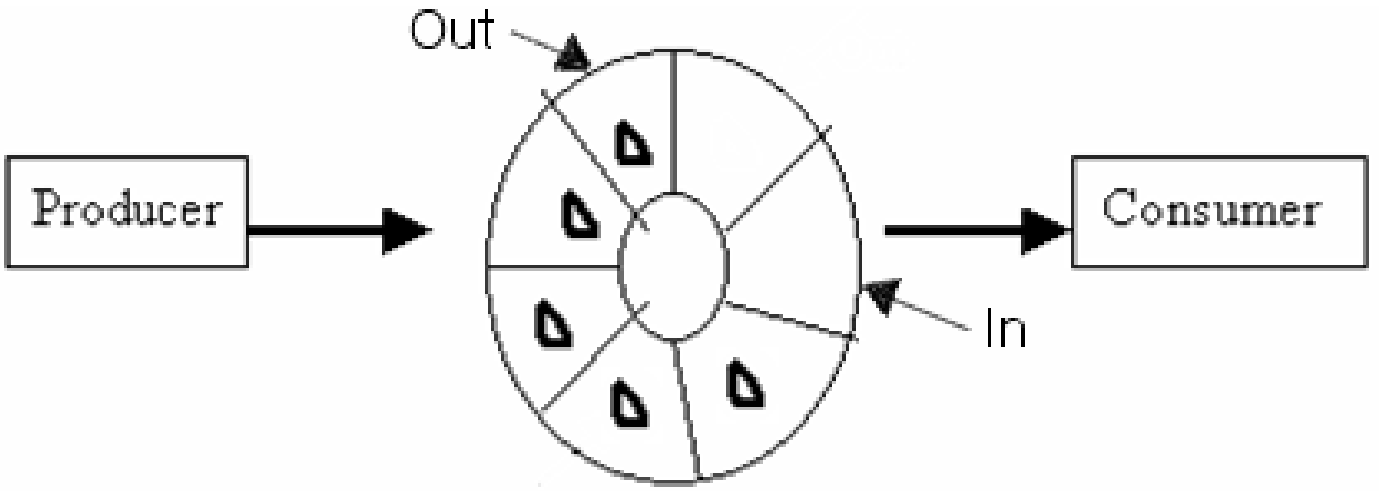


圖 3.12　生產者與消費者模式使用環狀緩衝區

生產者與消費者（Producer and Consumer）模式(3)

- 若Producer超過n筆則必須wait。
  - in+1 mod n = out, buffer is full.
- 若Consumer已拿空則必須wait。
  - in=out, buffer is empty.

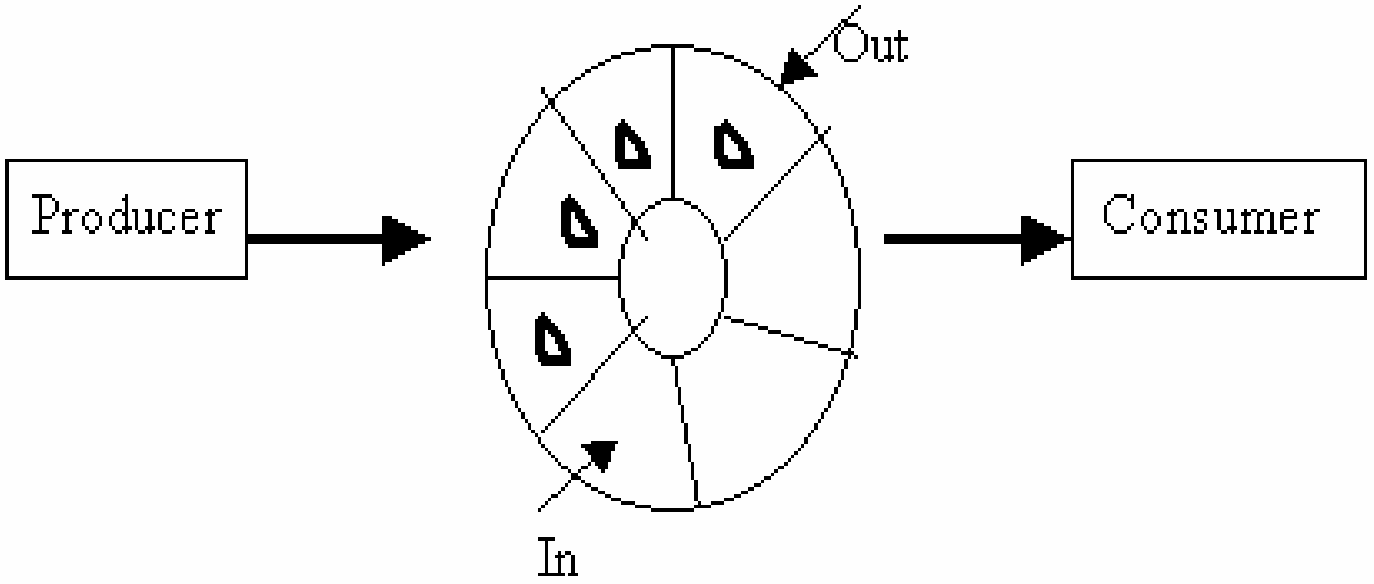


圖 3.12　生產者與消費者模式使用環狀緩衝區

執行線(Thread)(1)

4.5　執行線(Thread)

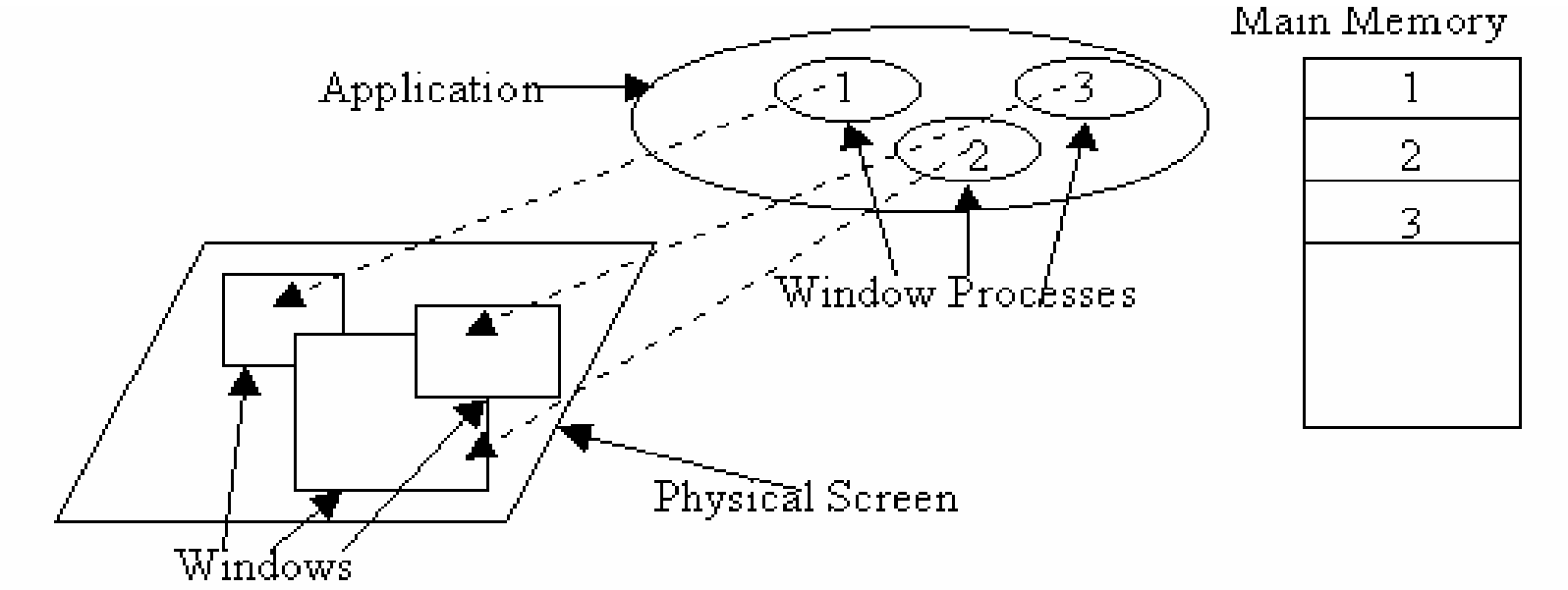


圖 3.14　　螢幕上同時開啓三個視窗

執行線(Thread)(2)

- 每個處理元內可以有一個或一個以上的執行線，這些執行線
  - 共用程式碼（Code Section）
  - 資料段（Data Section）
  - 處理元內所要到的資源

- 而每個執行線均擁有自己的
  - 程式計數器（Program Counter）
  - 暫存器組（Register Set）
  - 堆疊空間（Stack Space）。

執行線(Thread)(3)

- A basic unit of CPU utilization, and consists of a program counter, a register set, and stack space. It shares with peer threads its code section, data section, and operating system resources.

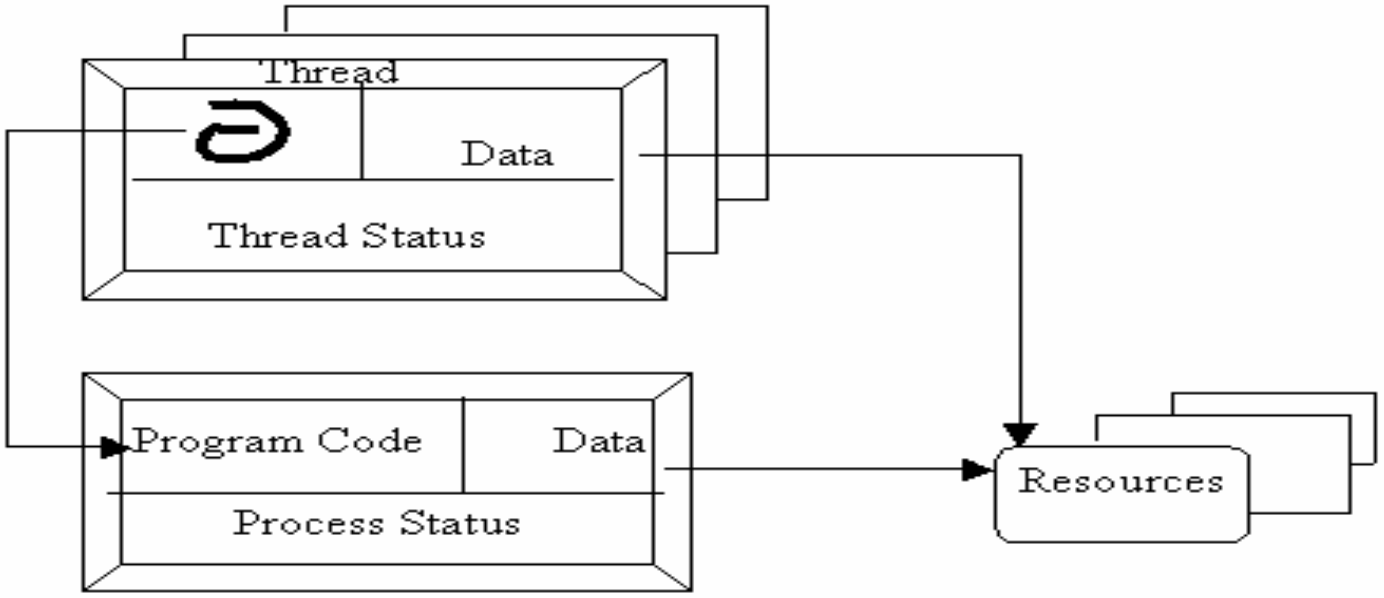


圖 3.15　使用執行線

執行線(Thread)(4)

- 一個處理元內的所有執行線共用地址空間（Address Space），此處的地址空間是指程式碼、資料段及資源，

- 因此它們在作環境切換（Context Switch）時，不用切換或複製地址空間，僅需儲存每個執行線之程式計數器、暫存器組及堆疊空間，所花的代價相對的比處理元作環境切換輕很多。

- 執行線又被稱為輕量級處理元（Light Weight Process），簡稱LWP。

- 而處理元被稱為重量級處理元（Heavy Weight Process），簡稱HWP。

執行線(Thread)(5)

- The producer and consumer could be threads in a task. Little overhead is needed to switch between them.

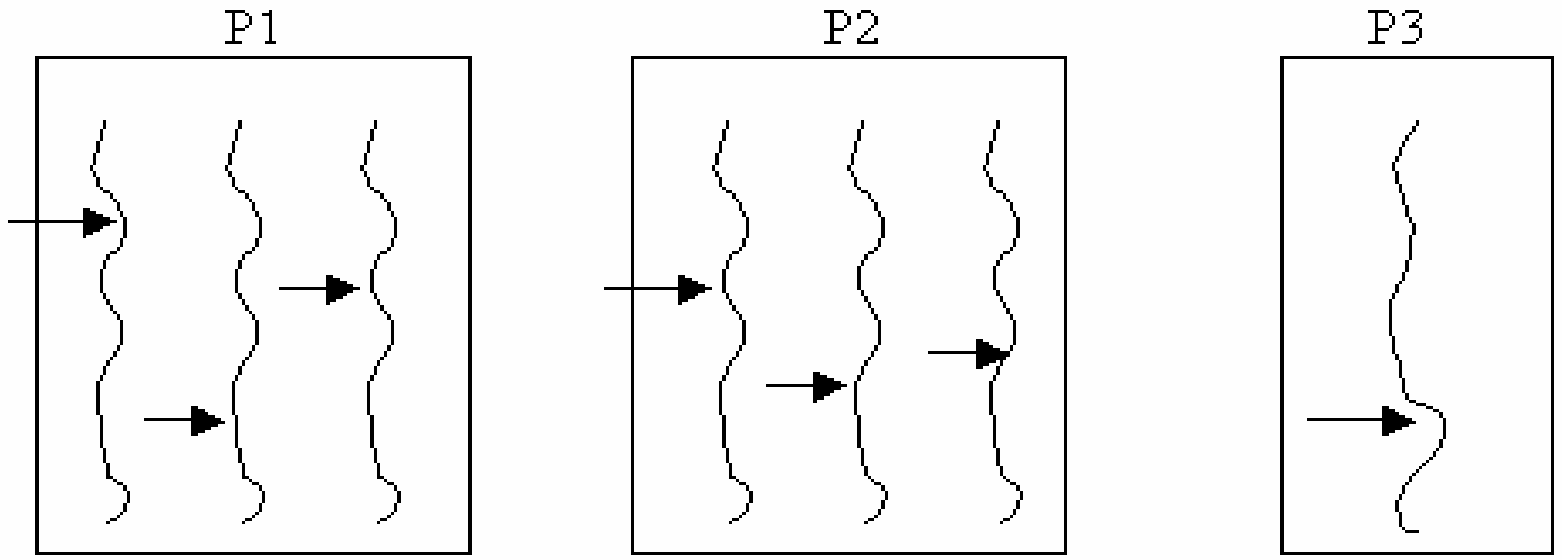


圖 3.16　處理元之執行線

執行線(Thread)(6)

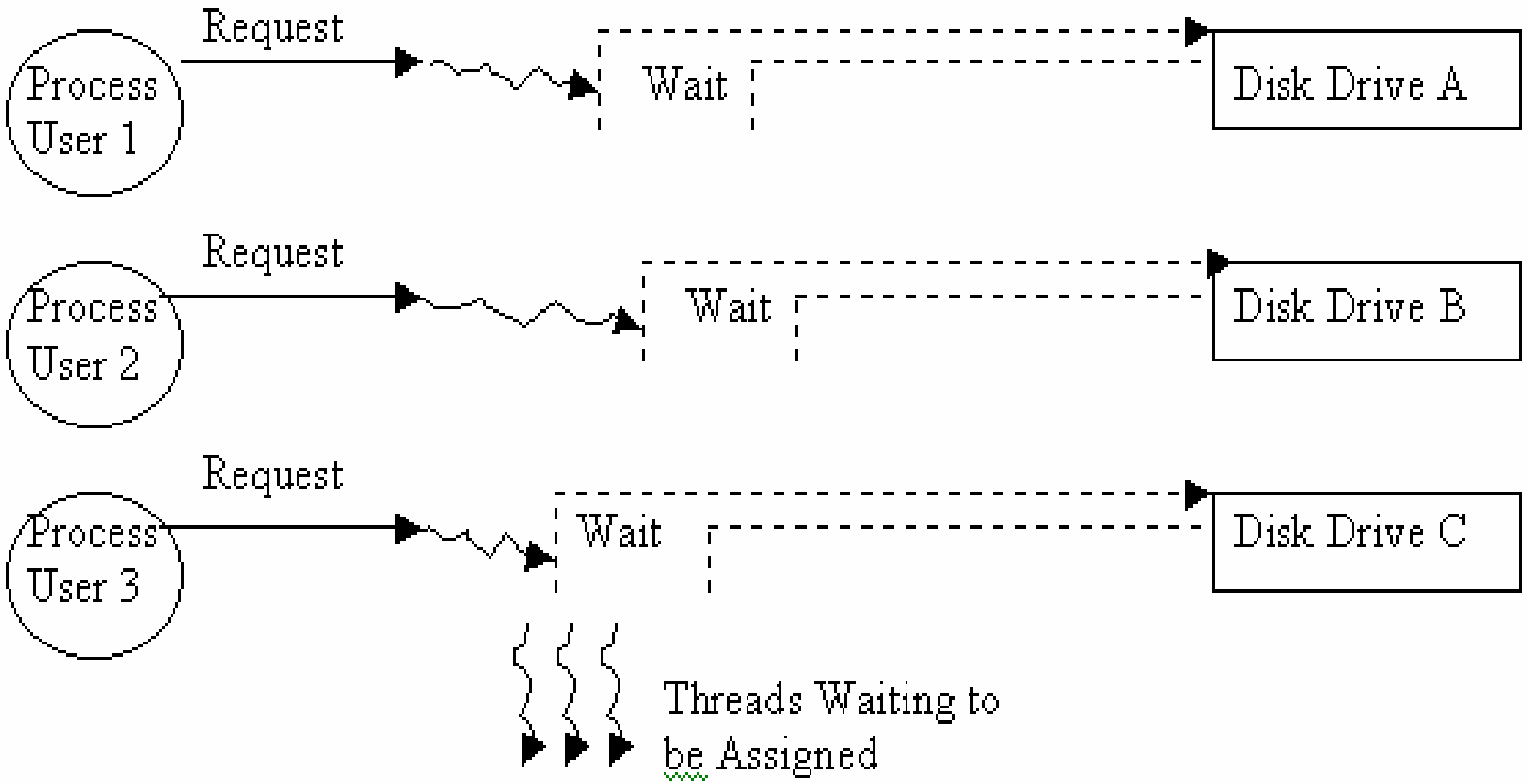


圖 3.17　執行線應用於磁碟輸出 / 輸入

執行線(Thread)(7)

- 範例：假如我們寫了一個試算表（Spread Sheet）的程式，在這個程式內有三個執行線，分別為計算試算表的執行線、在螢幕上顯示資料的執行線、以及接受使用者命令並執行該命令的執行線。

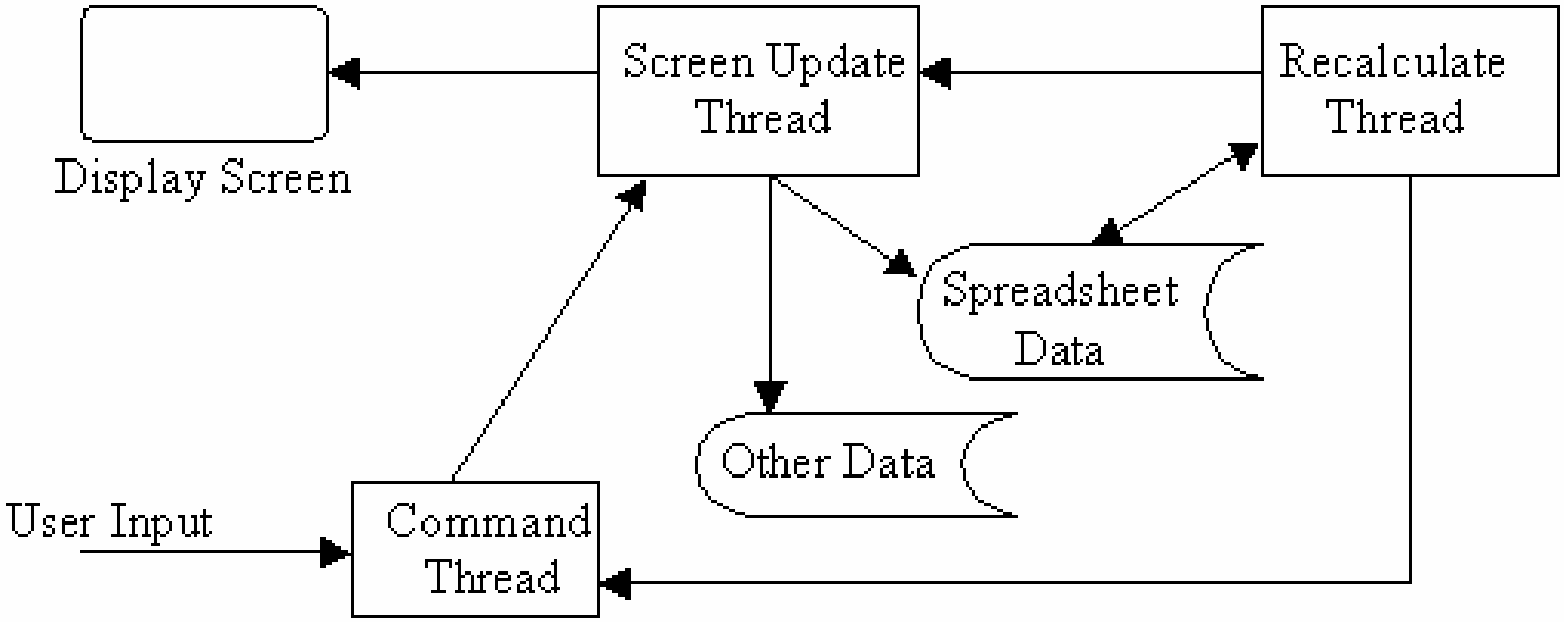


圖 3.18　使用執行線執行試算表程式

執行線的特性

- 執行線有下列特性
  - 它的衍生及消失的代價不高，因為它不需要長出新的地址空間，其地址空間隨著處理元出生或消失。
  - 執行線相互間可以非常快速的作環境切換。
  - 執行線相互間共用地址空間，所以資料是共享的，不必使用處理元內部通訊傳遞訊息。
- 由於執行線共用地址空間的關係，所以它的缺點是無法對共享資料作保護。

撰寫執行線的範例

```
t1( )
{
    for i = 1 to 5
        printf("1");
}
t2( )
{
    for j = 1 to 5
        printf("2");
}
main( )
{
    create_thread(t1);
    create_thread(t2);
    wait( );
}
```

圖 3.19　撰寫執行線的範例

使用者執行線（User Thread）及核心執行線（Kernel Thread)(1)

- 使用者執行線
  - 使用者應用程式所撰寫的執行線。
  - 使用者執行線程式庫內通常會提供執行線排程器（Thread Scheduler），以便自行於處理元內對執行線進行排程工作，因此在執行線進行環境交換過程中，電腦內不必產生中斷去通知作業系統核心，也不必作處理元環境交換的工作。

- 核心執行線
  - 為作業系統核心使用的執行線。
  - 使用者必須透過系統呼叫使用核心執行線，所以必須經由中斷執行核心執行線，它必須花費中斷的代價，但共用地址空間的優點仍然存在。

使用者執行線（User Thread）及核心執行線（Kernel Thread)(2)

- 使用者執行線是依附在處理元內，所以中央處理器排程器仍以處理元為單位進行排程及環境切換工作，當某一個處理元輪到中央處理器時，它將它所輪到的時間分享給它的執行線使用，例如時間片段為100ms，故某個處理元分配100ms時間，而它有五個執行線，則每個執行線分配20ms執行。

- 中央處理器在排程時，是以核心執行線為單位分配時間片段。例如若某個核心處理元有五個核心執行線，則此處理元分到五個時間片段，也就是每個執行線各有一個時間片段。

使用者執行線（User Thread）及核心執行線（Kernel Thread)(3)

- 使用者執行線有一個很大的缺點，那就是若某個處理元內有一個執行線使用系統呼叫進入懸置狀態，則整個處理元包含它的所有執行線均一起懸置，這是因為使用者執行線程式庫內的執行線排程器自行對執行線排程，從作業系統角度來看，作業系統不知道發生什麼事，所以只好讓大家一起懸置了。

- 有些作業系統在核心內又設置一個執行線排程器，當使用者執行線進行系統呼叫而懸置時，此執行線排程器出面進行排程 。

- 由於每個核心執行線是各自排程，所以核心處理元內若某一個執行線懸置，它不會造成其他的執行線被懸置。

Multithreading Models

- Many to one
  - Many user threads map to one kernel thread.
  - Solaris old version.
- One to one
  - Windows NT, OS/2
- Many to many
  - Solaris OS
  - 一個process有多個kernel threads對應,故一個user thread block某個kernel thread時,只有此kernel thread所對應之user threads被block,其他kernel threads仍然可以work.

Sun Solaris作業系統解決使用者執行線與核心執行線介面的方法

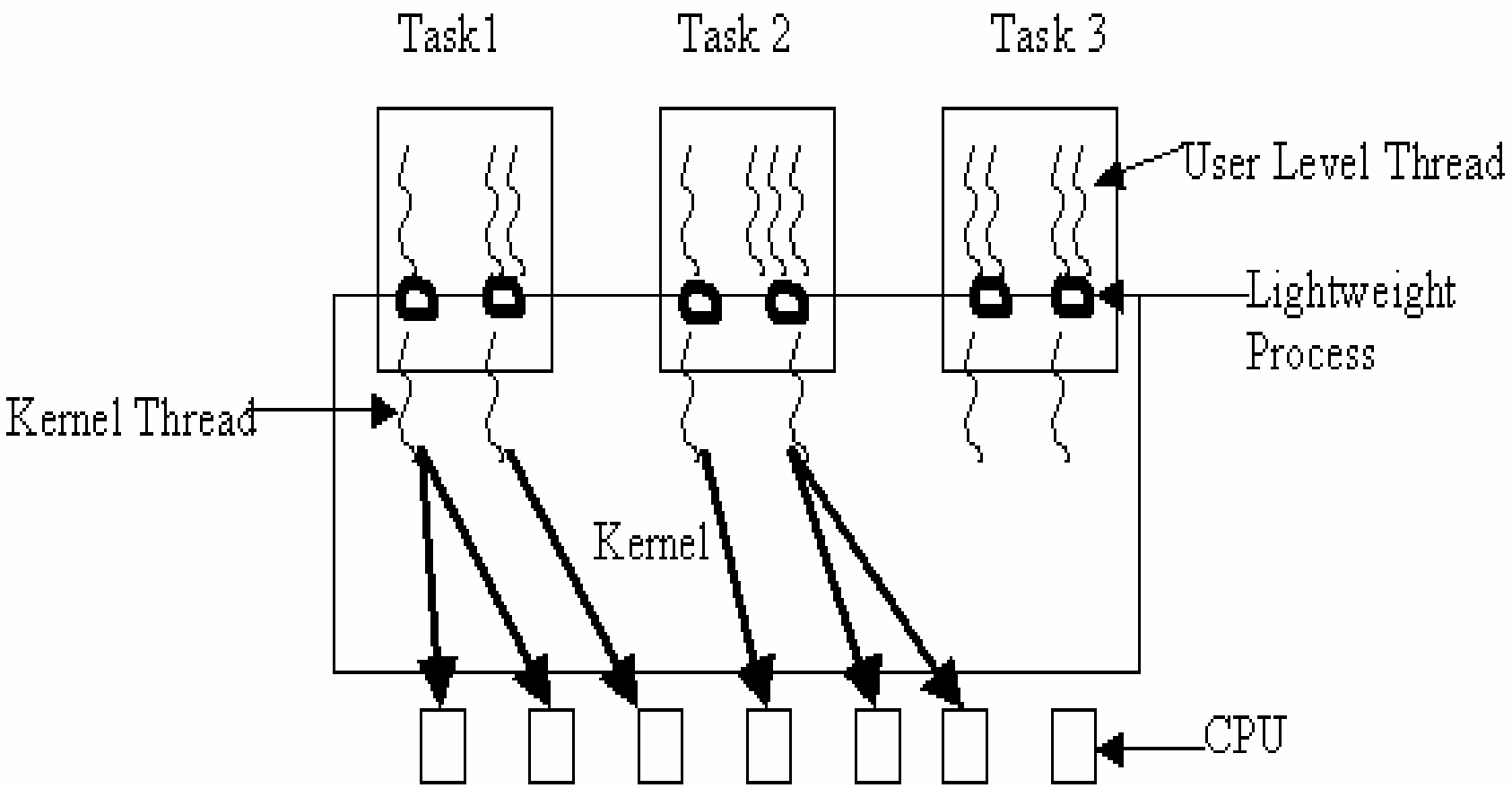


圖 3.20　Sun Solaris 作業系統之使用者執行線與核心執行線介面

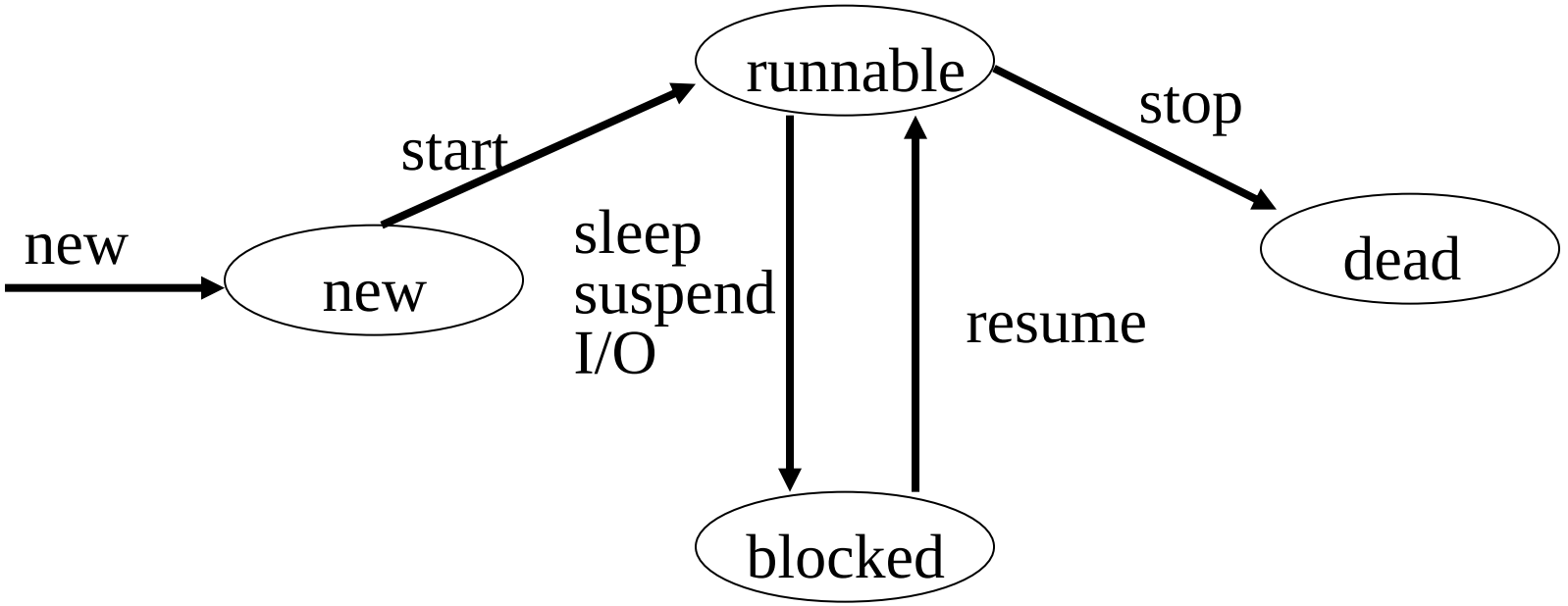
Solaris Threads

- Many user threads map to one lightweight process.
- One lightweight process maps to one kernel thread.
- User threads 依附在lightweight process,並均分 lightweight process 之time slice.
- Lightweight process及kernel process由system排程.

Java Threads

```
Class worker1 extends thread
{   public void run() {
        system.out.println(“I am worker thread”);}
}
public class first
{    public static void main(string args[]){
        worker1 runner = new worker1();
        runner.start();
        system.out.println(“I am main thread”);}
}
```

Java Thread States



Thread Scheduling

- Process local scheduling
  - Thread scheduling is done local to the application.

- System global scheduling
  - Process scheduling is done by the system.

Java Based Round Robin Scheduler(1)

- Priority
  - Round robin scheduler is 6,
  - Default thread is 2,
  - Thread at run state is 4.

- Algorithm
  - Scheduler select one thread from queue, and assign its priority to 4.
  - Scheduler go to sleep (Time slice).
  - Selected thread 擁有CPU執行.
  - Scheduler wakeup佔有CPU,並將此selected thread之priority改為 2.

Java Based Round Robin Scheduler(2)

```
Public class scheduler extends thread
{          :
   private void schedulersleep(){
       try {
             thread.sleep(timeslice);
           } catch(interruptedexception e) {}
   }
   public void run() {
        thread current;
        this.setpriority(6);
        while (true) {
             current =(thread)queue.getnext();
             if ((current != null) && (current.isalive() )) {
                  current.setpriority(4);
                  schedulersleep();
                  current.setpriority(2);
```

處理元內部通訊（Inter Process Communication）(1)

4.6 處理元內部通訊

- 子處理元除了繼承父處理元之資源之外，當子處理元結束執行之後，必須透過作業系統通知父處理元，這就是一種處理元內部通訊（Inter Process Communication）。

- 另外某個處理元可能需要將某些資訊傳給另一個處理元，這也是一種處理元內部通訊。

處理元內部通訊（Inter Process Communication）(2)

- 所謂處理元內部通訊，是一種用來讓處理元之間通訊（Communication）及同步（Synchronize）的機制，簡稱為IPC。
- IPC provides a mechanism to allow process to communicate and to synchronize their actions.

- 合作處理元便是透過處理元內部通訊機制來作通訊及同步工作。

處理元內部通訊（Inter Process Communication）(3)

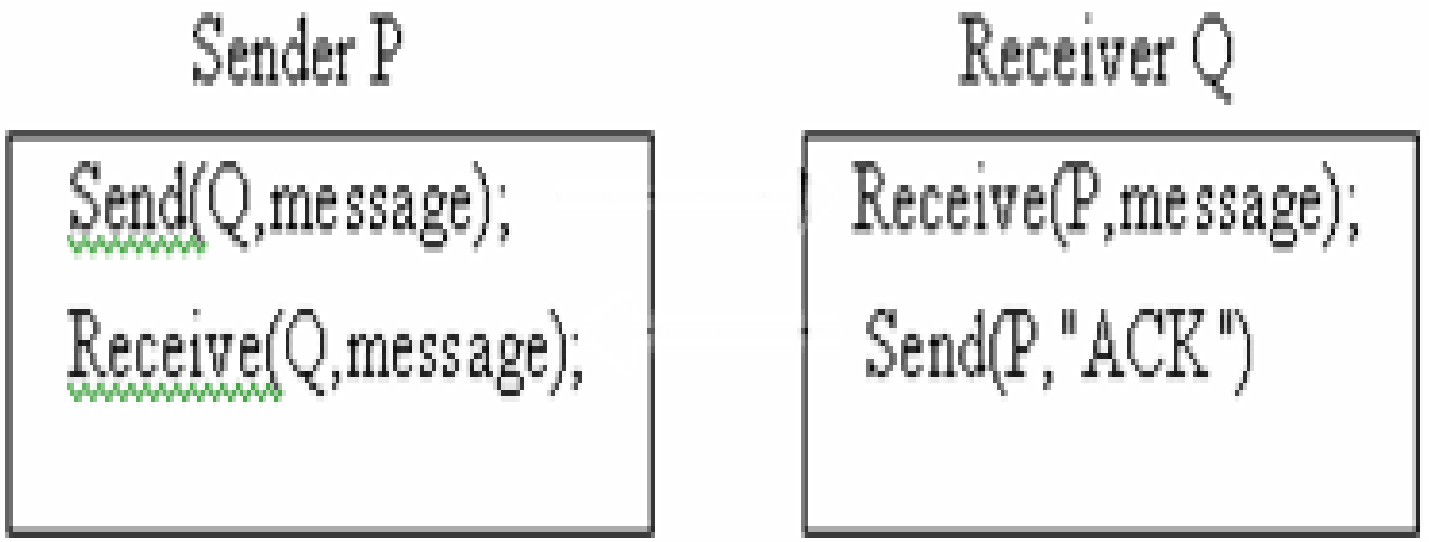


圖 3.13　二個處理元使用內部通訊機制傳送訊息

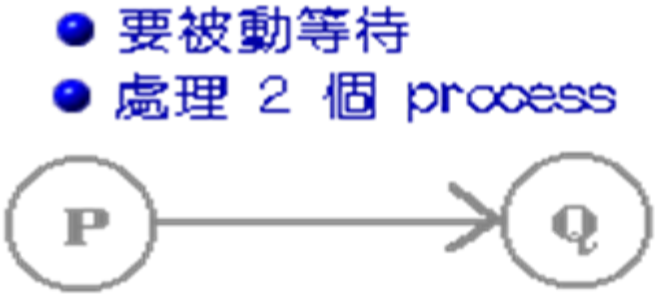
處理元內部通訊（Inter Process Communication）(4)

- 遠端程序呼叫（Remote Procedure Call）簡稱RPC，有時又叫遠端處理元通訊（Remote Process Communication），它與處理元內部通訊非常相似，只不過它是指分別在不同電腦內的二個處理元彼此進行通訊並達成同步。

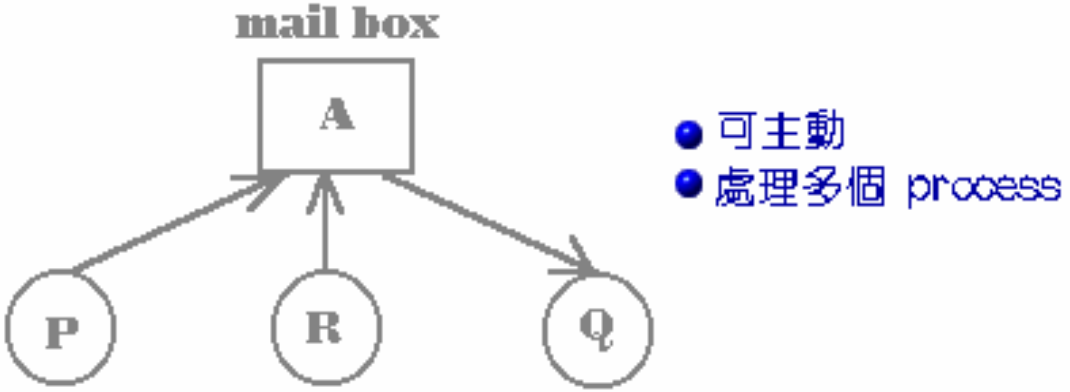
- 由於網路的發達，電腦與電腦的溝通情形愈加普遍，因此有時把遠端程序呼叫、遠端處理元通訊、及處理元內部通訊全都以處理元內部通訊稱呼。

處理元內部通訊（Inter Process Communication）(5)

- Direct Communication
  - send(Q, message)
  - receive(p,message)



- Indirect Communication
  - send(A, message
  - receive(A,messa



Inter Process Communication Methods

- Common event flags
- Mail box
- Shared memory
- shared file

Buffering

- Zero Capacity (No buffering)
  - Sender must wait until the recipient receives the message.
  - Rendezvous Synchronization
- Bound Buffer
  - Size n
- Unbounded Capacity

Asynchronous Communication

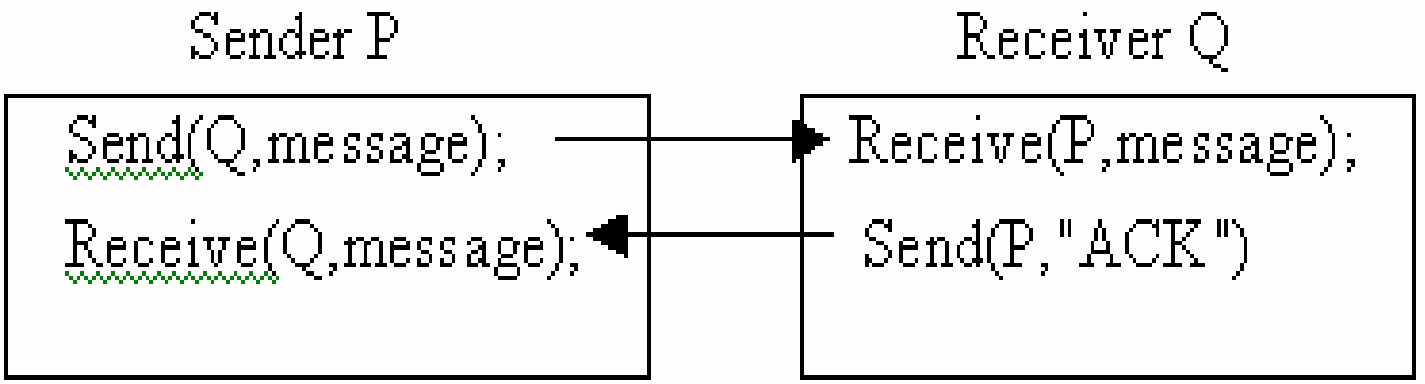


圖 3.13　二個處理元使用內部通訊機制傳送訊息

- 使成為同步
- 必須先run Q再run P，否則會lost message。
- Sender be blocked until ACK is sent back from receiver.

Message Lost

- 造成無限等待。
- Most common detection method is to use time out.

Scrambled Messages

- Noise in the message
  - Parity
  - Checksum
  - 週期性循環檢查(Cyclical Redundancy Check)
    - 網路傳輸、ZIP 壓縮檔案、磁碟存取
  - 錯誤校正編碼(Error Correction Coding)

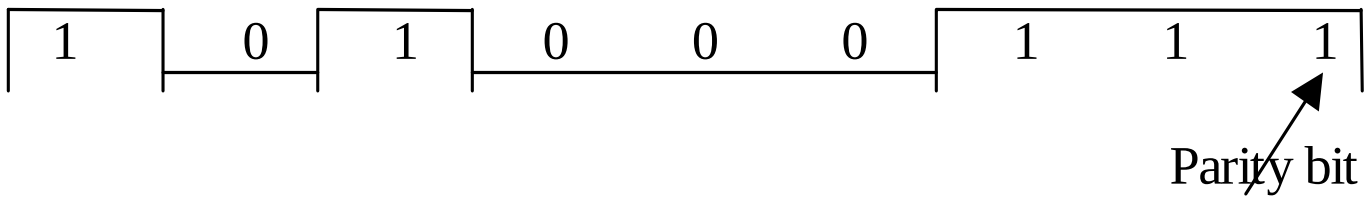


圖 7.15　奇同位檢查的範例

Exception Condition Handling (Error Recovery)

- 系統通知。

- Time out.

主從式模式（Client/Server Model）

- 在分散式系統環境下，於某部電腦內要求（Request）另一部電腦提供服務，並將服務的結果傳回，這就是一種主從式模式（Client/Server Model）。

- 要求服務的電腦稱為客戶端（Client）。

- 提供服務的電腦稱為伺服端（Server）。