Chapter 3:行程概念
Process Concept

Chapter 3: Process Concept

- 行程的觀念
- 行程排班
- 行程的操作
- 行程間通訊
- IPC系統的範例
- 客戶-伺服器的通信

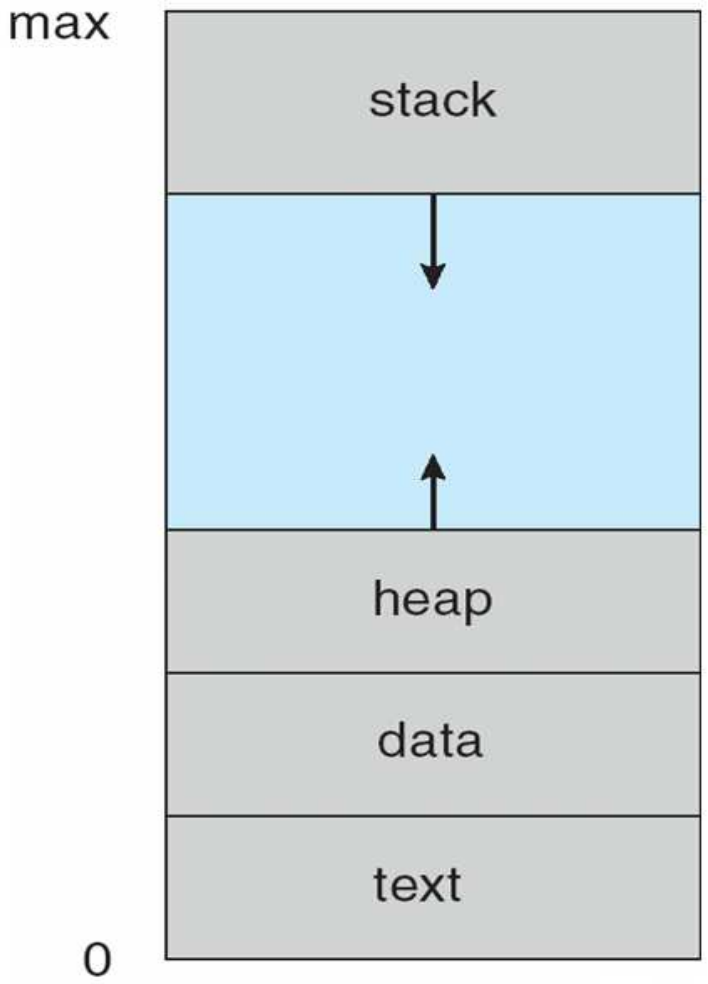
章節目標

- 介紹行程的觀念—執行中的程式
- 描述行程的不同特性，包含排班、行程的產生和結束
- 探討使用共用記憶體和訊息傳遞的行程間通信
- 描述客戶—伺服器系統的通信

行程的觀念

- 作業系統執行各種程式：
  - 批次系統 – jobs
  - 分時系統 – programs或tasks
- 行程是作業系統管理與分配資源的基本單位
- 行程 (Process) –執行中的程式
- 一個行程是程式碼、程式計數器 (Program Counter, PC)、暫存器、堆疊 (Stack)、及資料區 (Data Section) 等之集合體。
  - 程式碼也稱為本文區(text section)
  - 堆疊(stack)包含暫用資料
    - 函數的參數、返回位址，以及區域變數(local variables)
  - 資料區間(Data section)包含全域變數(global variables)
  - 堆積(Heap)包含了執行期間動態配置的記憶體

行程的記憶體結構



行程的觀念

- 行程需要許多資源，包括記憶體、暫存器、CPU時間、及 I/O 裝置等資源才能完成工作。
- 程式(program)是儲存在磁碟的被動個體(passive entity)(可執行檔)，行程(process)則是主動個體(active entity)
  - 當可執行檔案載入記憶體時，程式變成行程
- 執行檔案可以藉由按下滑鼠按鈕，或是在命令列輸入可執行檔案的檔名
- 一個程式可能有數個行程
  - 考慮幾個使用者執行相同的程式
  - 一個使用者可能執行許多份網頁瀏覽程式

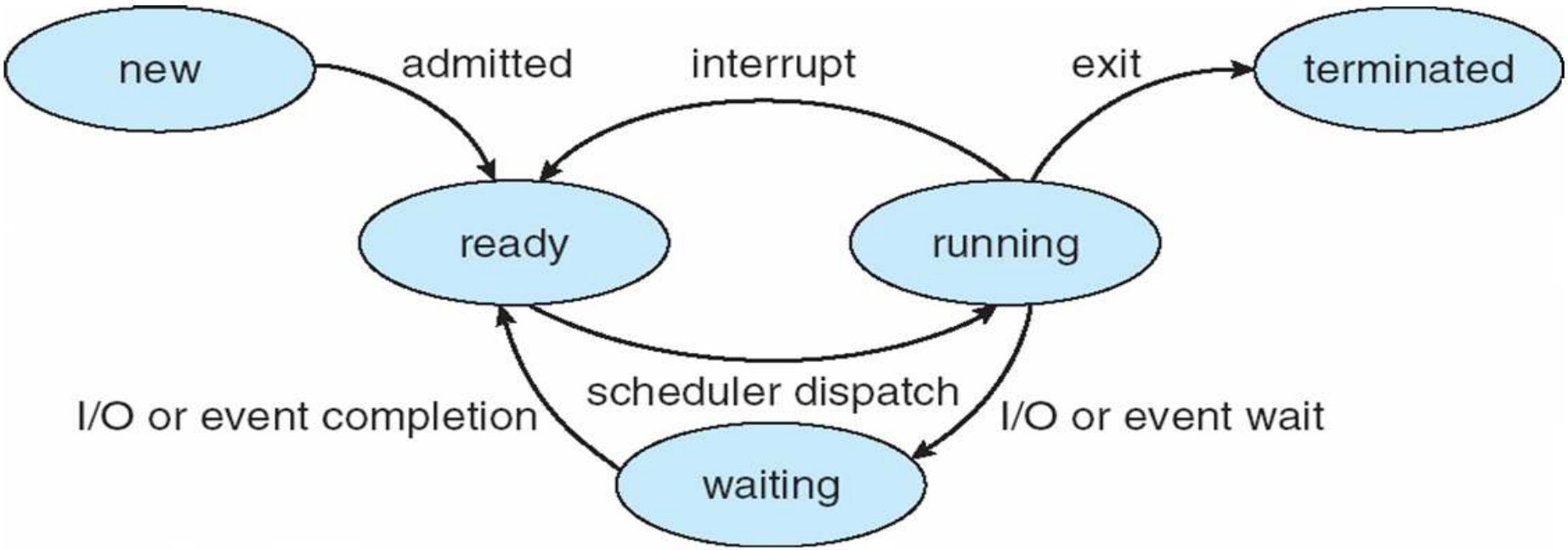
行程與程式的主要差異

- 行程是主動的個體，程式是被動的個體
- 行程是暫時的，程式是長存的
- 行程與程式的組成內容不同

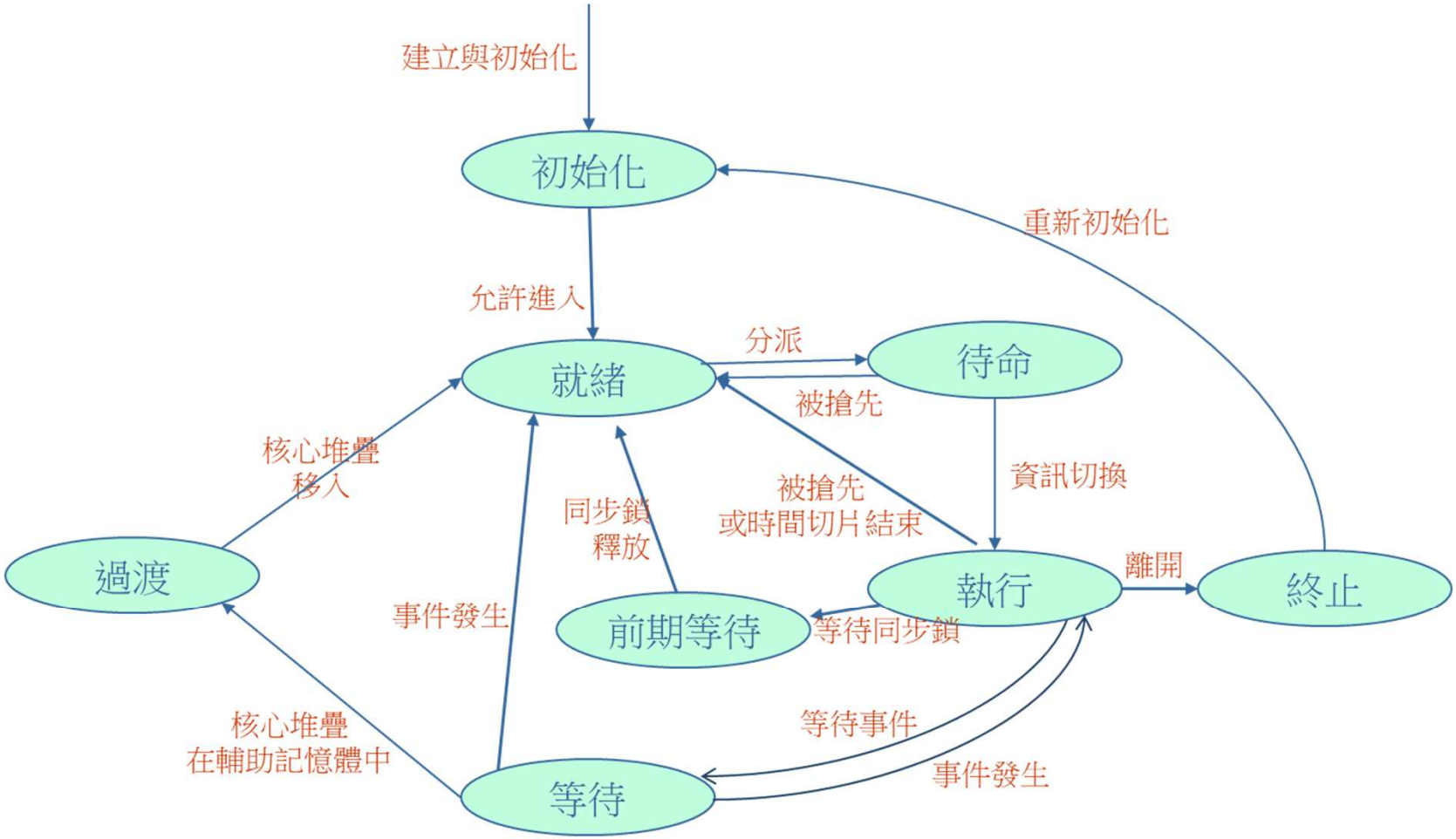
行程狀態(Process State)

- 當行程執行時，它會改變狀態，一般可能的狀態有:
  - 新建立 (New) 狀態：行程剛產生時的狀態
  - 就緒(ready)：行程已準備好，隨時可以被執行，亦即正在等待被分派一個處理器來執行。
  - 執行(running)：行程之指令正在被執行
  - 等待(waiting)：等待某件事件的發生，通常是等待I/O 完成。
  - 結束(terminated)：行程結束執行。
- 任何時間一個處理器僅會執行一個行程
- 可能會有許多行程是處於就緒及等待狀態

行程狀態圖


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Windows的執行緒狀態

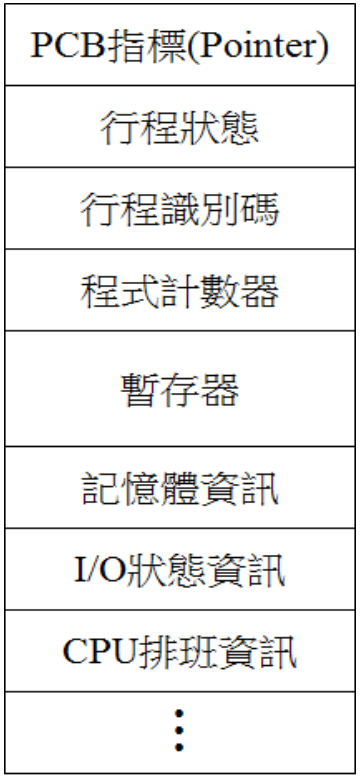


行程的建立及結束

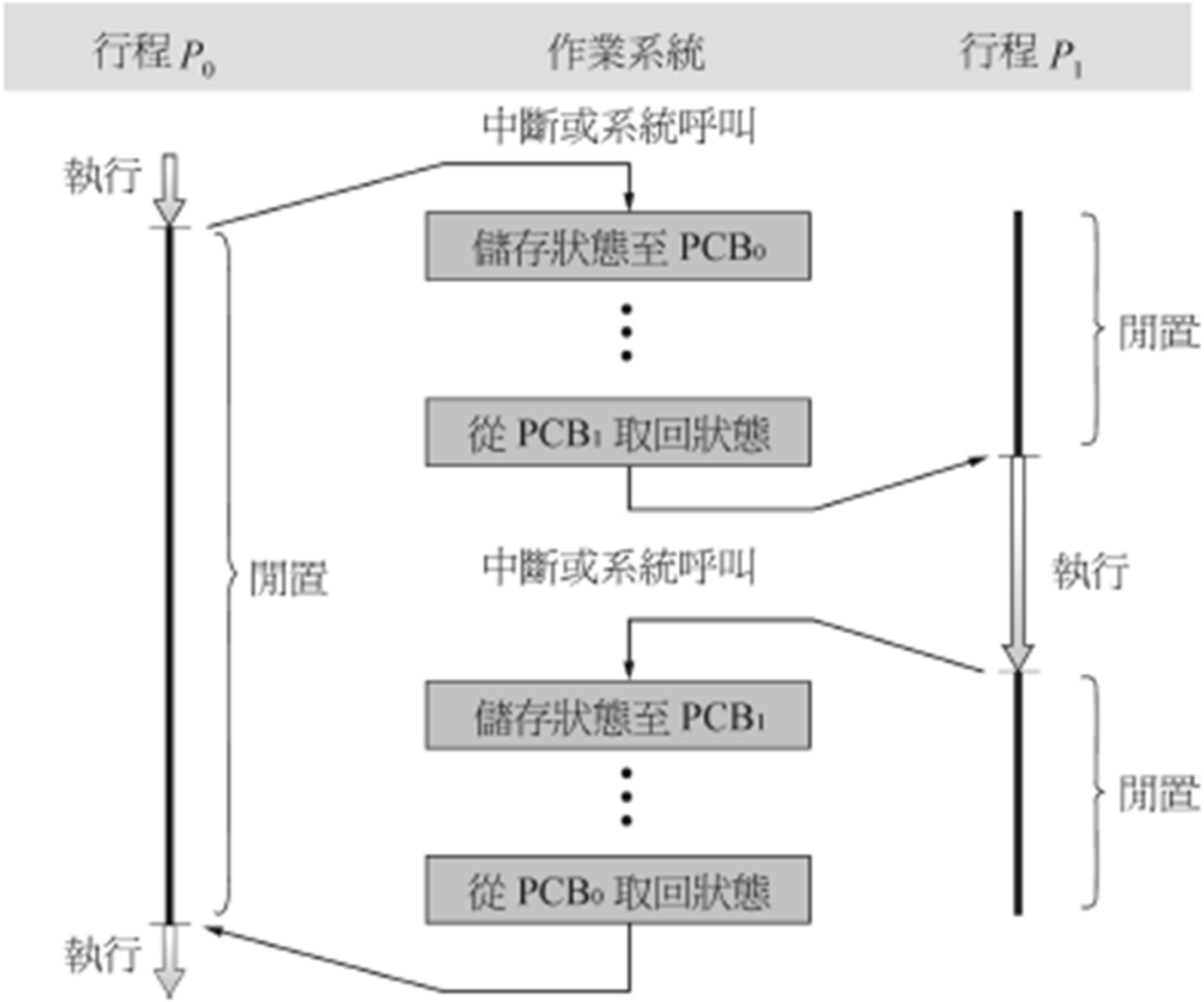
- 當行程處於建立狀態時
  - 作業系統已經完成行程控制資訊的建立，但是還沒有將行程移入記憶體中。
  - 此設計技巧有助於作業系統區分行程控制與記憶體配置的工作
    - 當系統內的記憶體或CPU處理能力不足時，OS可知道行程的存在，但不必立即開始執行。
- 產生新行程的時機：如使用者登入、使用者執行某個程式、請求系統提供服務 、由現有行程產生等。
- 當行程處於結束狀態時
  - 代表行程已經不能執行，但是它的一些相關資訊與表格仍然可以暫時保存在作業系統中
- 行程結束的可能原因：如任務完成 、錯誤 發生、I/O失敗 、外力終止 、父行程終止 、父行程要求等。

行程控制區塊(PCB，Process Control Block)

- 是作業系統核心中一種資料結構，用來紀錄行程相關的資訊，又稱為任務控制區塊(task control block)，內容包含：
  - 行程狀態：新建立 (New)、就緒 (Ready)、執行、等待、或暫停 (Halted) 等狀態。
  - 行程識別碼：作業系統分配的一個識別碼。
  - 程式計數器：程式計數器 (Program Counter) 記錄著該程式下一個準備要執行之指令位址 。
  - 暫存器：包括累積暫存器 (Accumulator)、索引暫存器 (Index Register)、堆疊指標暫存器 (Stack Pointer)、及通用用途暫存器。
  - 記憶體資訊：包含 基底 (Base) 與限制 (Limit) 暫存器的值、分頁 (Paging) 資訊、及虛擬記憶體 (Virtual Memory) 資訊等。
  - I/O 狀態資訊：該行程之 I/O 裝置之串列、及開啟之檔案之串列等資訊。
  - CPU 排程資訊：一個行程之執行優先權 (Priority)、排程佇列 (Scheduling Queue) 之指標 及其它排程參數。



CPU 行程切換示意圖


行程 P_0
作業系統
行程 P_1
中斷或系統呼叫
執行
儲存狀態至 PCB_0
從 PCB_1 取回狀態
閒置
閒置
中斷或系統呼叫
執行
儲存狀態至 PCB_1
從 PCB_0 取回狀態
閒置
執行

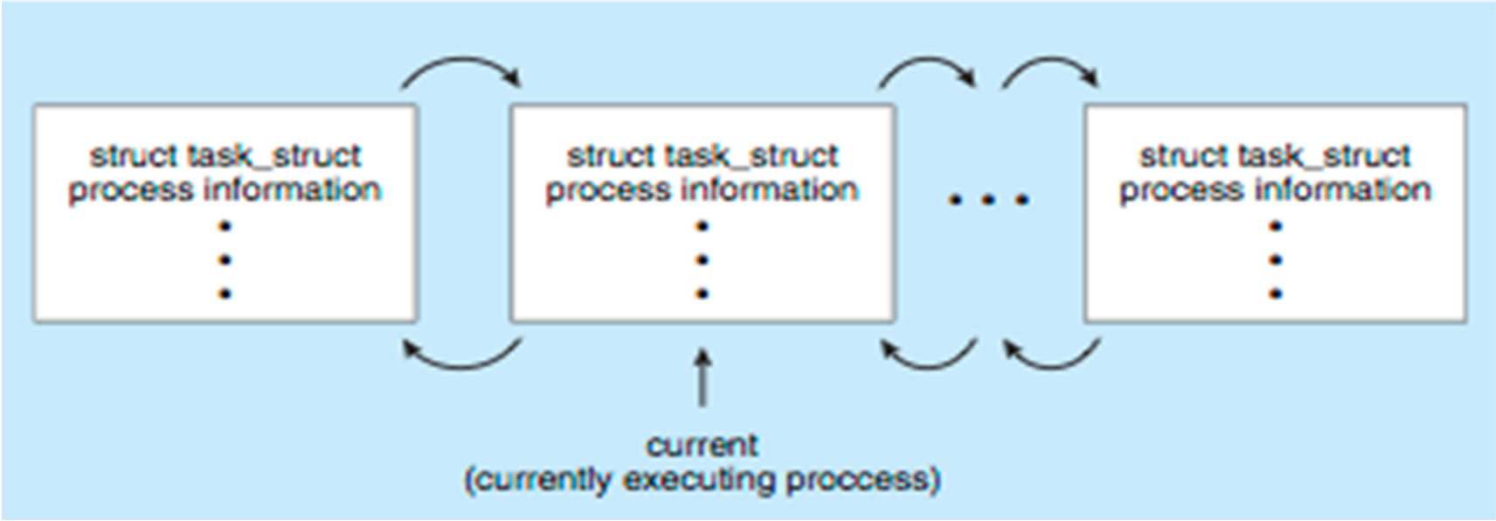
執行緒(Threads)

- 一個行程可以有一個以上的執行緒。因此，藉由多個執行緒，一個行程可以併行式地 (Concurrently) 同時執行多個工作。
- 執行緒 (Thread) 是作業系統能夠進行CPU配置的最小單位。執行緒又可被稱為輕量行程 (Lightweight Process)。
  - 擁有自己的程式計數器、暫存器、與堆疊空間
  - 和同一行程的其他執行緒共享相同的記憶體位址空間、程式區段、資料區段，和一些系統資源。

LINUX的行程表示

- 以C語言的 structure task_struct表示

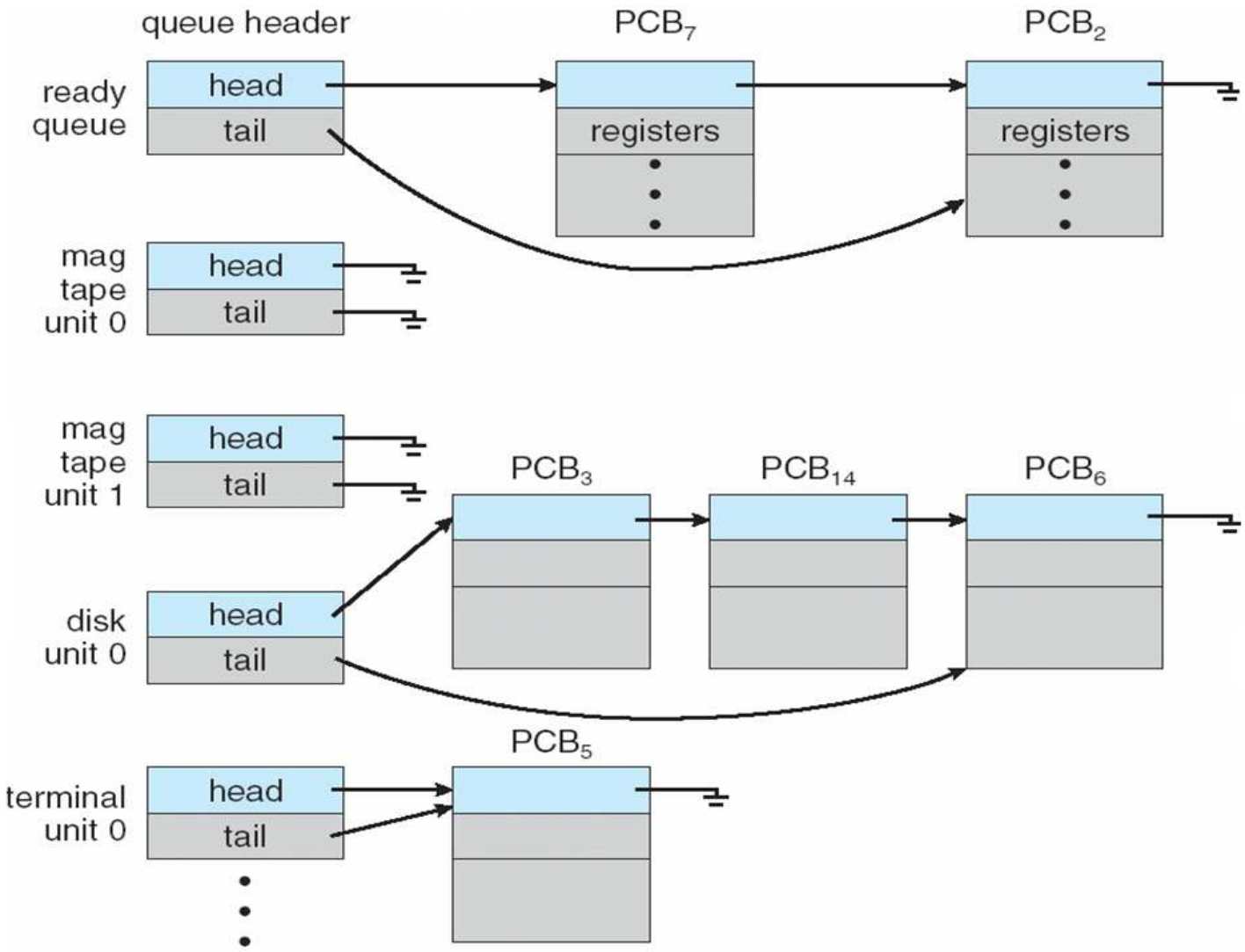
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



行程排班(Process Scheduling)

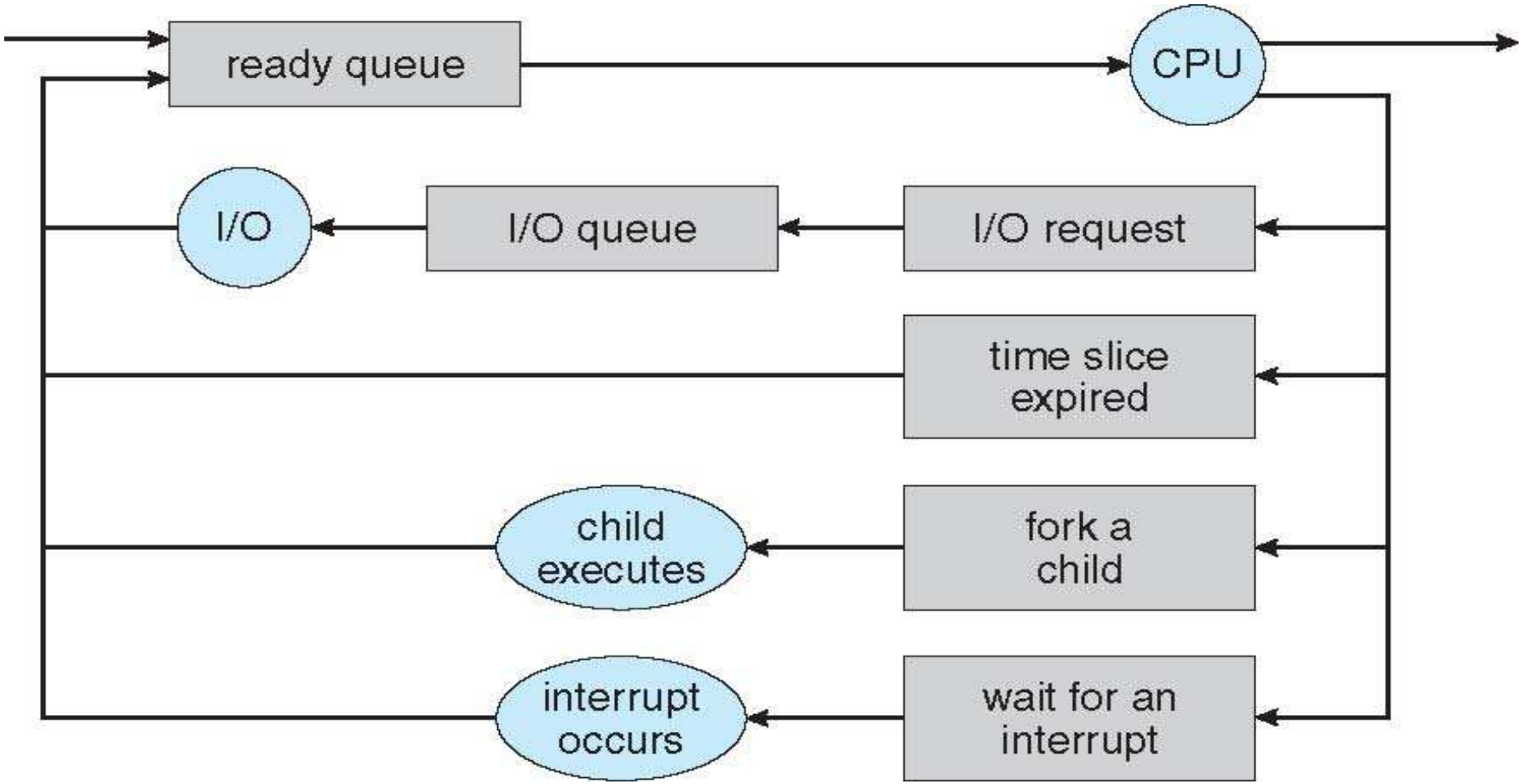
- 多元程式規劃 (Multiprogramming) 的目的是讓一些行程一直在執行，以讓 CPU 使用率達到極大化。
- 分時 (Time-Sharing) 的目的是經常切換行程，讓使用者能程式互動。
- 行程排班程式(Process scheduler) 從可執行行程中選擇下一個CPU要執行的行程
- 作業系統使用排班佇列(scheduling queues)來協助進行排程
  - 工作佇列(Job queue) –當一個行程被允許進入系統時，會被放置在工作佇列中，工作佇列記錄著系統中所有的行程。
  - 就緒佇列(Ready queue) –當一個行程進入就緒狀態時，會被放入就緒佇列中。
  - 裝置佇列(Device queues) –等待I/O完成所產生的佇列，每個裝置都有各自對應的裝置佇列
  - 行程會在不同的佇列間遷移

就緒佇列及I/O裝置佇列



行程排程之佇列圖

- 佇列圖(Queuing diagram)表示佇列、資源及流程

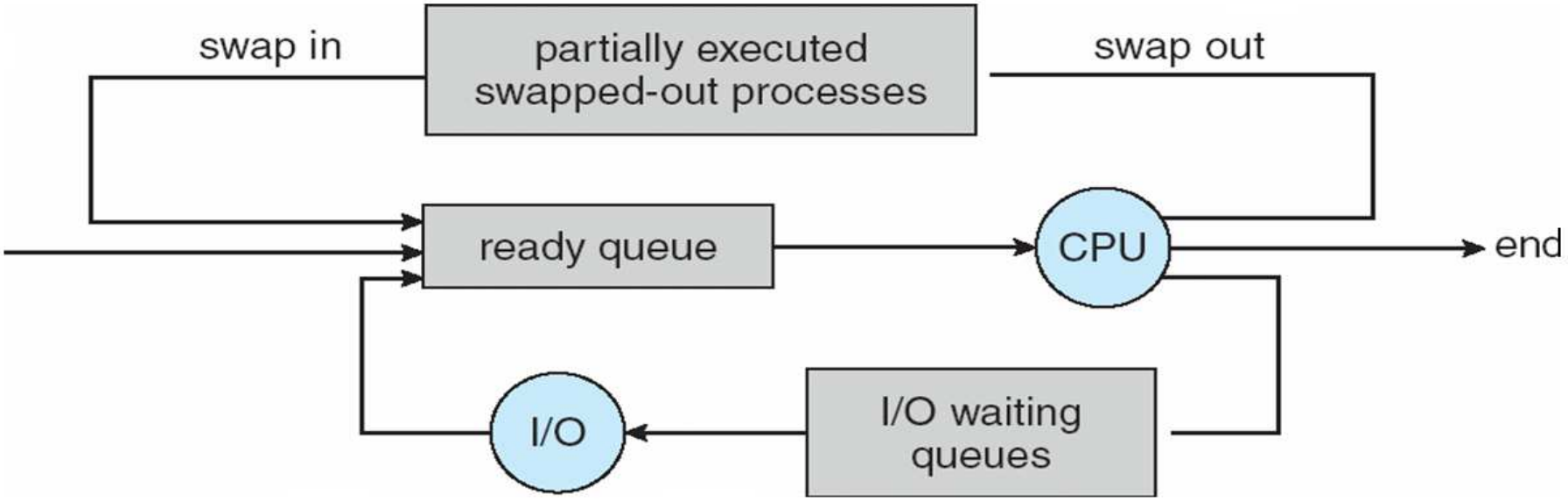


排班程式(Schedulers)

- OS排班程式可分成短程排班、中程排班及長程排班。
- 長程排班程式(Long-term scheduler,或稱job scheduler)
  - 從行程池中選出行程並將它們載入記憶體放入就緒佇列內，以便執行。
  - 執行頻率比較低，處理速度可以較慢。
  - 控制多元程式規劃的程度(Degree of Multiprogramming，代表被載入記憶體的行程總數量)。
  - 對I/O 型行程(I/O-bound process)和 CPU 型行程(CPU-bound process)之間作行程組合與調配(process mix)
    - I/O 型行程(I/O-bound process) –大部份的時間在做I/O，只有少部份的時間在做計算。
    - CPU 型行程(CPU-bound process) –大部份的時間在做計算，只有少部份的時間在做I/O。

中程排班(Medium-Term Scheduling)

- 若多元程式規劃的程度太高時，中程排程會選擇一些行程自記憶體中置換出去(swap out)到磁碟中，以便降低多程式程度。
- 若多元程式規劃的程度很低時，中程排程器會把原被置換出去的行程再把它們替換進來(swap in)到記憶體，並讓它們繼續執行。



短程排班(Short-Term Scheduling)

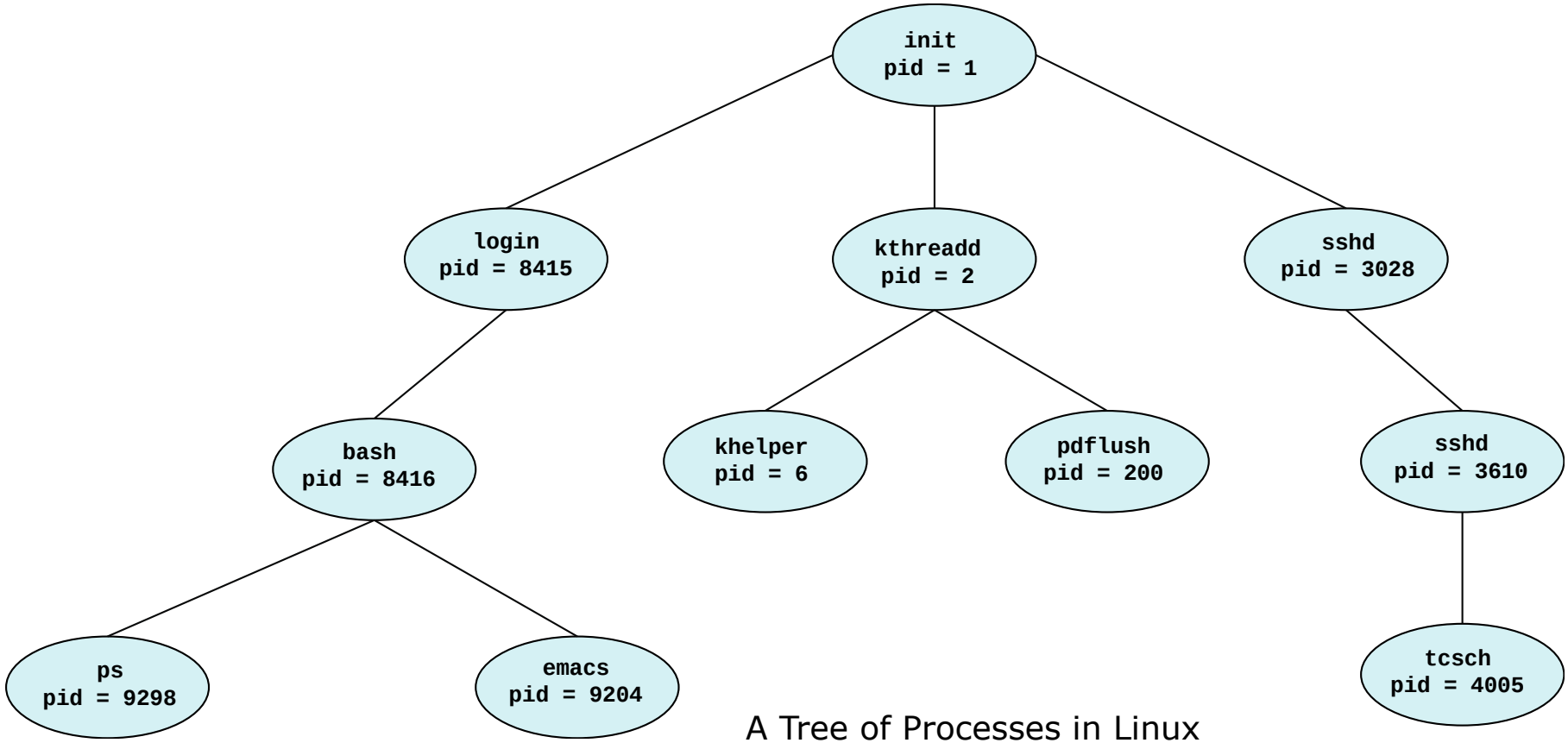
- 短程排程程式（Short-term scheduler,或稱CPU scheduler)是從就緒佇列（Ready Queue）中選一個行程將 CPU 分配給它執行。
- 短程排程程式的執行頻率比較高
- 短程排程程式必須要很快

全文切換(Context Switch)

- 當CPU的使用權由一個行程切換另一個行程時，須將舊行程的狀態存起來並且把另一個行程之儲存狀態載回，以便將執行環境復原為後者當初被中斷時的狀態，此過程稱為全文切換或內容轉換(Context Switch)。
- 一個行程之全文 (Context) 是放在 PCB 中，包含 CPU 暫存器的值、執行狀態、及記憶體管理資訊等資訊。
- 全文切換對系統來說是一種額外負擔 (Overhead)，所以應該盡可能快速完成。
  - OS 和PCB越複雜 ->全文切換的時間越長
  - 全文切換時間的長短與硬體的支援有關
    - 如有些硬體為每一個CPU提供多組暫存器，同時可以載入多組全文，速度較快。

行程的建立(Process Creation)

- 父行程(parent process)產生子行程(child process)，子行程可以再產生其它行程，形成一個行程樹(process tree)。
- 行程經由行程識別代碼(process identifier, pid)辨識與管理



A Tree of Processes in Linux

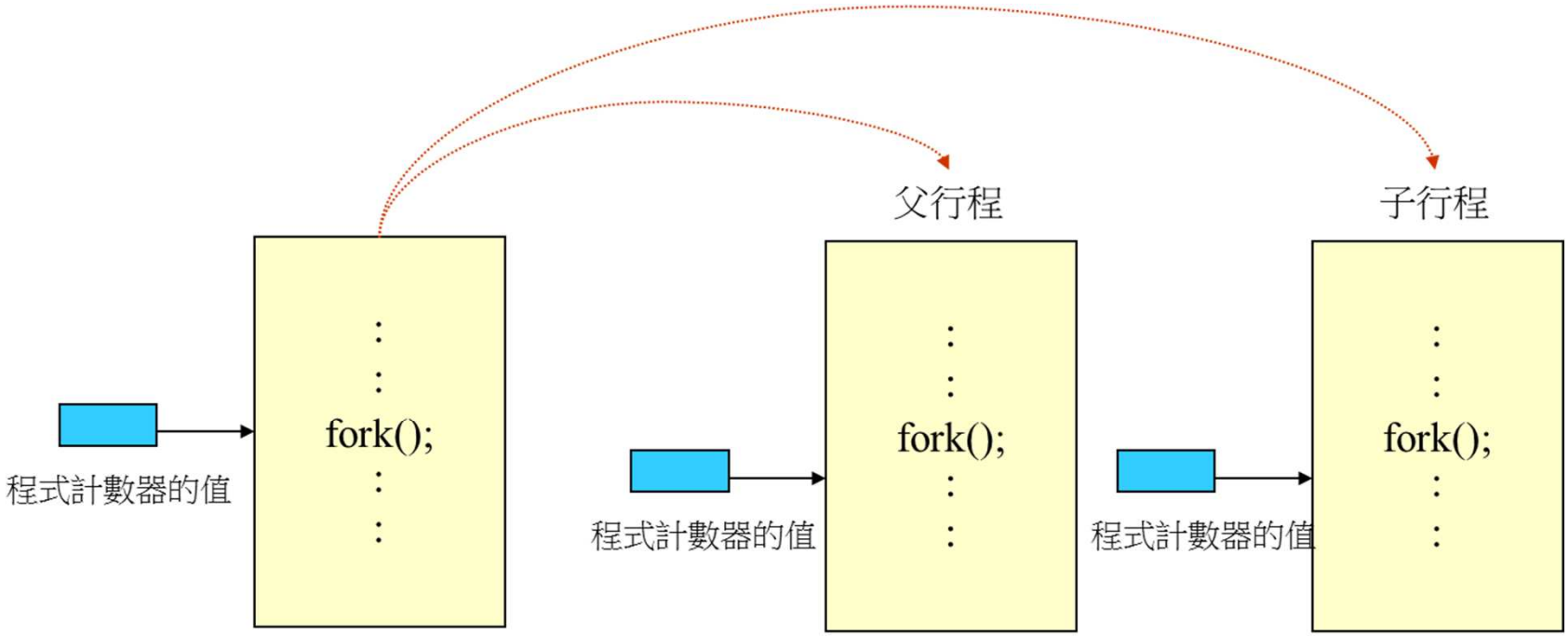
行程的建立(Process Creation)

- 當一個行程建立一個新行程時
  - 資源共用方面(resource sharing options)的幾種可能性
    - 父行程與子行程共用資源
    - 子行程共用父行程的部分資源
    - 父行程與子行程沒有共用資源
  - 執行方面(execution options)的兩種可能性
    - 父行程與子行程併行式地 (Concurrently) 一起執行。
    - 父行程等待某些或全部子行程執行結束。
  - 位址空間(address space)的兩種可能性
    - 子行程複製父行程的位址空間。
    - 子行程載入一新的程式

實作討論—Unix的父行程如何生出子行程

- 利用fork() 系統呼叫產生新的行程
  - 子行程會複製父行程的位址空間，建立一個一模一樣的複本
  - 當fork執行完畢之後，它會分別返回到父行程與子行程中，兩者皆從fork()後的下一指令繼續執行
  - 行程靠fork的傳回值來判斷自己是父行程(傳回值 pid > 0)或子行程(傳回值 pid=0)
  - 在fork()之後使用exec()系統呼叫，讓新程式取代行程的記憶體空間，變身成為完全不同的行程來完成其他的工作。

fork運作原理



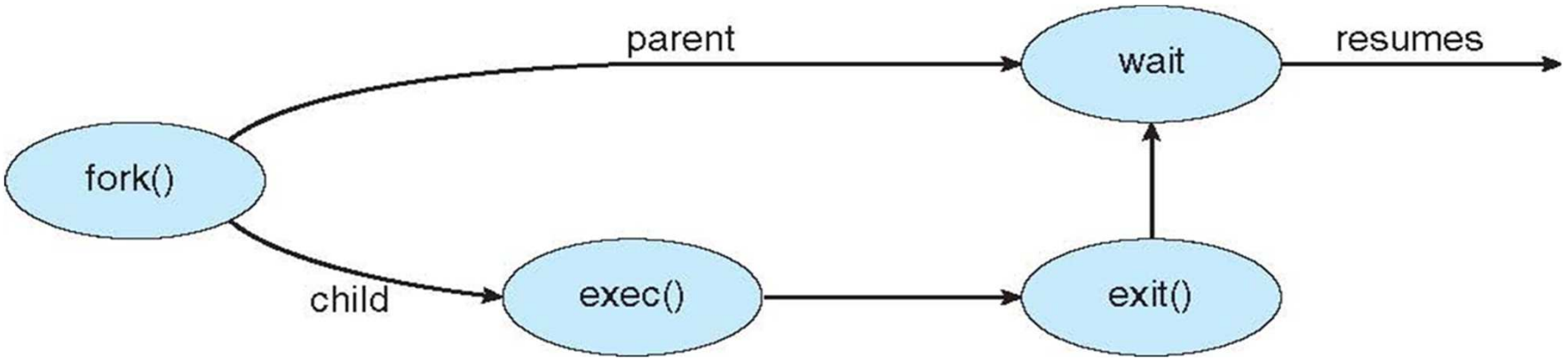
a. 執行fork之前　　b. 執行fork之後

資料來源：作業系統導論，陳宇芬、林慶德

C Program Forking Separate Process

```
#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;
}
```

C Program Forking Separate Process(cont.)



Creating a Separate Process via Windows API

```
#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);
}
```

行程的結束(Process Termination)

- 正常結束方式，行程執行最後一個敘述並透過exit()系統呼叫要求作業系統將行程刪除
  - 父行程可經由wait()系統呼叫取得子行程的返回狀態

    pid t pid; int status;

    pid = wait(&status);
  - 作業系統收回行程的資源(如:記憶體、開啟的檔案、I/O緩衝區等)

行程的結束(Process Termination)

- 其他結束行程方式，父行程可以強迫結束子行程的執行 (如透過(abort())
  - 強迫結束子行程的可能原因：
    - 子行程已經使用超過配置的資源數量
    - 指派給子行程的工作已經不再需要
    - 父行程結束離開
      - 若作業系統不允許子行程在父行程結束之後繼續存在，則會造成所有子行程都跟著結束 稱為串接式終止 (cascading termination)
- 當行程處於終止狀態，代表行程已經不能執行，但是它的一些相關資訊與表格仍然暫時保存在作業系統中。
- 當父行程尚未呼叫wait()而子行程已結束執行，此時子行程會暫時成為殭屍行程(zombie process)
- 當父行程未呼叫wait()且結束執行，子行程尚未結束則會成為孤兒(orphans)

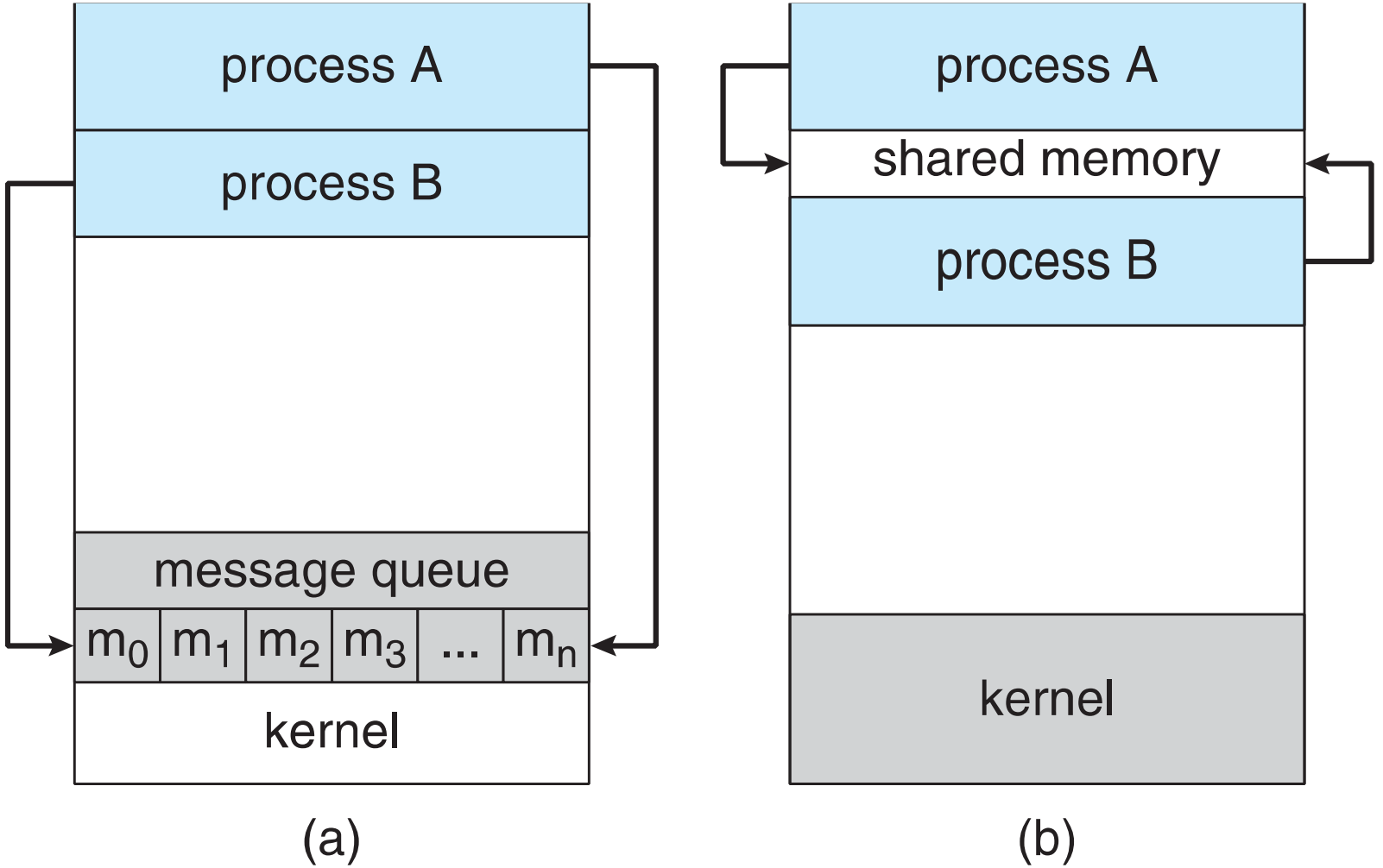
多行程架構—CHROME瀏覽器

- 許多網頁瀏覽器以單一行程的方式執行
  - 如果網站有問題，整個網頁瀏覽器可能停止回應或毀損。
- Google的Chrome網頁瀏覽器是多行程，有三種類型行程
  - 瀏覽器(browser)行程負責管理使用者介面、磁碟、和網路I/O
  - 渲染器(Render)行程呈現網頁，處理HTML、JavaScript、在每開啟一個網站都有一個新的渲染器行程產生
    - 在沙箱(sandbox)中執行，所以限制了磁碟和網路I/O，這將減少了安全漏洞的影響。
  - 插件(Plug-in)行程對每一個型態的插件

行程間通訊(Interprocess Communication IPC)

- 系統內的行程可以是獨立行程(independent process)或合作行程(cooperating process)
- 合作行程能影響其它行程或被其它行程影響，包括共用資料
- 行程合作的原因 ：
  - 資訊共享(Information sharing)
  - 加速運算(Computation speedup)
  - 模組化(Modularity)：可以把系統功能切分成幾個不同的行程或執行緒 (Threads) 來執行。
  - 方便性(Convenience)：一個使用者都可能同時間做幾件工作，行程合作可以提供許多便利。
- 合作行程需透過行程間通訊(IPC)機制進行合作
  - 共用記憶體(Shared memory)
  - 訊息傳遞(Message passing)

通信模式(Communications Models)



(a)訊息傳遞(b)共用記憶體

共享記憶體

- 由作業系統提供某種機制，讓不同行程可以同時存取到某塊記憶體
  - 比共用檔案的傳遞速度快，但長度比較受到限制
  - 執行緒本身透過一些全域變數來進行資訊交換，就內含了共享記憶體的概念

合作行程: 生產者—消費者問題

- 合作行程的典型的例子是生產者—消費者問題 (Producer—Consumer Problem)。生產者行程產生資訊，消費者行程消耗掉資訊。兩種做法:
  - 無界限 (Unbounded Buffer) 緩衝器之生產者—消費者問題
    - 緩衝器的大小沒有限制
    - 生產者可一直送出新資訊，但消費者可能須等待新資訊的到達。
  - 有界限緩衝器 (Bounded Buffer)
    - 有一個固定大小的緩衝器
    - 當緩衝區滿時，生產者必須等待；在緩衝區空時，消費者則須等待。

共用記憶體-有限緩衝區

- 使用一環形陣列

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; /* next free position */
int out = 0;/* the first full
   position*/
```

- 只能放置BUFFER_SIZE-1項資料

生產者演算法

```
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;
}
```

消費者演算法

```
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 */
}
```

訊息傳遞(Message Passing)

- 訊息傳遞 (Message Passing) 是行程間通訊(IPC)與同步之機制
- IPC 的通訊連結是一種邏輯層面的機制。
- 訊息傳遞系統通常會提供2個基本運算：
  - send(message) – 訊息大小固定或可變
  - receive(message)
- 若行程P和Q要溝通，則它們必須:
  - 先建立一個通訊連結 (Communication Link)
  - 藉由 send/receive 交換訊息
- 通訊鏈結的製作
  - 實體 (共用記憶體，硬體匯流排)
  - 邏輯 (直接或間接、同步或非同步、自動或指定緩衝區)

訊息傳遞實作需考慮之問題

- 如何建立通訊鏈結?
- 一個通訊鏈結可以被兩個以上的行程使用嗎?
- 每一組通訊的行程間可以有多少通訊鏈結存在?
- 一個通訊鏈結的容量?
- 通訊鏈結可以容納的訊息大小是固定還是可變的?
- 一個通訊鏈結的傳輸方向是單向還是雙向?

訊息傳遞-直接通訊(Direct Communication)

- 溝通雙方必須要知道對方的身分，
  - send (P, message) – 傳送訊息給行程 P
  - receive(Q, message) – 從行程 Q接收訊息
- 在這種通訊方式中，通訊鏈結的性質有
  - 通訊鏈結自動被建立
  - 一個通訊鏈結只有2個行程參與
  - 2個行程間僅存在一個通訊鏈結
  - 可以是單向或雙向
- 最大缺點是模組化 (Modularity) 程度受限，因為，若某一個行程更改名稱，哪麼其它跟它通訊的行程也要更改名稱。

訊息傳遞-間接通訊(Indirect Communication)

- 間接通訊是指訊息藉由信箱(mailbox)(也叫作埠，port)來傳送與接收訊息，行程間可以透過幾個郵件箱來通訊。郵件箱通訊的基本命令如下：
  - 建立一個信箱
  - send(A, message) –將訊息(message)傳送至信箱A
  - receive(A, message) –從信箱A接收一個訊息(message)
  - 刪除一個信箱
- 通訊鏈結具有下列的性質:
  - 只有在共用信箱的行程間會建立通訊鏈結
  - 一個通訊鏈結可以被多個行程共用
  - 每一組行程可以共用數個通訊鏈結
  - 可以是單向或雙向

間接通訊(2)

- 信箱共用問題
  - P1, P2,和P3 共用信箱A
  - P1, 傳送; P2 and P3 接收
  - 誰獲得訊息?
- 解決方案
  - 一個鏈最多只能兩行程使用
  - 一次只能一個行程執行receive()
  - 由系統能任意選取接收訊息的行程，傳送者會被通知接收者是誰

行程間通訊之同步(Synchronization)

- 不論是直接或間接形式的溝通，都會面臨收送雙方不同步的情況。
- 訊息傳遞可以是等待式(blocking)或非等待式(non-blocking)
- 等待式(blocking)被視為同步式(synchronous)
  - 等待式傳送(blocking send) ：傳送行程持續等待，直到訊息被接收為止。
  - 等待式接收(blocking receive) ：接收行程持續等待，直到接收到訊息為止。
- 非等待式(non-blocking)被視為非同步式(asynchronous)
  - 非等待傳送(non-blocking send)：傳送端行程送出訊息之後，不必等訊息被接收，就可以繼續執行後面的動作
  - 非等待接收(non-blocking receive)：接收端行程在嘗試讀取訊息之後，不論是否有收到訊息，都可以繼續執行後續的動作
- 當傳送端與接收端都是使用等待式通訊時，兩者之間就會會合(rendezvous)而同步

訊息佇列之緩衝處理(Buffering)

- 一個通訊鏈結的訊息佇列可有三種緩衝處理方式
  - 零容量(Zero capacity)緩衝區– 傳送端必須等候接收端接收完畢(會合)
  - 有限容量(Bounded capacity)緩衝區–如果鏈已經填滿，那麼傳送端必須等待，否則不需等待。
  - 無限容量(Unbounded capacity)緩衝區–傳送端完全不需要等待

Examples of IPC Systems - POSIX

- POSIX 共用記憶體
  - 行程首先建立共用記憶體物件，或開啟現存的共用記憶體物件
    shm_fd = shm_open(name, O CREAT | O RDRW, 0666);
  - 設定物件的大小

    ftruncate(shm fd, 4096);
  - 對共用記憶體物件進行記憶體映射(memory map)

    ptr = mmap(0, 4096, PRO_WRITE, MAP_SHARED, shm_fd, 0);
  - 行程寫入共用記憶體

    sprintf(ptr, "Writing to shared memory");
  - 調整指標指向接下來寫入位址

    ptr += strlen("Writing to shared memory");

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

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

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_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;
}
```

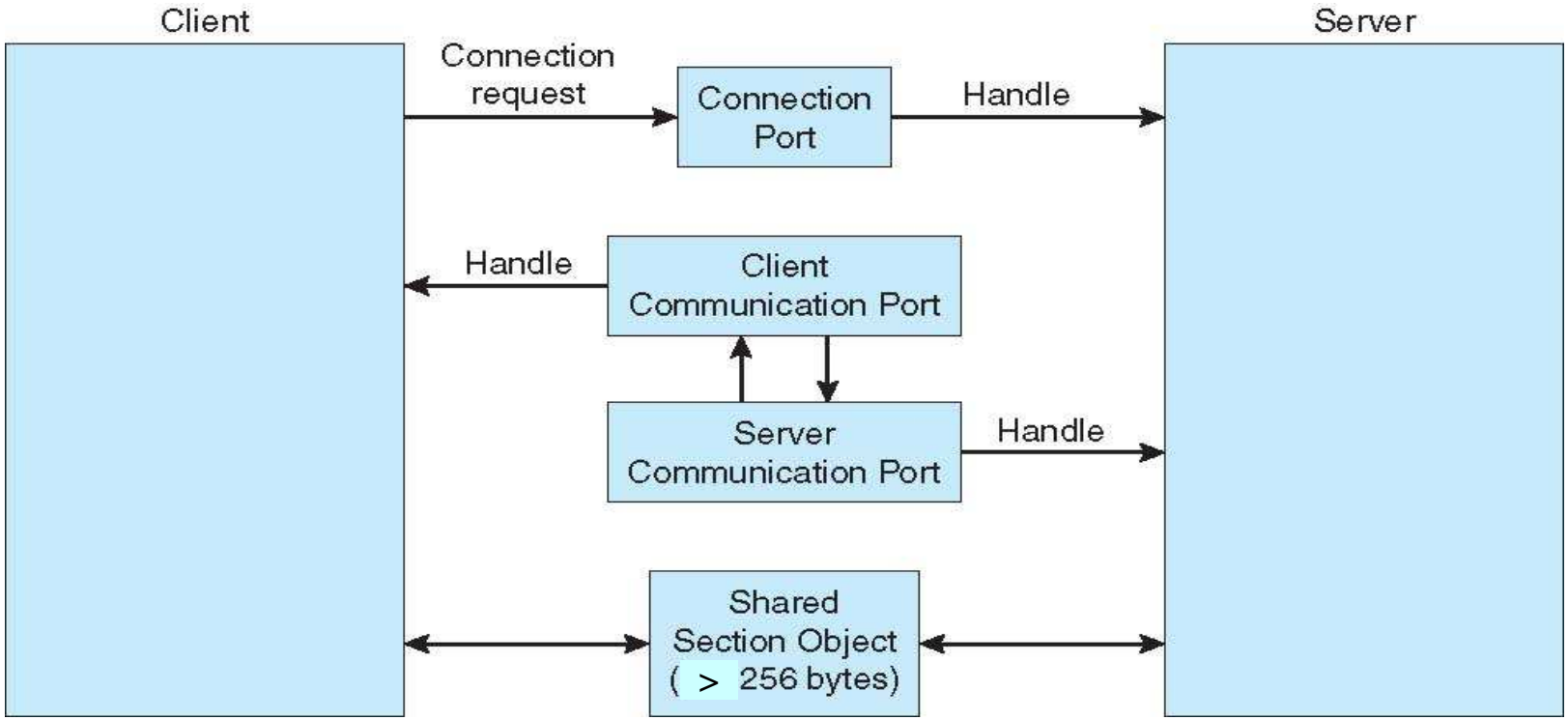
IPC POSIX Consumer

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

    return 0;
}
```

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - Server processes publish connection-port objects.
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
      - One for client-server messages, the other for server-client messages
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.
- When the RPC is being invoked on a process on the same system, the RPC is handled indirectly through an ALPC procedure call.

Local Procedure Calls in Windows XP

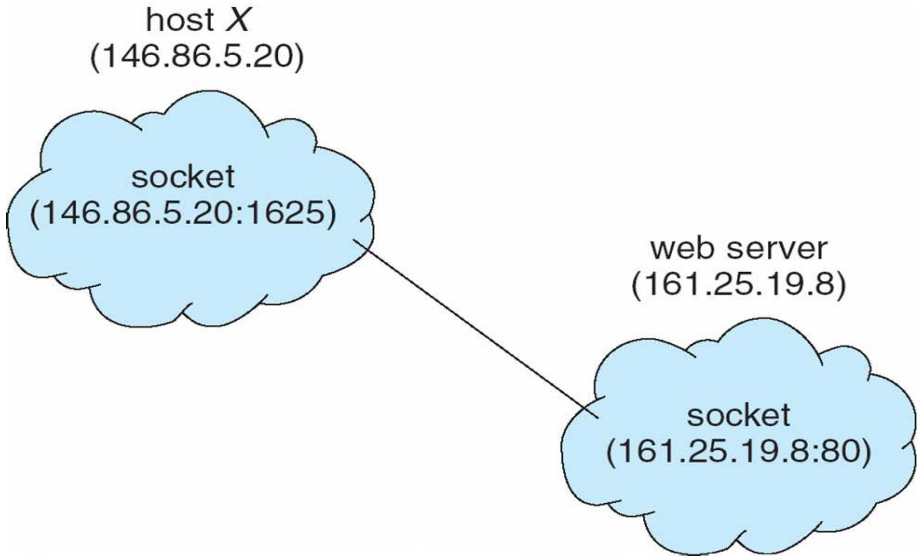


客戶-伺服器的通信(Communications in Client-Server Systems)

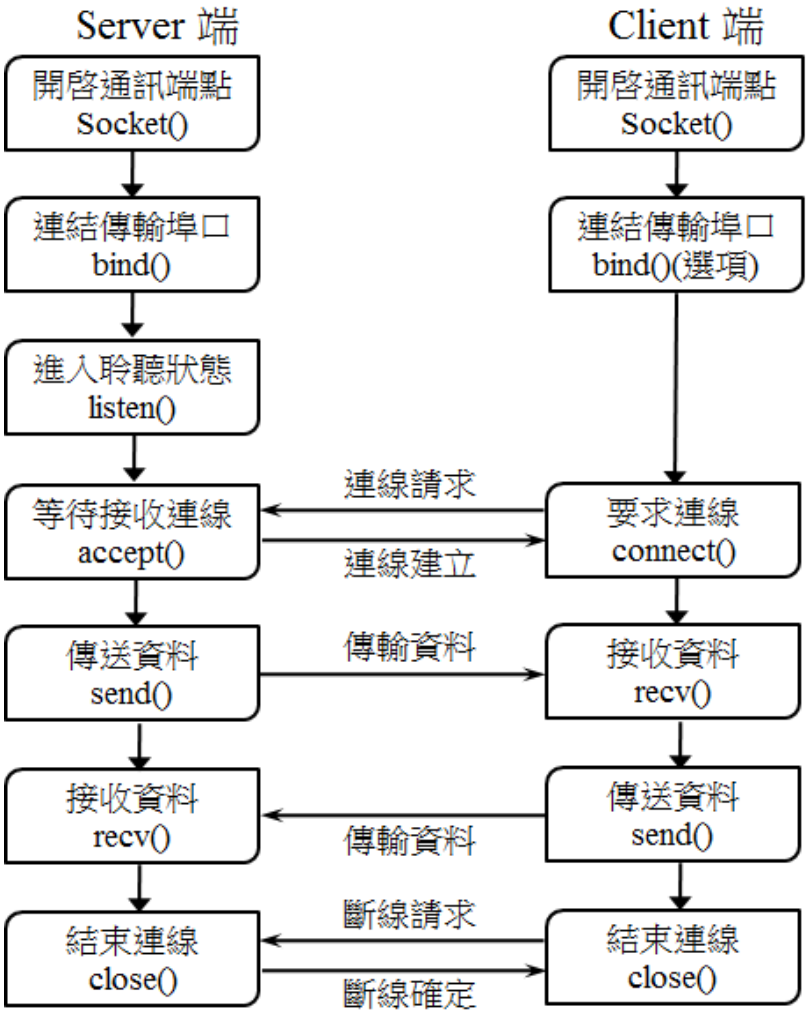
- 插座(Sockets)
- 遠程程序呼叫(Remote Procedure Calls)
- 管道(Pipes)
- 遠端方法呼喚(Remote Method Invocation, Java)

插座(Sockets)

- 插座(socket) 代表通訊的端點
- 插座是由一個IP位址和一個埠號碼(port number)所組成
- 插座161.25.19.8:1625 是指在主機host 161.25.19.8上的埠 1625
- 通信包含一組插座
- 1024以下的埠只能用來製作標準服務
- IP位址127.0.0.1是內部迴圈網路(loopback)，它連到行程自己執行的系統



Socket 連線流程


Server 端
Client 端
開啓通訊端點
Socket()
開啓通訊端點
Socket()
連結傳輸埠口
bind()
連結傳輸埠口
bind()(選項)
進入聆聽狀態
listen()
連線請求
等待接收連線
accept()
要求連線
connect()
連線建立
傳送資料
send()
傳輸資料
接收資料
recv()
接收資料
recv()
傳輸資料
傳送資料
send()
斷線請求
結束連線
close()
結束連線
close()
斷線確定

Sockets in Java

- 三種型態的插座
  - 連接傾向(TCP)
  - 無連接傾向(UDP)
  - MulticastSocket 類別–資料可被送到許多接收者

```
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);
      }
   }
}
```

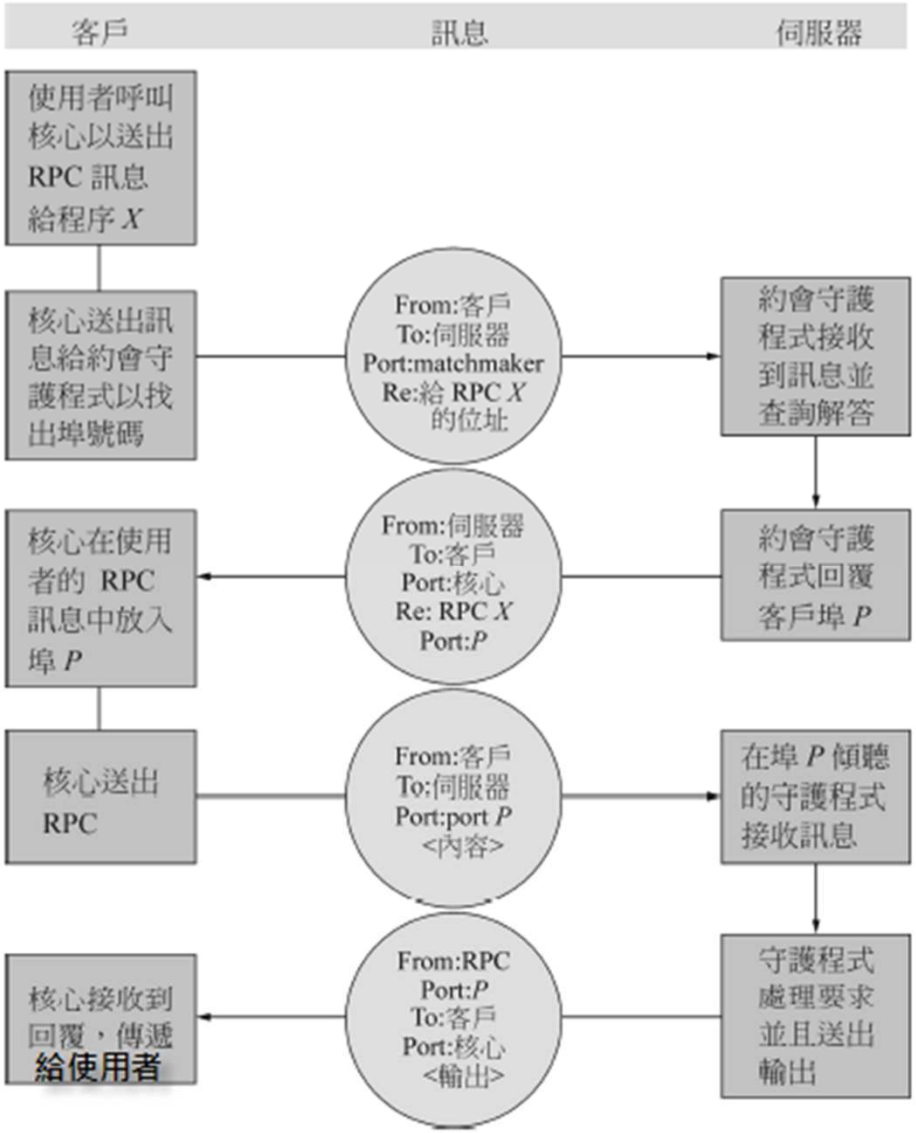
遠程程序呼叫(Remote Procedure Calls)

- 遠方程序呼叫 (RPC) 則將網路之系統連結抽象化 (Abstract) 成程序呼叫形式。相對於 IPC，RPC 所交換的訊息是結構化形式，而不是封包形式的資料。
- **Stubs(插樁程序)** – client-side proxy for the actual procedure on the server
  - 用來替換一部分功能的程序段。插樁程序可以用來模擬已有程序的行為（比如一個遠端機器的過程）或是對將要開發的代碼的一種臨時替代。
- 客戶端stub會找到伺服器端之埠口，並重新匯整(marshalls)參數
- 伺服器端的stub接收訊息, 解開被匯整的參數，並執行伺服器端的程序

遠程程序呼叫 (2)

- 在 Windows, stub程式碼是以Microsoft Interface Definition Language (MIDL)規格寫的程式碼編譯而成
- 資料表示是經由External Data Representation (XDL)處理須考慮到不同的架構
  - Big-endian和little-endian
- 遠端通訊比區域通訊有更多失敗的情況
  - 訊息傳送正好只有一次，而非至多一次
- 本地程序呼叫基本上不太容易出錯，但是遠方程序呼叫卻會因為網路之服務品質不佳而出錯。RPC 機制應該增加可靠度 (Reliability) 等機制
- 客戶端與伺服器端之埠口的繫結 (Binding)通常可以藉由一個會合機制 (rendezvous (or matchmaker))來達成，系統在一個固定的 RPC 埠口提供一個會合服務程式 (Rendezvous Daemon)。

RPC的執行


客戶
訊息
伺服器
使用者呼叫核心以送出RPC 訊息給程序 X
核心送出訊息給約會守護程式以找出埠號碼
From:客戶 To:伺服器 Port:matchmaker Re:給 RPC X 的位址
約會守護程式接收到訊息並查詢解答
核心在使用者的 RPC 訊息中放入埠 P
From:伺服器 To:客戶 Port:核心 Re: RPC X Port:P
約會守護程式回覆客戶埠 P
核心送出 RPC
From:客戶 To:伺服器 Port:port P <內容>
在埠 P 傾聽的守護程式接收訊息
核心接收到回覆，傳遞給使用者
From:RPC Port:P To:客戶 Port:核心 <輸出>
守護程式處理要求並且送出輸出

遠方方法呼叫(Remote Method Invocation，RMI)

- 遠方方法呼叫(Remote Method Invocation，RMI)是一個 Java 的遠方方法呼叫機制。 RMI 允許一個執行緒 (Thread) 呼叫遠方之物件 (Object) 的方法 (Method)。
- RMI 和 RPC 的差別
  - RPC 只能以程序或函數方式呼叫； RMI 支援遠方物件之方法的呼叫。
  - RPC 之參數是一般資料結構；而 RMI 是將參數以物件 (Object) 形式傳給遠方物件。