Chapter 4

執行緒與並行性

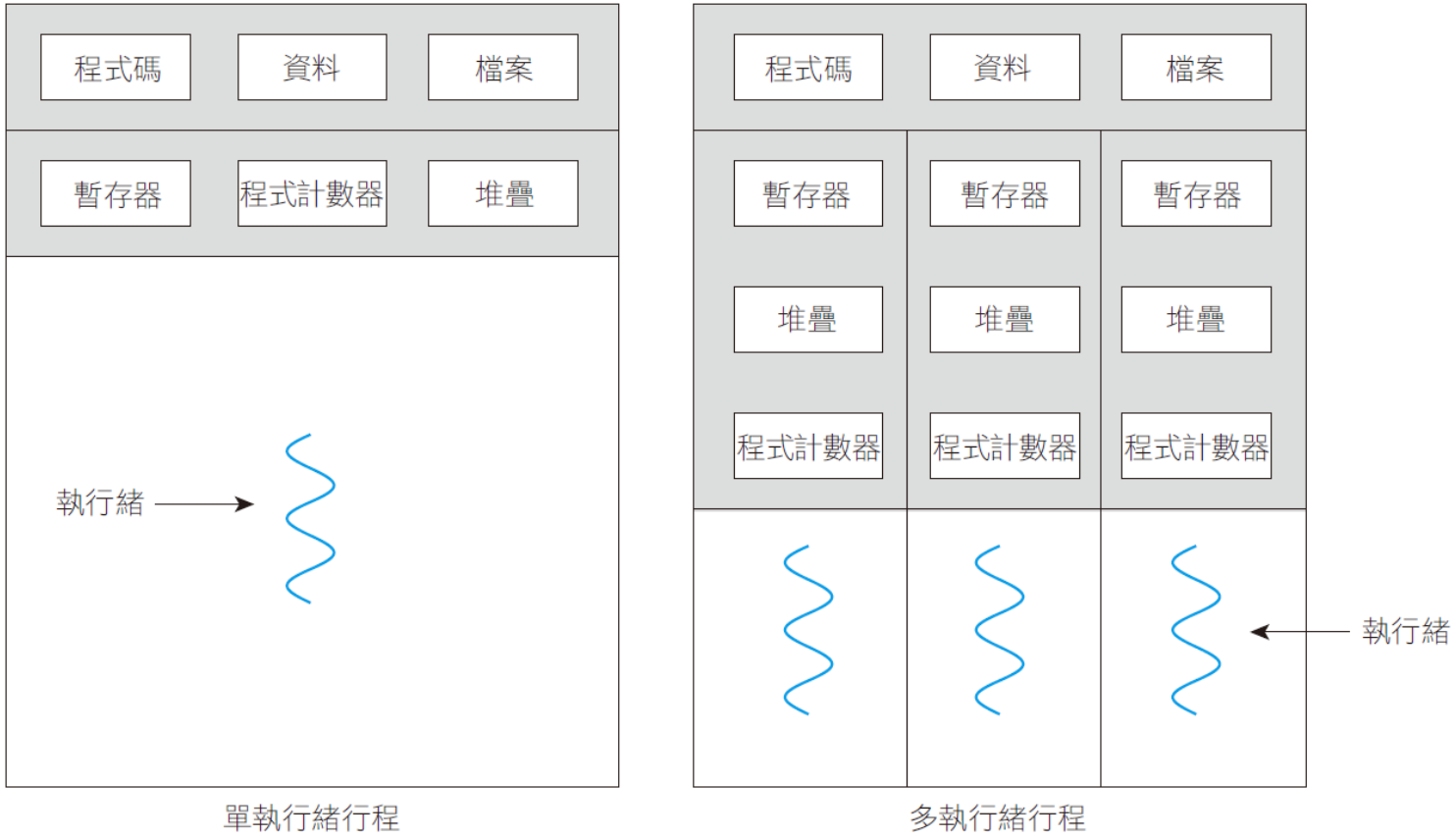
章節目標

- 確定執行緒的基本元素，並比較執行緒和行程
- 描述設計多執行緒行程的主要好處和重大挑戰
- 說明不同執行緒的處理方法，包括執行緒池、fork-join 和 Grand Central Dispatch
- 描述 Windows 和 Linux 作業系統的表示方式執行緒
- 使用 Pthreads、Java 和 Windows 設計多執行緒應用程式的執行緒 API

4.1 概 論

- 執行緒是 CPU 使用時的一個基本單位，是由一個執行緒 ID、程式計數器、一組暫存器，以及一個堆疊空間所組成
  - 它和屬於同一行程的其它執行緒共用程式碼區域、資料區域和作業系統資源
  - 傳統的行程只有單一執行緒控制
  - 如果一個行程有多個執行緒控制，可以一次執行一項以上的任務
  - 圖4.1 描述傳統的**單執行緒** (singlet-hreaded) 行程和**多執行緒** (multi-threaded) 行程的差別

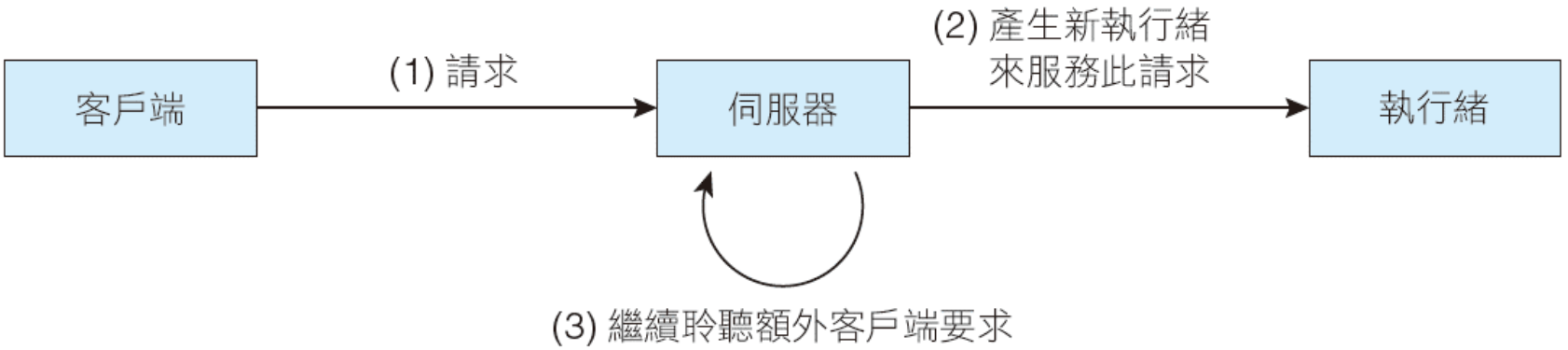
圖 4.1　單執行緒行程與多執行緒行程



4.1.1 動 機

- 許多在現代電腦和行動裝置執行的套裝軟體都是多執行緒
  - 應用程式通常製作成有許多執行緒控制的個別行程
  - 以下重點介紹一些多執行緒應用程序範例：
    - 應用程式從照片縮圖產生的圖像集合，或是使用單獨的執行緒從每個單獨的圖像生成縮圖
    - 網頁瀏覽器可能有一個執行緒顯示圖像或文件，而另一個執行緒則從網路取得檢索資料
    - 文書處理可能有一個用於視窗顯示的執行緒、另一個用於回應使用者輸入的執行緒，以及第三個執行緒在負責後台執行拼字和語法檢查
  - 大部份的作業系統核心是多執行緒

圖 4.2　多執行緒伺服器架構



4.1.2 利　益

1. 應答：將交談式的應用程式多執行緒化，可以在一個程式中的某一部份被暫停，或程式在執行冗長的操作時，依然繼續執行，因此增加對使用者的應答
   - 這項特性在設計使用者介面時特別有用
     - 例如，考慮當使用者按一下按鈕產生一個費時的操作會發生什麼事
     - 一個單執行緒的應用程式將無法回應使用者，直到操作結束
     - 如果這個費時的操作用一個單獨的執行緒執行，則應用程式仍可保持對使用者的回應

4.1.2 利 益

2. 資源分享：行程只能經由共用記憶體和訊息傳遞等技巧分享資源
   - 這些技巧必須由程式人員明確地安排
   - 而執行緒間將共用它們所屬行程的記憶體和資源
   - 程式碼和資料的好處是讓應用程式有數個不同的執行緒在同一位址空間活動

4.1.2 利　益

3. 經濟：對於行程產生所配置的記憶體和資源耗費很大

- 因為執行緒共用它們所屬行程的資源，所以執行緒的產生和內容交換就比較經濟
- 憑經驗測量產生和維護行程比執行緒多出多少時間可能很困難
  - 通常產生和維護行程會比執行緒更費時
- 執行緒之間的內容轉換通常比行程之間的更快

4.1.2 利　益

4. 可擴展性：在多處理器的架構下，多執行緒的利益可以大幅提升
   - 因為每一執行緒可以並行地在不同的處理核心上執行
   - 不論有多少處理器可以使用，單執行緒只能在一個處理器上執行

4.2 多核心程式撰寫

- 並行 (concurrency)
  - 在一個單運算核心的系統，並行僅意味著執行緒的執行是隨著時間的推移而交錯
  - 因為處理核心一次只能執行一個執行緒

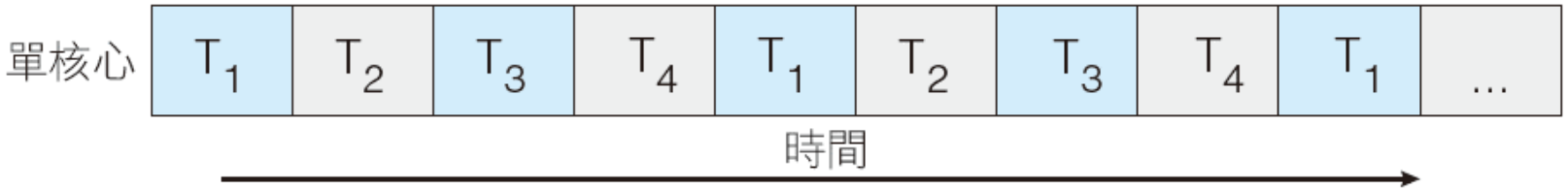


圖 4.3 在單核心系統的並行執行

4.2 多核心程式撰寫

- 平行 (parallelism)
  - 在一個多運算核心的系統，並行執行只表示執行緒的執行可以平行的執行
  - 因為系統可以指定一個單獨的執行緒給每一個核心

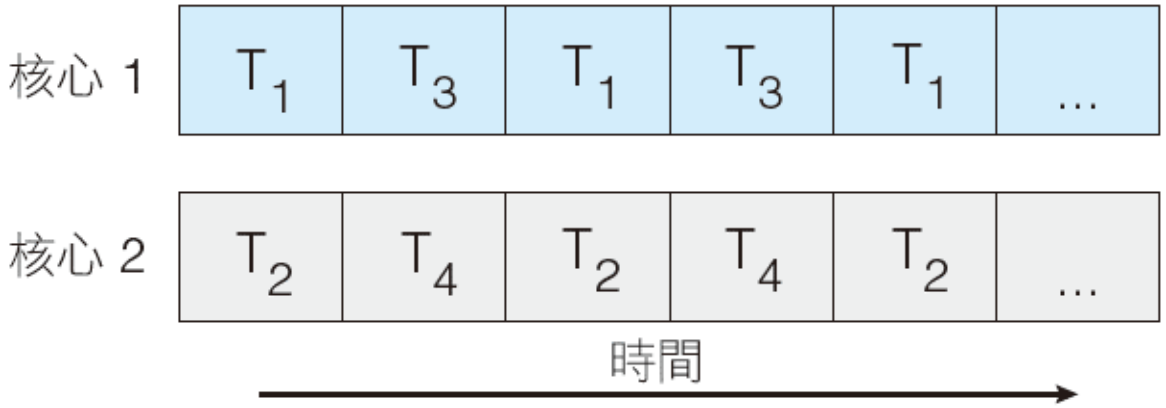


圖 4.4 在多核心系統的平行執行

4.2 多核心程式撰寫

- 一個系統能同時執行一個以上的任務時就是平行
- 一個並行系統則是藉由允許所有的任務有進展來支援一個以上的任務
  - 因此，有可能有並行但沒有平行
- 在多處理器和多核心架構來臨前，大部份的電腦系統只有一個單一處理器
  - CPU 排班程式被設計成藉由快速地在系統的行程間切換來提供平行的錯覺，因此允許每個行程有進展
  - 這些行程並行地執行，但不是平行

4.2.1 程式撰寫的挑戰

- 在撰寫多核心系統的程式時，有五個領域面臨挑戰：
  - 辨識任務
  - 平衡
  - 資料分割
  - 資料相依
  - 測試與偵錯

阿姆達爾定律

- 同時當具有串行功能的應用加入其它計算核心 (非平行) 和平行元件
  - 如果 S 是應用程式必須在具有 N 個處理核心的系統以串行執行，公式如下所示：

$$speedup \leq \frac{1}{S + \frac{(1-s)}{N}}$$

  - 假設我們有一個 75% 的平行，25% 串列。如果我們在具有兩個處理能力的系統上執行這樣的應用，就可以獲得 1.6 倍的加速 (speedup)
  - 當 N 接近無窮大時，加速收斂到 $1/S$
  - 應用的串列部分對於我們加入額外運算核心在效能的獲得上有決定性影響

4.2.2 平行的類型

- **資料平行** (data parallelism) 強調分配同一筆資料的子集合到多個運算核心，並在每一個核心執行相同的操作
- **任務平行** (task parallelism) 牽涉到分配任務 (執行緒) 而非資料到多個運算核心
  - 每一個執行緒執行一個獨一無二的操作

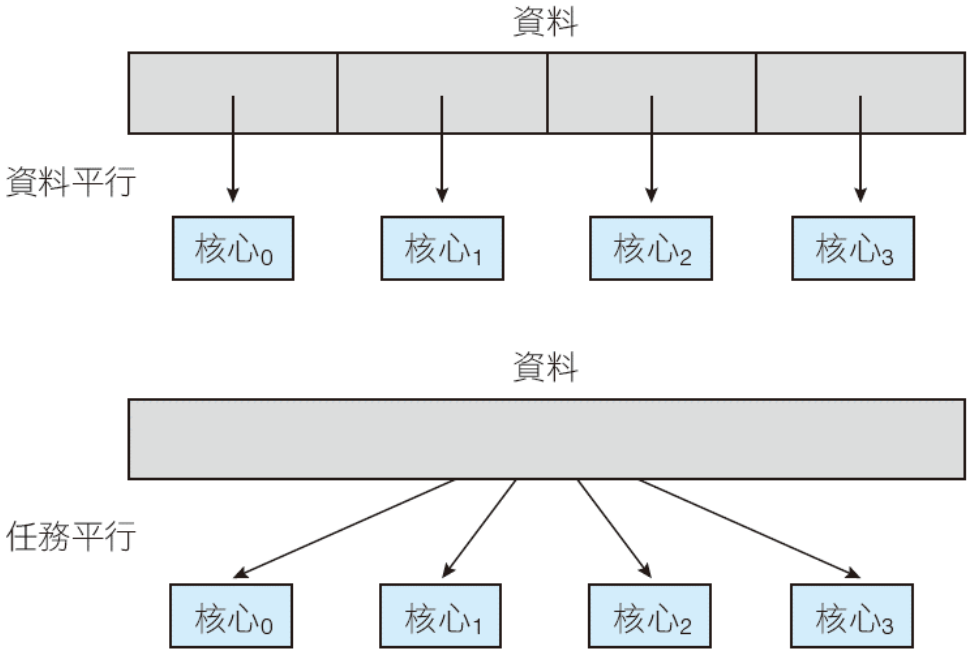


圖 4.5 資料平行與任務平行

4.3 多執行緒模式

- **使用者執行緒** (user thread)，或是由核心提供**核心執行緒** (kernel thread)
- 使用者執行緒的支援是在核心之上，而且在沒有核心支援下管理，因此核心執行緒直接由作業系統支援和管理
- 幾乎所有近代的作業系統都支援核心執行緒—包括 Windows、Linux 和 macOS

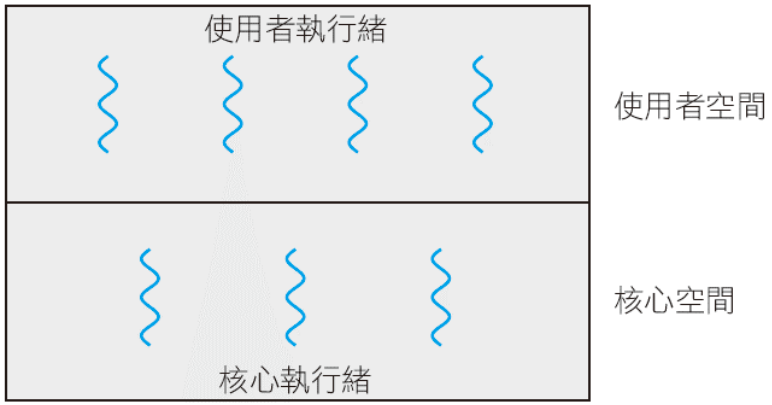


圖 4.6 使用者與核心執行緒

4.3.1 多對一模式

- 多對一模式 (圖 4.7) 將許多個使用者層級的執行緒映射到一個核心執行緒
- 但如果一個執行緒呼叫一個暫停的系統呼叫時，整個行程就會暫停
  - 同時，因為一次只有一個執行緒可以存取核心，數個執行緒不能在多核心系統上平行地執行
  - **綠執行緒** (green thread) —— Solaris 系統的執行緒程式庫
  - 很少有系統繼續使用此模式

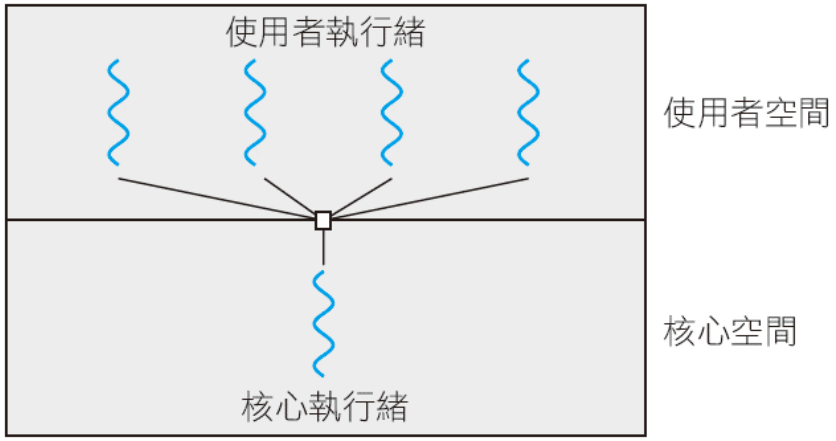


圖 4.7 多對一模式

4.3.2 一對一模式

- 一對一模式 (圖 4.8) 將每一個使用者執行緒映射到一個核心執行緒
  - 它提供比多對一模式更多的並行功能
  - 產生使用者執行緒時就要產生相對應的核心執行緒，這種模式的大部份實作都限制系統支援的執行緒個數
  - Linux 和 Windows 作業系統的家族都實作一對一模式

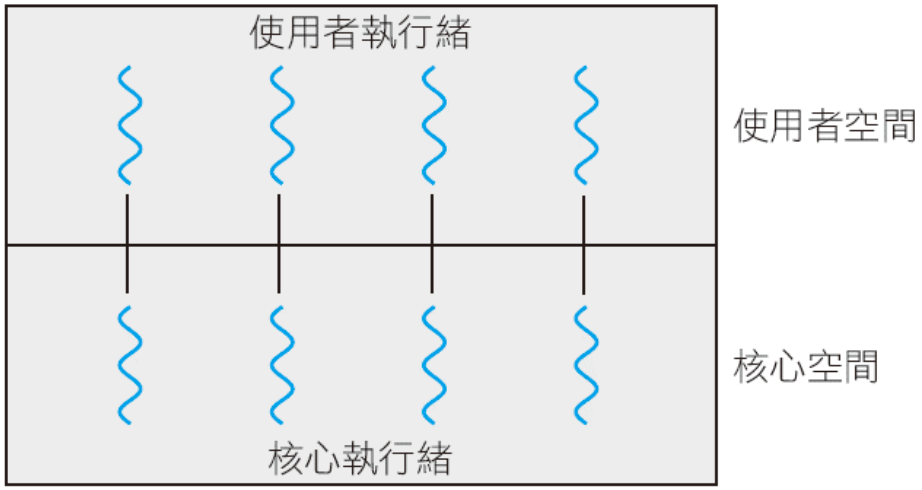


圖 4.8 一對一模式

4.3.3 多對多模式

- 多對多模式 (圖 4.9) 將許多使用者執行緒映射到較少或相等數目的核心執行緒
- 允許程式開發人員產生她所希望數目的執行緒
- 多對多模式中一種常用的變化形式
  - 允許使用者層級的執行緒被連結一個核心執行緒

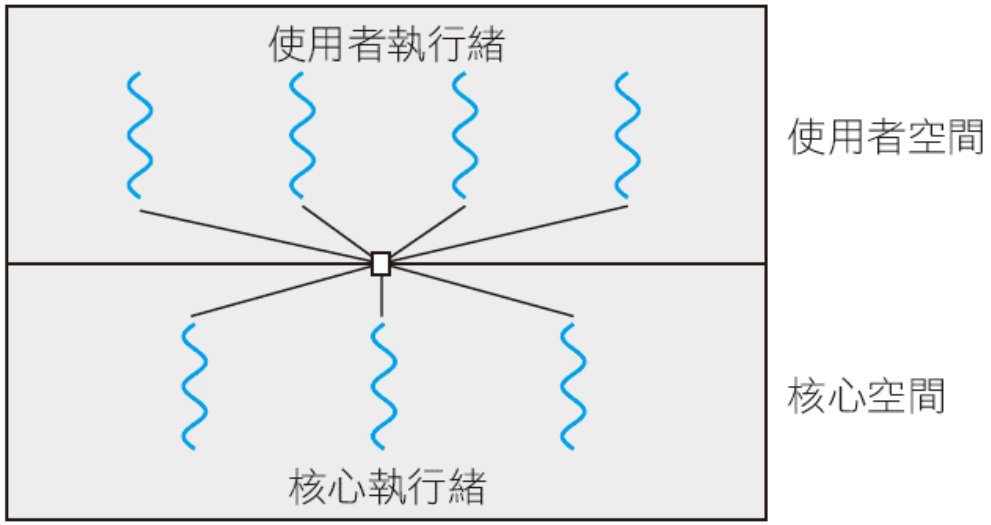


圖 4.9 多對多模式

二層模式 (two-level model)

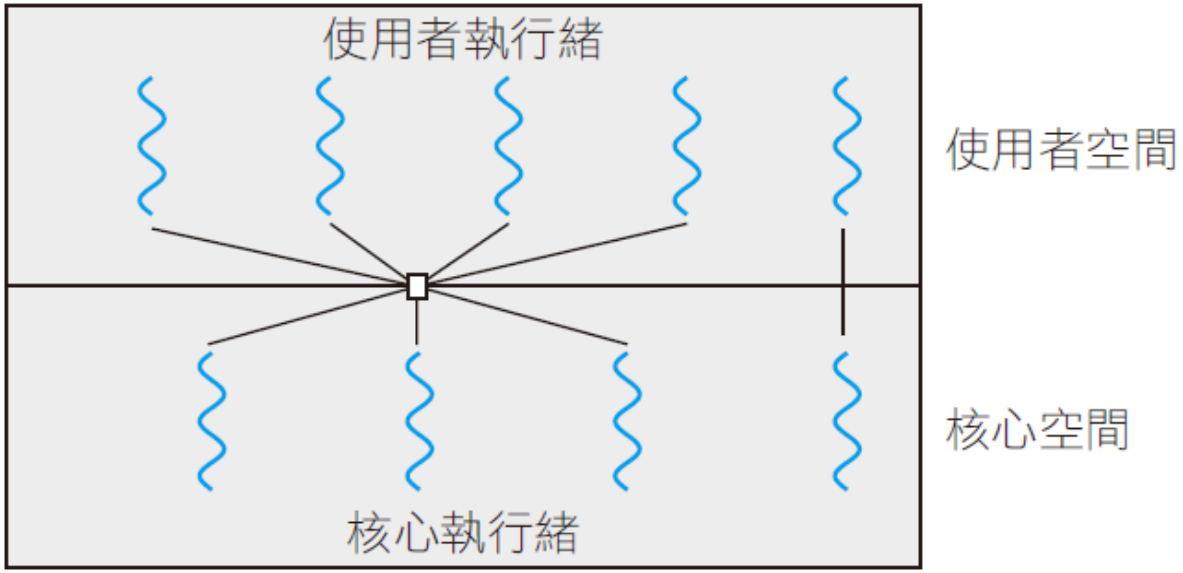


圖 4.10　兩層模式

4.4 執行緒程式庫

- **執行緒程式庫** (thread library) 提供程式設計師一個 API 來產生和管理執行緒
- 製作執行緒程式庫有兩個主要的方法
  - 使用者空間提供完整的程式庫
  - 直接由作業系統支援的核心層級程式庫

4.4.1 Pthreads

- POSIX 標準 (IEEE 1003.1c) 定義執行緒產生和同步的 API
  - Pthreads 是執行緒行為的規格，而非製作
- 大多數是 UNIX 型態的系統，包括
  - Linux
  - macOS

圖 4.11 使用 Pthreads API 多執行緒的 C 程式

```c
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

圖 4.12　加入 10 個執行緒的 Pthread 程式碼

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```


4.4.2 Windows 執行緒

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
```


4.4.2 Windows 執行緒

```
Param = atoi(argv[1]);
/* create the thread */
ThreadHandle = CreateThread(
   NULL, /* default security attributes */
   0, /* default stack size */
   Summation, /* thread function */
   &Param, /* parameter to thread function */
   0, /* default creation flags */
   &ThreadId); /* returns the thread identifier */

 /* now wait for the thread to finish */
WaitForSingleObject(ThreadHandle,INFINITE);

/* close the thread handle */
CloseHandle(ThreadHandle);

printf("sum = %d\n",Sum);
}
```


4.4.3 Java 執行緒

- Java 執行緒在任何提供 JVM 的系統都可以使用，包括 Windows、Linux 和 macOS
- Java 程式產生執行緒：
  - 一種方法是繼承自 Thread 類別所產生的新類別，並且覆蓋 Thread 類別的 run() 的方法
  - 一種常用的變通方法則是，定義一個製作 Runnable 介面的類別
    - 該介面定義了一個抽象的簽署方法為 public void run()
    - 實現 Runnable 的 run() 類別方法中的程式碼能單獨在執行緒中執行

4.4.3 Java 執行緒

- 如以下的例子所示：

```
class Task implements Runnable
{
  public void run()  {
    System.out.println("I am a thread.");
  }
}
```


圖 4.14 Java Executor 框架 API 的圖示

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}

public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
   }
}
```


4.5 隱式執行緒

- 在多核心處理的持續成長下，包含數百個—甚或數千個—執行緒的應用程式隱約可見
- 設計這種應用程式不是一件簡單的工作：
  - 解決這些困難和對設計多執行緒應用程式更佳支援的一個方法是，轉換執行緒的產生和管理
  - 從應用程式開發人員到編譯器和執行階段程式庫，這種策略稱為**隱式執行緒** (implicit threading)，是日益受到歡迎的趨勢
  - 這些策略通常要求應用程式開發人員確定可以平行執行的任務，而不是執行緒

4.5.1 執行緒池

- 產生一些執行緒
  - 並且放入一個池中，這些執行緒就坐著等待工作
- 優點：
  1. 服務一項要求時，使用現存的執行緒會比等待產生一個執行緒要快
  2. 執行緒池限制任何時候執行緒的個數，這對無法支援大量並行執行緒的系統特別重要
  3. 將執行任務與產生任務的機制分開來，讓我們使用不同的策略執行任務
     - 例如，任務可能在延遲一段時間後被排班執行，或是週期性地執行。

4.5.1 執行緒池

- Windows API 提供與執行緒池相關的一些功能

```
DWORD WINAPI PoolFunction(PVOID Param) {
   /*this function runs as a separate thread.*/
}
```


4.5.3 OpenMP

- 一組編譯指示和一個 API 給使用 C、C++ 或 FORTRAN 寫的程式用
  - 它提供共用記憶體環境下的平行程式支援
  - 會辨認**平行區域** (parallel region) 為可以平行執行的程式碼區塊

4.5.3 OpenMP

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
     printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}

#pragma omp parallel
```


4.5.4 Grand Central Dispatch

- Apple macOS 和 iOS 作業系統的一項技術
  - 它是擴展 C 語言、API 和執行階段程式庫的組合
  - 允許應用程式開發人員區分出要平行執行的程式碼區塊 (任務)
  - 管理大部份執行緒的細節
- GCD 將區塊放入**分派佇列** (dispatch queue) 為執行階段的區塊排班任務執行
  - 當分派佇列從佇列移除一個區塊時，它會指定此區塊給所維護之執行緒池的一個可使用執行緒
  - 區塊由插入在一對大括號 {} 前的插入符號 ^ 表示，括號中的程式碼標識所要執行的工作單元

```
^{ printf ("I am a block"); }
```


4.5.4 Grand Central Dispatch

- GCD 可分辨兩種型態的分派佇列：
  - 串列 (serial)
  - 並行 (concurrent)
- 串列
  - 區塊是以 FIFO (先進先出) 的順序移除
  - 被稱為**主佇列** (main queue)]
    - 開發人員可以產生局限於特殊行程的其它串列佇列
- 並行
  - 以 FIFO (先進先出) 的順序移除
  - 一次可以有幾個區塊被移除

4.6 執行緒的事項

- 信號處理
  - 同步或非同步
- 執行緒取消
  1. 非同步取消 (Asynchronous cancellation)
  2. 延遲取消 (Deferred cancellation)
- 執行緒的局部儲存
- 排班器活化作用

4.6.1 fork() 和 exec() 系統呼叫

- 如果程式中的一個執行緒呼叫 fork()，則新的行程會複製所有的執行緒
  - 或是新的行程只是單執行緒呢？
  - 有些 UNIX 系統選擇擁有兩種版本的 fork()
- exec() 參數所指定的程式將取代整個行程
  - 包括所有的執行緒

4.6.2 信號處理

- 在 UNIX 系統中，**信號** (signal) 被用來通知行程，一個特殊的事件已經發生了
- 無論信號是同步或非同步，所有的信號都遵循相同的形式：
  1. 信號由於特定事件的發生而產生
  2. 產生的信號被送到一個行程
  3. 一旦送達後，此信號必須處理

4.6.2 信號處理

- 每一個信號可能會被兩種可能的處理器之一所處理：
  1. 預設的信號處理器
  2. 使用者定義的信號處理器
- 每一個信號有一個**預設的信號處理器** (default signal handler)，是在核心處理該信號時執行
  - 這個預設動作可能被**使用者定義的信號處理器** (user-defined signal handler) 函數所覆蓋
- 在單執行緒的程式處理信號很直接：信號直接送給行程

4.6.2 信號處理

- 在多執行緒的程式一個信號應該被傳送到哪裡呢？
  1. 傳送信號到此信號作用的執行緒
  2. 傳送信號到行程中的每一個執行緒
  3. 傳送信號到行程中特定的執行緒
  4. 指定一個特定的執行緒來接收該行程的所有信號

4.6.3 執行緒取消

- **執行緒取消** (Thread cancellation) 是在一個執行緒完成之前結束它
  - 被取消的執行緒通常稱為**目標執行緒** (target thread)
    1. 非同步取消 (Asynchronous cancellation)
       - 一個執行緒立即終止目標執行緒
    2. 延遲取消 (Deferred cancellation)
       - 目標執行緒可以週期地檢查它是否該被取消，這允許目標執行緒有機會以有條不紊的方式結束自己

4.6.3 執行緒取消

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_ cancel (tid);

/* wait for the thread to terminate */
pthread join(tid,NULL);
```


4.6.3 執行緒取消

- 呼叫 pthread_cancel() 表示只是要求取消目標執行緒
  - 實際的取消是根據目標執行緒如何被設定來處理此要求

模式	狀態	型態
Off	關閉	—
Deferred	啟用	延遲
Asynchronous	啟用	非同步

  - 如果取消功能被關閉則執行緒不能被取消
  - 而這個取消要求留在等待狀態，所以執行緒可稍後啟用取消功能，並對此要求做回應

4.6.3 執行緒取消

- 預設取消型態是延遲取消
  - 取消只發生在一個執行緒抵達**取消點** (cancellation point)
  - pthread_testcancel() 函數
  - **清除處理器** (cleanup handler) 的函數會被呼叫
- 在 Linux 系統使用 Pthreads API 取消執行緒是經由信號完成的

4.6.4 執行緒的局部儲存

- 每一個執行緒可能需要某些資料的自我複製
  - 執行緒區域儲存器 (thread-local storage, TLS)
- TLS 和區域變數
  - 區域變數只有在一個單一函數被呼叫期間才看得到，而 TLS 資料則在函數呼叫前後都看得到
  - 開發人員無法控制執行緒建立的過程時，則需要一種替代方法
    - 例如使用諸如執行緒池之類的隱式技術時
- TLS 類似靜態 (static) 資料
  - TLS 資料對每一個執行緒是唯一的

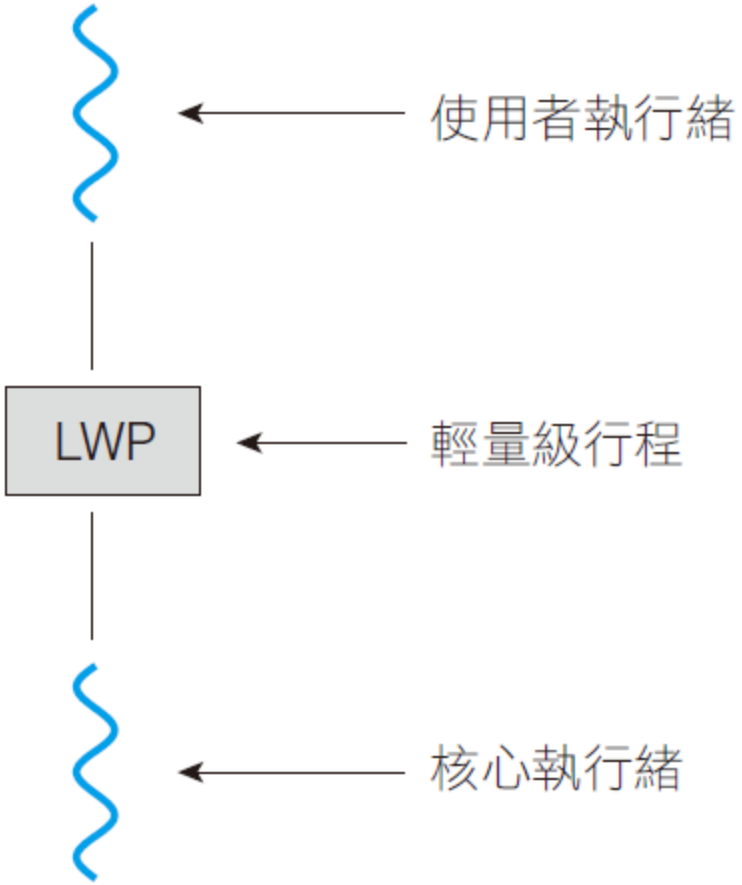
4.6.5 排班器活化作用

- 在核心與多執行緒程式之間的通信，這個通信可能是 4.3.3 節討論的多對多和二層模式所要求
- 會放一個中間資料結構在使用者執行緒與核心執行緒之間
  - 這個資料結構——通常稱為**輕量級行程** (lightweight process, LWP)
    - LWP 看起來像是一個虛擬處理器，應用程式可以排班一個使用者執行緒在其上執行
    - 每一個 LWP 連到一個核心執行緒，然後作業系統排班這些核心執行緒在實體處理器上執行

4.6.5 排班器活化作用

- 核心以一組虛擬處理器 (LWPs) 提供一個應用程式，並且應用程式在可用的虛擬處理器上排班執行
- 核心必須通知應用程式一些事件，這種流程稱為**向上呼叫** (upcall)
  - 由執行緒程式庫用一個**向上呼叫處理程式** (upcall handler) 處理
  - 而且向上呼叫處理程式必須在虛擬處理器上執行

圖 4.20　輕量級行程 (LWP)



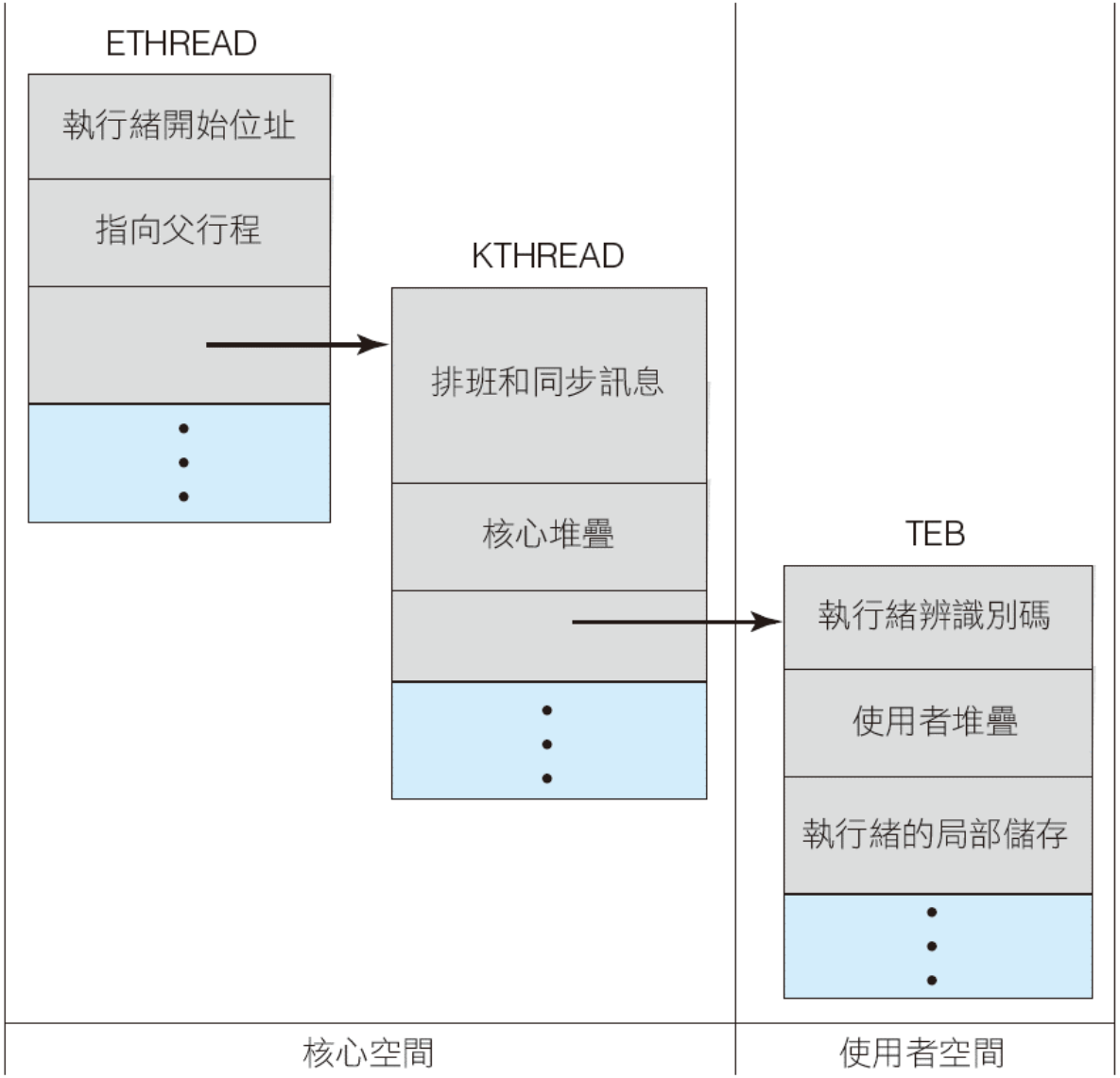
4.7 作業系統範例

- **Windows 執行緒**
  - Windows 使用一對一映射，其中每一個使用者層級的執行緒映射到一個相關的核心執行緒
  - 一個行程的一般元件包括：
    - 唯一識別此執行緒的執行緒 ID
    - 表示處理器狀態的暫存器組
    - 程式計數器
    - 一個使用者堆疊和一個核心堆疊
    - 被不同的執行時程式庫和動態鏈結函式庫 (dynamic link library, DLL) 所使用的一個私有儲存區域
  - 暫存器組、堆疊和私有儲存區域通稱為執行緒的**內容** (context)

4.7 作業系統範例

- 執行緒的主要資料結構包括：
  - ETHREAD —— 執行的執行緒區塊
    - ETHREAD 同時包含一個指向相對應 KTHREAD 的指標
  - KTHREAD —— 核心執行緒區塊
    - 包括此執行緒的排班和同步資訊 KTHREAD
    - KTHREAD 包括核心堆疊(當此執行緒在核心模式執行時使用)，和一個指向 TEB 的指標
  - TEB —— 執行緒環境區塊
    - 在其它欄位間包含執行緒辨識碼，一個使用者模式的堆疊和一個執行緒特有的儲存陣列

圖 4.21 Windows 執行緒的資料結構



4.7 作業系統範例

- Linux 執行緒
  - clone() 系統呼叫來產生執行緒的能力
    - Linux 通常使用任務這一個名詞——而不是行程或執行緒
  - 當呼叫 clone() 時，它被傳遞一組旗標來決定有多少共用發生在父任務與子任務之間
  - 一個唯一的核心資料結構 (struct task_struct) 存在系統中每一個任務

圖 4.22　當 clone() 被呼叫時一些被傳遞的旗標

旗標	意義
CLONE_FS	共用檔案系統訊息
CLONE_VM	共用相同記憶體空間
CLONE_SIGHAND	共用信號處理程式
CLONE_FILES	共用一組的開啟檔案