行程

資科系
林偉川

行程概念

- 行程與程式主要的不同點：
  - 程式是被放在外部的儲存裝置如磁碟上，而行程則被放在記憶體中。
  - 程式在儲存裝置中是靜態的，而行程在記憶體中是動態的，它會隨著一些事件的發生而產生相對的改變。
- 行程，就是一個執行中的程式。
- 對只有一顆CPU的系統 而言，同一個時間只會有一個行程真正被執行。該由哪個行程執行？行程間要如何交換資料才能有效率地利用系統資源？➔行程管理

2

行程簡介

- 一個行程包括了
  - 相對應的程式碼
  - 程式計數器(PC)➔記錄下一個要被執行的指令
  - CPU 中各暫存器的值
  - 行程堆疊➔暫時儲存子程式的參數、返回位址
  - 資料區段➔紀錄全域變數
- 一個程式很可能被重複執行多次而產生多個行程，但是這些行程是彼此獨立的

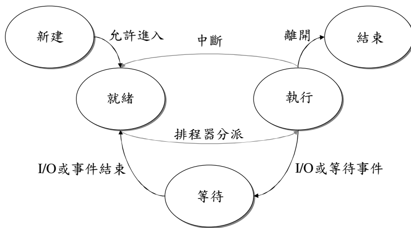
3

行程的狀態

- 一個行程在執行過程中，會改變很多狀態，狀態為目前行程所處的執行情況。
- 一個行程的狀態通常有下列幾種：
  - 新建➔行程剛被產生的狀態
  - 執行➔行程中指令正被執行的狀態，佔有CPU資源
  - 等待➔行程目前正在等待事件發生的狀態
  - 就緒➔行程準備好被執行的狀態
  - 終結➔行程已執行結束的狀態
- 單CPU中同一時間內，只會存在一個狀態為執行的行程，在同一時間內可以存在很多個狀態為等待或就緒的行程

4

行程狀態圖(91tku、93ncu、96,92nttu、97tku)



5

行程控制區塊

- 行程控制區塊(PCB)，儲存行程在執行時相關的資訊(開哪些檔、行程所處狀態)。
- PCB 中通常包括了(94tku)
  - 行程狀態，如上所述
  - CPU 暫存器➔行程在執行時發生中斷，作業系統需要將暫存器的值儲存起來，當返回時行程才能正確地繼續執行
  - 排程資訊➔包含行程的優先權和其他排程的相關資訊
  - I/O 狀態➔包含行程在執行時使用到I/O裝置、開啟檔案

6

行程控制區塊

- CPU暫存器種類，分別為一般用途、算術、索引、堆疊、程式計數器及狀況暫存器
- 在80386 CPU以下的規劃節段或區段暫存器分別有CS、SS、DS、ES四個16位元的暫存器，386以後加入FS、GS兩個，CS就是規劃來指定給指令碼存放的區段，所以稱為指令段(code segment)暫存器，而SS為堆疊段(stack segment)暫存器，DS、ES分別為資料段(data segment) 及額外段(extra segment)暫存器，FS、GS分別為旗號段(flag segment) 及總體段(global segment)暫存器

7

行程控制區塊

- 一般用途暫存器:AX(算術)、BX、CX、DX
  AX:累積暫存器，BX:基底暫存器，CX:計數暫存器，DX:資料暫存器
- 索引暫存器:SI、DI
  SI:來源索引暫存器，DI:目的索引暫存器
- 堆疊、基底暫存器:SP、BP
  SP:堆疊指標暫存器，BP:基底指標暫存器
- 指位/指標暫存器(指位器):IP
- 程式計數器稱為LC或稱IC
  作為存放下一個電腦要執行指令位址

8

行程控制區塊

- 旗標暫存器:FLAG

  OF DF IF TF SF ZF AF PF CF

  AF：輔助進位旗標

  CF：進位旗標

  OF：溢位旗標

  SF：符號(負號)旗標

  PF：奇偶旗標

  ZF：零值旗標

  DF：方向旗標

  IF：中斷旗標

  TF：單步旗標

9

行程控制區塊

- 行程其他資訊包括所使用CPU時間、記憶體大小、行程代號等都可放於PCB中
- 當行程進行切換時，需要將目前行程的相關資訊記錄在該行程的 PCB 中，並將另一個行程的 PCB 載入至系統中，這個動作稱為內文切換(context switch)。(96nttu)
- 當 CPU 的使用權由一個行程轉到另一個行程時需將上一個行程的PCB儲存起來，並把要使用CPU的行程之PCB載入系統，稱為內文切換。

10

內文切換

- 內文切換動作所花的時間對系統而言是額外的負擔 ，因為此過程所做的事並不是真正有生產力的工作，此時間是由要儲存CPU暫存器個數，記憶體的速度、CPU是否提供特別的指令來決定。
- 執行緒降低內文切換所花的時間。

11

行程控制區塊(PCB)

- 行程指的是正在執行的程式。行程不只是程式碼 (有時也稱為本文區，text section) ，它還包含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。
- 行程還包括存放暫用資料(譬如:副程式的參數、返回位址，以及暫時性變數)的行程堆疊(stack)，以及包含整體變數的資料區間(data section)。行程也包含堆積 (heap)，所謂堆積就是在行程執行期間動態配置的記憶體。

12

行程控制區塊(PCB)

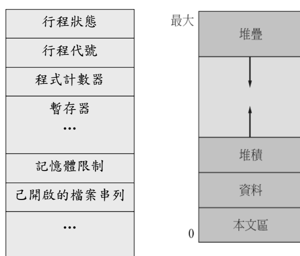
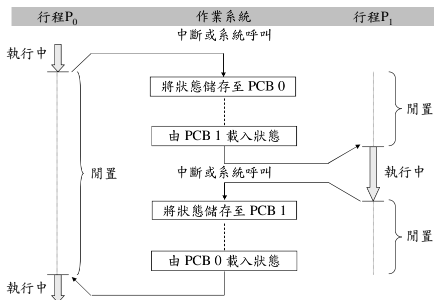


圖 3.1 行程記憶體結構

13

行程的切換



14

行程排程

- 為了增加 CPU 的使用效率而提出多個行程的觀念。
- 一個單 CPU 的系統來說，隨時只能有一個行程在執行。
- 其他行程則必須等待 CPU 空閒下來，然後再經由排程器選出，才能取得 CPU 的使用權。
- 如何排程是影響作業系統效能最重要的因素。

15

考題

- List the information stored in PCB. Describe the action taken by a kernel to context switching.
- Draw the process state transition diagram. Explain why a context switch may happen.

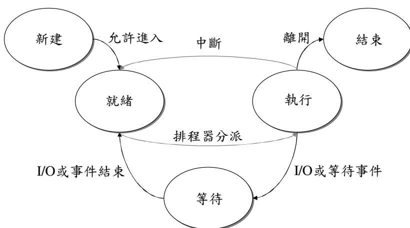
16

排程佇列

- 一個行程在執行期間會在各種不同的佇列中進出。
- 一個系統中通常有
  - 工作佇列➔紀錄系統中所有的行程
  - 就緒佇列➔一個行程(PCB)準備要執行時，會被移至此，形成一個PCB串列，有時還加上雜湊表來加快PCB的搜尋
  - 等待佇列➔行程在執行期間要等待某些事件完成才能執行時，此行程就會被移至此
  - 裝置佇列➔為了等待I/O裝置動作完成而產生的佇列，每個裝置都有自己的裝置佇列

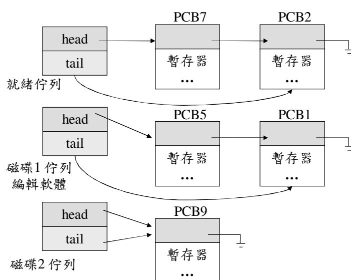
17

行程狀態圖



18

就緒佇列與裝置佇列



排程器

- 一個行程從開始執行到最後離開系統，會在前述的各種佇列進出，作業系統必須針對不同目的在各種佇列中採取不同的選取行程的方法，為了實現不同的選取方法，必須使用不同的排程器
- 作業系統中主要的排程器有：
  - 長程排程器(Job Scheduler)➔從磁碟中選取合適的行程，將行程放置於記憶體中準備執行
  - 短程排程器(CPU Scheduler)➔在就緒佇列中選出一個行程，然後將CPU的使用權交給該行程

排程器

一個新的行程最初是置於就緒佇列。它就一直在就緒佇列中等待，直到選來執行或被分派。一旦這個行程配置CPU並且進行執行，則會有若干事件之一可能發生:

・行程可發出I/O要求，然後置於一個I/O佇列中

・行程可產生出一個新的子行程並等待後者的結束。(fork指令)

・行程可強行地移離CPU(如用中斷的結果一樣)，然後放回就緒佇列中。

21

排程器

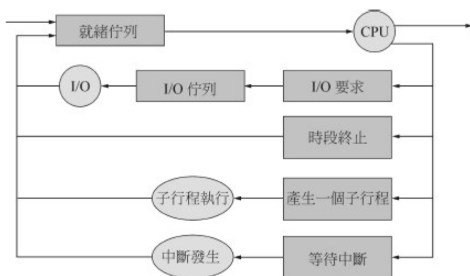


圖 3.6 行程排班的佇列圖表示

排程器

- 長程排程器和短程排程器最大的不同點
  - 執行的頻率
- 一個行程在執行幾毫秒後就會產生I/O中斷，這時短程排程器就必須執行，找出下一個可以取得CPU資源的行程
- 若短程排程器每90毫秒會被執行一次，每次花10毫秒才選出下一個行程，如此約有10%CPU資源(10/(10+90))沒有真正用來執行使用者工作而浪費➔排程工作

23

排程器

- 長程排程器就可能幾分鐘才執行一次，有足夠時間選出一組對系統最好的行程，以系統資源使用程度來看，可將行程分成以I/O為主及CPU為主
- 長程排程器控制整個系統多工的程度➔同時有幾個行程被放於記憶體
- I/O為主的行程是指執行期間花在I/O上的時間比花在計算上的時間長。 CPU為主的行程則相反。如果系統如果都存在某一類型的行程，其效能都比較差

24

排程器

- 如果系統都是I/O為主的行程，則大部分的行程都停留在I/O的等待佇列等待I/O完成，而就緒佇列卻經常是空的，造成CPU的利用率很低
- 如果系統都是CPU為主的行程， I/O的等待佇列大部分是空的，所以I/O裝置閒置不用

25

排程器

- 這兩種狀況都造成系統資源分配不平均，並降低整體效能，如何挑出兩者比例合適的工作，就是長程排程器最重要的課題，但是很難，因為行程未執行前，系統很難預測它到底是屬於哪一類型的行程，若對系統所有行程的執行行為做分析而進行長程排程，這對系統整體效能會有相當正面的影響

26

排程器

- 一般作業系統已無長程排程器，分時系統上至少每個行程有一部分放在記憶體中，然後用短程排程器來排程，這樣的系統受限於記憶體大小及終端機數目上限等，使用者可憑感覺來推判效能的高低，當系統反應時間超出使用者可容忍範圍，使用者可選擇離開系統或是移除一些不必要的行程

27

排程器

- 中程排程器最主要的用途在於降低系統多工的程度，以增加系統可用記憶體的大小。中程排程器將依些行程從記憶體暫時移除，在一段時間後某些行程會再度被載回記憶體(就緒佇列)中準備執行，這種動作稱為置換(swapping in/out)

28

行程的建立與結束

- 不同行程在系統中可以同時執行，而且必須能動態地被建立與刪除，如此才能有效地運用或共享系統資源來完成系統的目標。
- 如何有效地建立和刪除行程也是影響作業系統效能的重要因素。

29

行程的建立

- 一個行程能在執行的期間透過系統呼叫建立很多新的行程。
- 建立新行程的行程稱為父行程，而新建立的行程稱為子行程，可將行程間的關係看為行程樹。
- 父行程建立了子行程後，子行程可以直接取得父行程的資源來使用，也可能只取用其中一部份，就父行程而言，需將部分資源(記憶體或檔案)與子行程分享

30

行程的建立

- UNIX 系統中，使用行程代號(PID)來分辨不同的行程，PID是獨一無二的行程身分證明。
- 系統呼叫
  - fork()：新行程會複製父行程的位址空間，子行程中fork()傳回值為0，父行程的回傳值則為子行程的PID
  - execlp()：將程式的執行檔載入記憶體，並用程式的內容覆蓋舊程式的記憶體空間，然後開始執行
  - wait()：父行程若沒有其他工作要做，則將自己從準備佇列移到等待佇列，等待下一次被排程器選出來執行或是子行程執行結束。

31

行程的建立

```
if ((pid = fork()) == 0) {
    // child process
    execlp(“execfile”);
}
else {
   // father process
}
```

32

行程的產生

- 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。原先的行程就叫做父行程，而新的行程則叫做子行程。每一個新產生的行程可以再產生其它的行程，這可以形成一幅行程樹。

33

行程的產生

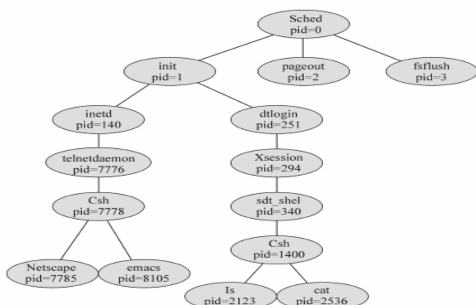
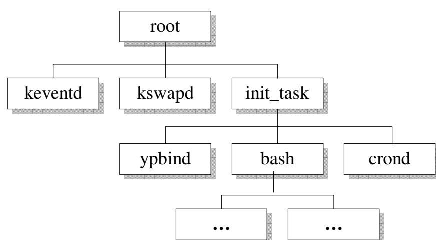


圖 3.8 典型的 Solaris 系統的行程樹

行程樹



35

行程的結束

- 一個行程在結束最後一個指令會利用exit()請求作業系統將自己從系統中移除，執行期間用到的資源如實體記憶體、虛擬記憶體、開啟的檔案和使用的 I/O 裝置等，都會交還給作業系統。
- 除自己正常結束外，行程也可以透過abort()強迫另一個行程結束執行。通常只有父行程才能使用此系統呼叫去中斷子行程

36

行程的結束

- 父行程使用wait()等待子行程結束，而子行程用exit()結束執行，子行程結束後，作業系統則呼叫wait()傳回結束執行的子行程PID，如此父行程就可知道是哪個子行程結束了。如果父行程結束，則它所有的子行程都會被作業系統結束或是交由另一特定行程(init)代管，在UNIX系統中若沒有父行程，作業系統根本不知道要向誰回報子行程的結束狀態
  fork(95tpu、95ncu), wait, join(thread), exit??
  ps、top、kill(97nttu)

37

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() { pid_t pid;
  pid=fork();
 if (pid == 0) execlp("/bin/ls", "ls", NULL);
  else { // father process
    wait(NULL);
     execlp("/bin/date", "date", NULL);
     fprintf(stderr,"Child Complete"); exit(0); }
}
```

38

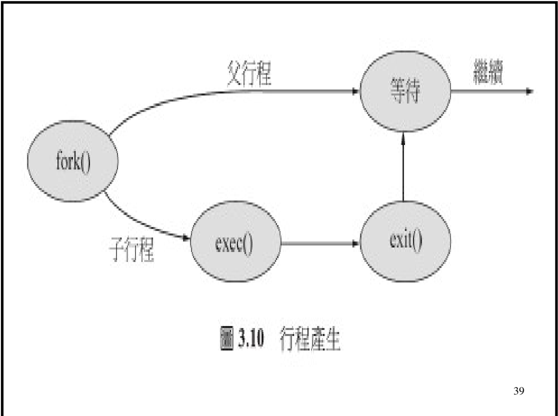


圖 3.10 行程產生

39

```
#include <stdio.h>
#include <windows.h>
int main(void) { // dev-C++
   STARTUPINFO si;  PROCESS_INFORMATION pi;
  ZeroMemory(&si, sizeof(si)); si.cb=sizeof(si);
  ZeroMemory(&pi, sizeof(pi));
  if (!CreateProcess(NULL,"c:\\WINDOWS\\system32\\mspaint.exe",
    NULL,NULL,FALSE,0,NULL,NULL, &si, &pi))
    {    fprintf(stderr,"create fail"); return -1;    }
  if (!CreateProcess(NULL,"c:\\WINDOWS\\system32\\notepad.exe",
    NULL,NULL,FALSE,0,NULL,NULL, &si, &pi))
    {    fprintf(stderr,"create fail"); return -1;    }
  WaitForSingleObject(pi.hProcess, INFINITE);
  printf("Child complete");  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);   }
```

40

執行緒

- 系統呼叫 fork() 的缺點
  - 系統呼叫fork()，會複製父行程的位址空間，需要做大量記憶體的複製
  - 進行內文切換時需付出相當的代價
  - 兩個行程因各自擁有自己的位址空間，無法直接進行溝通(IPC)，需透過資源共享機制如共享記憶體或是訊息傳遞
- 若行程間可以共用一部分的記憶體空間，那麼額外的負擔就能減少，這也就是建立執行緒的基本理由。

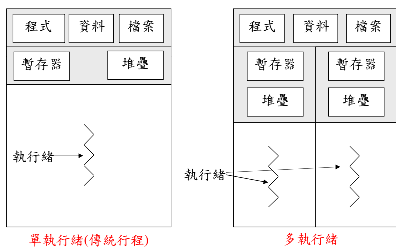
41

執行緒觀念

- 執行緒
  - 輕量級行程(LWP)
  - 使用 CPU 資源的基本單元
  - 包含了一個程式計數器、一組暫存器和一個堆疊空間
  - 與其他的執行緒共用同一個位址空間，共享程式區段、資料區段和從作業系統取得的資源(開啟檔案及信號)
- 傳統的行程
  - 重量級行程(HWP)
  - 可看成是只有一個執行緒在執行的行程

42

傳統行程與執行緒行程



43

執行緒的優點

- 大部份應用程式都是多執行緒的架構
- 使用執行緒來取代傳統行程有幾項優點：
  - 資源共享容易：不需透過行程溝通機制(共享記憶體或是訊息傳遞)就能直接交換資料，節省溝通的額外負擔
  - 節省記憶體空間：系統呼叫 fork()會複製一份與父行程內容相同的位址空間給子行程，造成浪費，執行緒不需要複製所有的位址空間

44

執行緒的優點(94ncu)

- 使用執行緒來取代傳統行程有幾項優點：
  - 快速的內文切換：執行緒只需存一個程式計數器、一組暫存器和一個堆疊空間，而行程還需加上如位址空間的儲存
  - 平行處理：若有多個CPU，執行緒可以提升整個系統的效能，若兩個執行緒沒有相依性，此二執行緒可同時在不同的CPU上執行(格網運算)

45

使用者和核心執行緒(92tku、97tku、96ncu、93tpu、94tpu、94nttu)

- 在作業系統中，有兩種方式來支援執行緒
  - 使用者執行緒
    - 利用執行緒函式庫來提供的，建構在核心之上，但不經由核心直接支援
    - 建立、結束與排程都在使用者空間完成，系統核心並不知道使用者執行緒的存在，如此使得使用者執行緒在建立與管理時比較有效率
    - 若行程中的執行緒因系統呼叫而暫停，則同行程中其他所有執行緒也都會暫停執行
    - 常見的使用者執行緒函式庫包括：POSIX Pthreads(94tpu)、Mach C-threads、Solaris threads

46

使用者和核心執行緒

- 在作業系統中，有兩種方式來支援執行緒
  - 核心執行緒
    - 由作業系統直接支援
    - 建立、結束、排程等都在核心空間完成，需要從使用者空間轉換至核心空間，所以建立與管理執行緒時比使用者執行緒來得慢
    - 由於直接由核心管理，若行程中的執行緒暫停，核心可以安排其他在同行程中的執行緒繼續執行，在多CPU的執行環境下，核心可以安排不同的執行緒在不同的CPU上執行，以充分利用系統資源

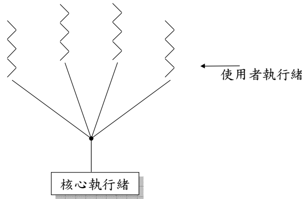
47

多執行緒的模型

- 實作執行緒時通常有三種模型(96tpu)
  - 多對一模型：將許多個使用者執行緒對應到同一個核心執行緒，執行緒在使用者空間執行，所以執行緒的建立和管理速度都很快，如果行程中某執行緒暫停執行，那麼整個行程的所有執行緒都會暫停，同一時間內只有一個執行緒可以存取核心，所以在多個CPU環境下，這類模型的執行緒不能並行地執行。若作業系統不支援核心執行緒但想要實作使用者執行緒，也可以使用多對一的模型。

48

多對一模型



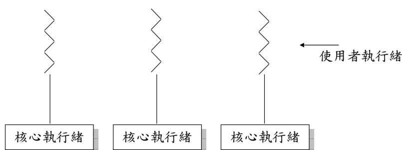
49

多執行緒的模型

- 實作執行緒時通常有三種模型
  - 一對一模型：將一個使用者執行緒對應到一個核心執行緒，比多對一模型提供更多並行處理的能力，當行程中的執行緒暫停執行，其他執行緒還能繼續執行，而且此模型允許多個執行緒同時在多個CPU上並行執行，此模型缺點是每當產生一個新的使用者執行緒時，必須產生相對的核心執行緒，如此造成系統額外的負擔，使用此模型時，大都會限制系統最大執行緒數目。

50

一對一模型



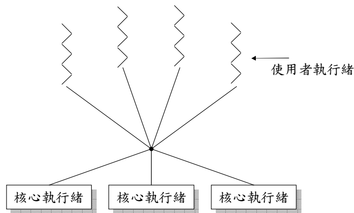
51

多執行緒的模型

- 實作執行緒時通常有三種模型
  - 多對多模型：將使用者執行緒對應到相同或是較少數目的核心執行緒，有些應用程式會需要一定數量的執行緒，雖然多對一模型可以滿足，但在同一時間內各個執行緒無法並行地執行，若使用一對一模型，程式設計師必須小心地設計程式，不能在程式中產生過多核心執行緒而增加太多額外負擔，若利用多對多模型，程式設計師可以產生自己所需數目的使用者執行緒，相對的核心執行緒也可以在多CPU的環境下並行地執行。

52

多對多模型



53

行程合作(IPC)(96nttu、92nttu)

- 當多個行程同時在一個系統中執行時，可能會形成：
  - 多個獨立的行程 - 指行程之間沒有任何共享的資料。
  - 一些合作的行程 - 指行程之間有共享的資料，為達此目的作業系統提供一些方法，來支援行程間的溝通或進行行程間的同步。

54

合作行程間通訊

- 合作行程(cooperating process)
  - 資訊共享：因為數個使用者可能對相同的資訊(例如：公用檔案)有興 趣，因此必須提供一個環境允許使用者能同時使用這些資源。
  - 加速運算：如果希望某一特定工作執行快一點，則可以分成一些子工作， 每一個子工作都可以和其它子工作平行地執行。(多CPU)
  - 模組化：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。
  - 方便性：即使是單一使用者也可能同時執行數項工作。

55

合作行程間通訊

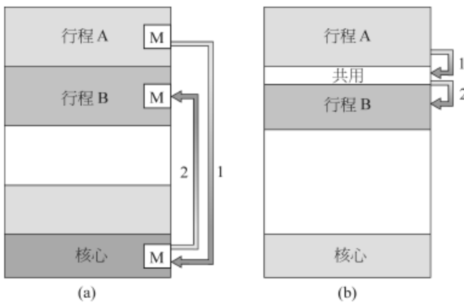


圖 3.12 通信模式；(a) 訊息傳遞；(b) 共用記憶體

合作行程間通訊- Mach

- 在訊息式(94nttu)的作業系統例子中，考慮由Carnegie Mellon大學所發展出的Mach作業系統。Mach核心支援多元任務的產生和刪除，這些任務類似於行程，但具有多重的控制執行緒。
- Mach中大部份的通訊(包括大部份系統呼叫和所有任務之間的資訊)都是由訊息完成。訊息都是由信箱(在Mach中乃稱為埠，port)來傳送及接收。

57

合作行程間通訊- Windows XP

- Windows XP作業系統是現代設計的一個例子，它採用模組化設計以便增進功能和降低製作新特性所需的時間。Windows Xp 提供多元操作環境的支援，或是經由一個訊息傳遞的功能完成應用程式之連結的子系統。這些應用程式可以視為WindowsXP系統伺服器的委託人。

圖 3.16 Windows XP 中的區域程序呼叫

58

遠程程序呼叫(RPC)

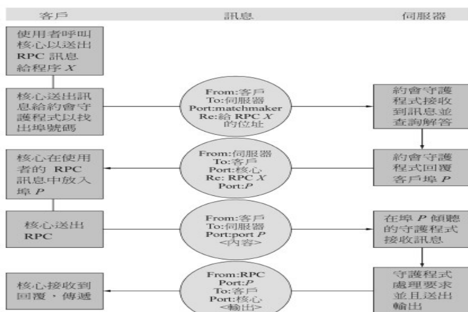


圖 3.20 遠程程序呼叫（RPC）的執行

59

遠端方法呼叫(RMI)

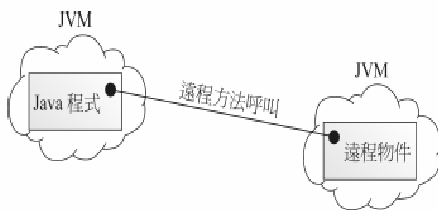


圖 3.21 遠程方法呼叫

60

遠端方法呼叫(RMI)

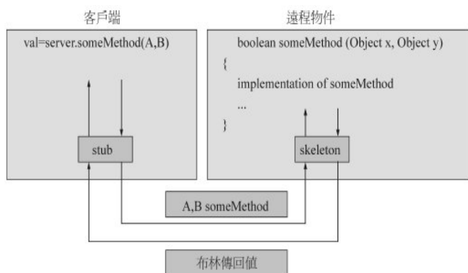


圖 3.22 排列參數

61

共享緩衝區

- 典型行程合作的問題：
  – 生產者 - 產生資訊給消費者
  – 消費者 - 消耗生產者所產生的資訊
- 當生產者和消費者同時執行時，需要有一個共享緩衝區給生產者存放產品，以提供給消費者使用，此二行程的動作必須同步，才不會造成當緩衝區滿了，生產者還繼續生產，緩衝區空了，消費者還繼續消費。

62

緩衝區

- 緩衝區可分為：
  - 無限緩衝區➔緩衝區大小沒有限制，生產者不用擔心緩衝區會不夠用，但消費者在緩衝區空了的時候要等待新產生的資料到達才能開始動作
  - 有限緩衝區➔緩衝區大小有限制，緩衝區滿了生產者必須暫停生產，當緩衝區空了消費者必須等待直到緩衝區有資料，才能進行資料取得的動作。
- 緩衝區由作業系統提供的IPC機制來產生，或是直接經由程式設計師在程式中使用共享記憶體的機制來進行。

63

生產者與消費者的例子

- 使用共享記憶體機制
- 生產者及消費者的行程共享了下列的變數：
  - #define BUF_SIZE n
  - int iIn = 0, iOut = 0
  - user_def_type buffer[BUF_SIZE]
- 這個例子中同時最多只能使用到 n-1 個緩衝區的空間why?

64

生產者與消費者的例子

- iIn代表緩衝區內下一個可放資料的位置，及生產者下一個可存放資料的位置。iOut代表緩衝區內最先可用位置。若iIn=iOut 代表緩衝區是空的，(iIn+1)%BUF_SIZE=iOut代表緩衝區已滿why?

65

生產者

```
do {
        …
        產生一個型別為 user_def_type 的資料並存放於 nextp，
         nextp為生產行程內的區域變數
        …

        /* 若緩衝區已滿則等待 */
        while ((iIn + 1) % BUF_SIZE == iOut)  ;                    ;
        buffer[iIn] = nextp;
        iIn = (iIn + 1) % BUF_SIZE;
}
while (True);
```

66

消費者

```
do {
        while (iIn = iOut)  ; /* 若緩衝區為空則等待 */
                ;
        nextc = buffer[iOut];
        iOut = (iOut + 1) % BUF_SIZE;
                …
        將存於 nextc 的資料消耗掉，
         nextc為消費行程內的區域變數
                …
}
while (True);
```

67

行程間溝通

- 共享記憶體利用一塊行程間共享的緩衝區來進行合作的溝通，而溝通的方式完全由程式設計師自行設計。
- 行程間溝通(IPC)由作業系統提供。
- IPC包含了一些機制，讓行程與行程間能夠進行溝通與同步。
- 訊息傳遞系統是相當基本的IPC機制，行程間可建立溝通管道來進行同步或資料共享，不必使用相同的位址空間。
- 共享記憶體或是IPC機制是可以同時使用。

68

基本架構

- 行程間溝通(IPC)為作業系統所提供用來達到行程間資料交換與共享的機制。
- IPC 通常會提供兩個函式：
  - send() - 傳送訊息
  - receive() - 接收訊息
- 行程在傳送訊息時可有兩種選擇：固定大小的訊息、可變大小的訊息。
- 固定大小訊息在系統實作時比較簡單，但這限制會讓程式設計較為複雜，因程式設計者需自行處理超過固定大小的訊息。

69

基本架構

- 可變大小訊息在系統實作時比較複雜，但會使程式設計較為簡單且有彈性。
- 行程間在溝通時會建立一條通訊鏈結。透過鏈結send() 和 receive() 就可以傳送和接收訊息。
- 實作上要考慮的問題包括：
  - 鏈結是否可以共享給2個以上的行程使用？
  - 訊息大小應是固定大小還是可變大小？
  - 鏈結是單向(只能傳送或是接收)的還是雙向的？

70

基本架構

- 實作傳送與接收功能時有幾種方法：
  - 直接或間接的溝通
    - 對稱或非對稱溝通
  - 傳送複製或只傳送參考

71

直接溝通

- 每個行程要與其他行程進行溝通時，必須要明確地指出訊息傳送的目的地或接收訊息的來源。

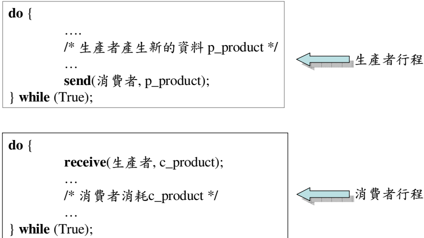
  send(A, message)　/*傳送訊息給 A 行程*/
  receive(B, message) /*從 B 行程接收訊息*/

- 鏈結的特性是兩個行程在進行溝通時雙方必須要知道對方的身分，一個鏈結只會有兩個行程參與，且鏈結通常是雙向的
- 生產者產生一新的資料後會呼叫send()，消費者要消耗一份資料時會呼叫receive()。

72

直接溝通(續)

- 使用 IPC 機制程式較容易撰寫(比對Slice 66)。



直接溝通

- 可分為對稱與非對稱，而send() 和receive() 分別被定義成
  - 對稱 – 發送端與接收端必須互相指定對方的名字
    send(A, message)　/*傳送訊息給 A 行程*/
    receive(B, message) /*從 B 行程接收訊息*/
  - 非對稱 – 只有發送端需指定對方的名字而接收端則不需要
    send(A, message)　/*傳送訊息給 A 行程 */
    receive(id, message)　/*從任何一個行程接收一個訊息。其中 id 這個變數代表正在與接收端進行溝通的行程 */

直接溝通

- 兩者皆有相同的缺點，行程的名稱若改變了，則必須把所有參考到這的行程的參照一一更新，此舉會造成維護程式的額外負擔。

75

間接溝通

- 訊息的傳送與接收是透過信箱來完成。
- 每個信箱都有一個獨一無二的編號，行程可透過不同信箱編號與其他數個行程進行溝通
  send() 與 receive() 可定義為
  - send(A, message) /*傳送一個訊息到 A 信箱*/
  - receive(A, message) /*從 A 信箱接收一個訊息*/
- 兩個行程溝通時，他們一定共用相同的信箱，鏈結可以同時給數個行程使用，而且可以是單向或是雙向

76

間接溝通

- 行程A呼叫send(X, message) ，行程B、C又呼叫receive(X, message)？
- 多個行程同時共用相同的信箱，若多個行程同時接收訊息，那個行程會收到訊息？
  - 同時只允許兩個行程共享信箱
  - 同時只允許一個行程呼叫 receive()
  - 交給作業系統，讓作業系統決定由誰得到訊息

77

間接溝通

- 信箱擁有者可以是一個行程或是作業系統，每個信箱都有唯一的擁有者，當一個擁有者行程結束時，信箱也跟著被銷毀，若被銷毀的信箱還有其他行程傳送訊息給這個信箱，則這些傳送訊息的行程必須被告之此信箱已經不存在。

78

間接溝通

- 行程宣告一個型別為mailbox的變數，即表示擁有此信箱，其他知道此信箱名稱的行程就是信箱的使用者
- 若信箱數於作業系統，則作業系統必須提供一些方法給行程，允許行程去：
  - 建立一個新的信箱
  - 透過信箱去傳送和接收訊息
  - 銷毀一個信箱

79

間接溝通

- 透過作業系統建立信箱的行程即信箱的擁有者，一開始只有信箱的擁有者，可以透過該信箱接收訊息
- 行程A建立信箱M，行程A呼叫fork()建立一個新的行程B，行程A與B即共享信箱M
- 信箱銷毀後，所有對此信箱存取的行程就不能再用，且作業系統必須回收信箱所使用的資源，並作後續處理如轉信到新信箱等

80

同步

- 行程間可透過 send() 與 receive() 來進行同步。
- 實作 send() 與 receive() 時，可為
  - 阻隔式發送：發送訊息的行程將會被阻隔，直到訊息被接收端或信箱收到為止，才可執行下一個動作
  - 非阻隔式發送：發送訊息的行程於送出訊息後，馬上可繼續執行下一個動作(UDP)
  - 阻隔式接收：接收端的行程將會被阻隔，直到收到訊息才可執行下一個動作
  - 非阻隔式接收：接收端的行程不論收到訊息與否，都可以執行下一個動作
- 當 send() 與 receive() 皆為阻隔式時，發送端與接收端之間就會是同步的。

緩衝

- 一個鏈結中，可能設置緩衝區來暫時儲存正在鏈結中傳遞的訊息，一般是用訊息佇列來實作。
- 實作訊息佇列時可使用下列幾種方法：
  - 無緩衝：正在傳遞的訊息會停留在鏈結中，發送端必須等待接收端接收到資料，才能繼續傳送下一個訊息，發送端與接收端是同步
  - 有限緩衝：鏈結中緩衝區最多只有n個訊息，發送端若緩衝區沒滿，可以繼續傳送訊息，不需等待接收端迴應接收到資料，但若緩衝區已滿，則必須等待緩衝區有空的位置才能發送訊息
  - 無限緩衝：鏈結中緩衝區沒有大小限制，無論訊息多少都可以存在緩衝區，發送端不需等待緩衝區，隨時可以傳送

緩衝

- 使用有緩衝的訊息佇列時，若行程間需要同步則需另外處理。
- 行程A送訊息給行程B，行程A必須等待行程B收到訊息才能繼續執行➔同步。
  行程A：send(B,msg);   receive(B, msg);
  行程B：receive(A, msg); send(A,"ACK");

83

例外狀況

- 在分散式系統中，行程在進行溝通出現錯誤的機率會比在單一機器上來得高，因為分散式系統相對較為複雜。
- 在單一機器環境下，訊息傳遞會用共享記憶體來實作，若有一錯誤產生，整個系統皆會受到影響
- 在分散式的環境下，若一個訊息在傳遞過程發生錯誤，這個錯誤不一定會影響整個系統的運作

84

例外狀況

- 無論在單一機器或是分散式的環境都需要有錯誤回覆的機制將系統回復正常運作，例外處理就是用來處理各種例外狀況的一種錯誤回復機制

85

例外狀況

- 當下列狀況發生時，系統可能需要進行錯誤的回復：
  - 行程結束：無論是傳送端或是接收端，都有可能在未完成前不正常地結束，這產生訊息遺失或是行程在等待永遠不會被發送的訊息。如此會有兩種問題：
    - 行程A在等待行程B送來的訊息，但行程B已經結束。OS應將行程A結束，或通知行程A行程B已經結束
    - 行程A傳送訊息給已經結束的行程B，若行程A需要知道行程B是否已經收到訊息，那麼行程A必須要等待行程B的回覆➔有緩衝則沒問題，否則若以阻隔式發送的行程將會被永遠隔絕

86

例外狀況

– 訊息遺失：訊息被傳送過程很有可能會遺失，造成遺失的可能原因有很多，通常是硬體或網路壅塞，此時有些基本的處理方法：
  - 由OS來負責偵測此類問題，當遺失訊息時由OS自動重送，但是此法會造成系統負擔，且系統不知如何對個別的行程特別處理
  - 由發送訊息的行程來負責偵測，若有遺失訊息則此行程會自動重送，但此法會增加行程負擔與應用程式設計的困難，且行程也無系統全局的資訊做特別的處理
  - 由OS來負責偵測此類問題，當遺失訊息時由OS不直接重送，而是通知行程訊息遺失，由行程決定是否該重送

87

例外狀況

– 訊息遺失的偵測並不是必要的動作。在網路通訊協定中UDP不會做遺失偵測，TCP則會偵測並保證訊息的到達，若訊息遺失則會重送遺失封包

– 可用逾時的機制(TCP/IP連接網站逾時內定是多久？ TCP/IP傳送封包若遺失最多會重傳幾次才宣告失敗？)

88

例外狀況

- 訊息的錯誤：訊息被傳送至目的地時，訊息資料有錯誤，這錯誤來自外界的干擾所造成，例如電磁波，這種狀況與訊息遺失類似，通常OS會重送訊息，檢查碼(CRC？、checksum？、parity)通常被用來偵測這類的錯誤(chap. 15 CRC16、CRC32、ECC)

同位元偵錯法(Parity)

無論什麼碼，在傳送時都可能產生邏輯上的辨認錯誤，同位元偵錯法就是一種能夠分辨傳送碼有無錯的方法，分為偶同位元(even parity bit)及奇同位元(odd parity bit)兩種編碼檢查方式，做法上是將一個尚未傳送的編碼，附加上一個額外的位元，若要整個編碼內保持偶數個1，這個位元就叫做偶同位元，若要整個編碼內保持奇數個1，這個位元就叫做奇同位元.

同位元偵錯法(Parity)

- 例如某二進碼尚未加入同位元之前為0100101，此碼內有奇數個1，若採用偶同位元編碼，編碼後為10100101，使其變為偶數個1，若採用奇同位元編碼，編碼後為00100101，使其保持奇數個1。加了同位元的編碼在傳送後，若已與接收端取得偶同位元或是奇同位元約定，接收端可以經由同位元檢查電路偵察出目前的傳送碼是偶數個1還是奇數個1，進而知道傳送過程有無產生錯誤

摘要

- 行程簡介
  - 行程就是一個執行中的程式。
- 行程的狀態
  - 新建
  - 執行
  - 等待
  - 就緒
  - 終結
- 作業系統中的佇列
  - 就緒佇列
  - I/O 佇列
  - 等待佇列

摘要

- 作業系統中主要的排程器
  - 長程排程器
  - 短程排程器
- 內文切換
  - 將上一個行程的相關資訊先保存起來，並且把即將要使用 CPU 的行程的相關資訊載入系統中。
- 執行緒的概念
  - 減少行程間切換所造成的額外負擔。

93

摘要

- 同一個行程中的執行緒共用
  - 程式區段、資料區段
  - 一些從作業系統取得的資源
- 行程同時存在系統中執行時，可能會是
  - 獨立的行程
  - 合作的行程
- 作業系統提供一些方法讓行程之間能夠溝通
  - 共享記憶體
  - 訊息傳遞系統

94