Chapter 4: 多執行緒 Multithreaded Programming

Chapter 4: 多執行緒

- 概　論
- 多核心程式撰寫
- 多執行緒模式
- 執行緒程式庫
- 隱式執行緒
- 執行緒的事項
- 作業系統範例

章節目標(Objectives)

- 介紹執行緒的觀念—CPU使用的基本單元，構成多執行緒電腦系統的基礎
- 討論Pthreads、Windows、和Java執行緒程式庫的API
- 探討提供隱含執行緒的一些策略
- 檢查關於多執行緒程式的議題
- Windows和Linux作業系統對執行緒的支援

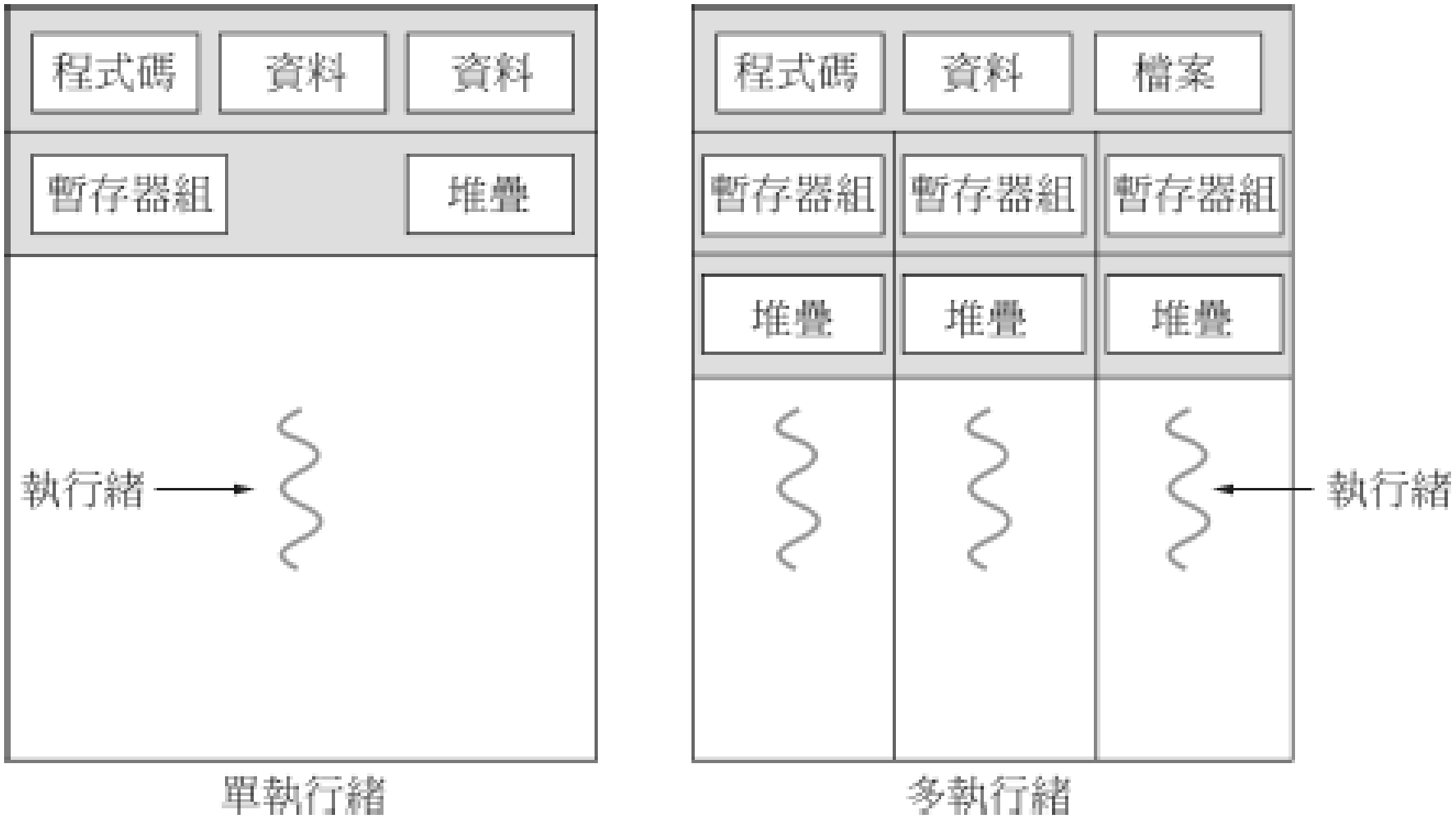
動機(Motivation)

- 目前多數的應用程式都是多執行緒
  - A web browser a displaying thread, and network data retrieving thread.
  - A word processor: a displaying thread, keystrokes reading thread, and a spelling and grammar checking thread
  - A web server: a separate thread that would listen for client requests
- 執行緒(Thread)-輕量級行程
  - CPU配置的基本單位
  - 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

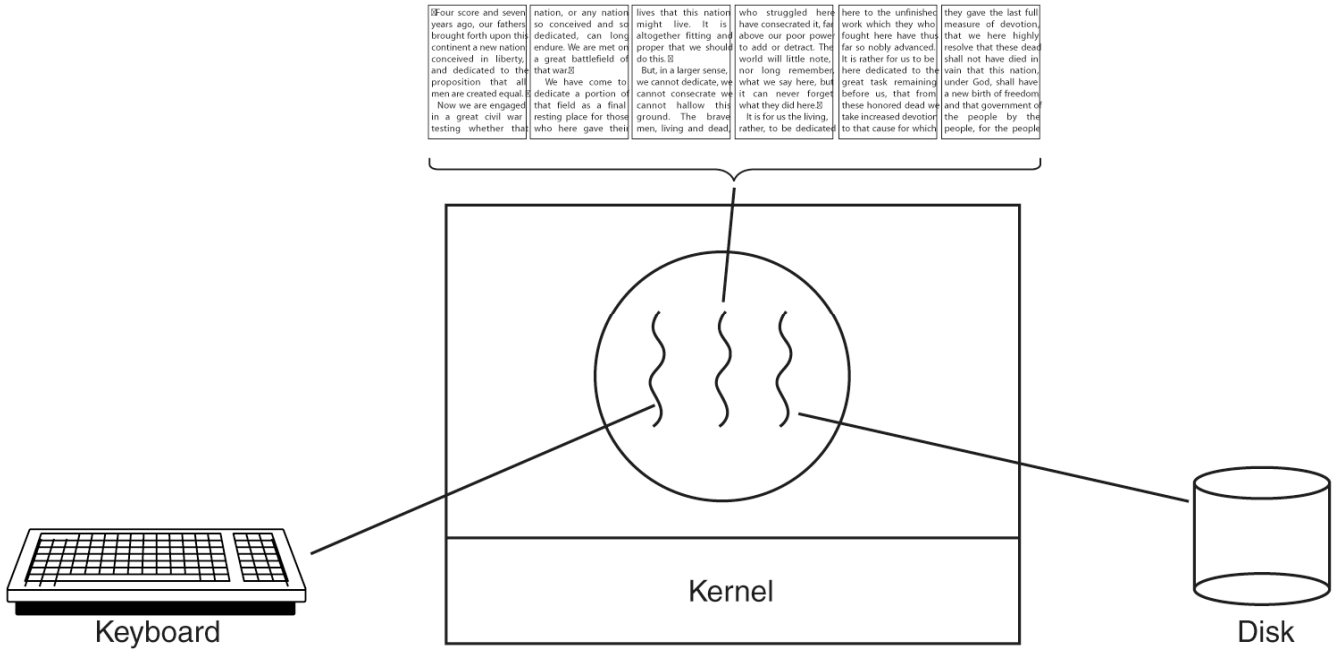
動機(Motivation)

- 傳統的行程相當於是只有一個執行緒的單執行緒行程
  - 單執行緒行程的缺點在於當它同時接收到多項要求的時候，如果不是採取循序執行的方式，就是必須產生多個子行程，以提供較佳的互動
  - 這樣的做法需要頻繁地建立大量的行程與執行內文切換，不僅耗費系統處理時間，而且每個行程都要佔用一塊記憶體空間
- 應用程式中有多項任務時可以用不同的執行緒製作
  - 更新顯示畫面
  - 擷取資料
  - 拼字檢查
  - 回應網路的要求
- 行程的產生是重量級，而執行緒是輕量級
  - 可以簡化程式碼，增進效率
- 核心通常是多執行緒

單執行緒和多執行緒的行程



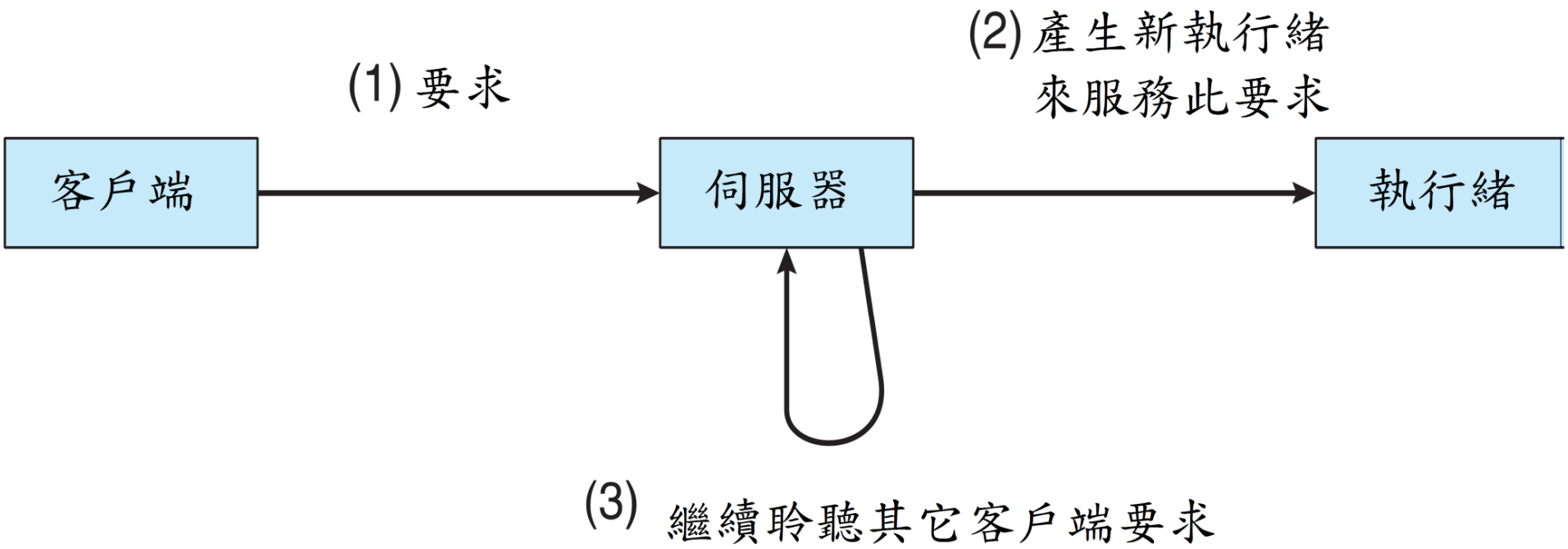
Multithread Examples



A word processor with three threads

- A displaying thread
- A keystrokes reading thread
- An auto file-saving thread

多執行緒伺服器架構



行程與執行緒的主要差異

- 位址空間：行程間的位址空間是相互獨立的，而同一行程的執行緒間則是共用相同的位址空間。
- 通訊方式：行程間的通訊必須利用作業系統所提供的機制進行；而執行緒間因為共享相同的位址空間，可以透過直接讀寫其全域資料來進行溝通。
- 內文切換：同一行程中的執行緒間進行內文切換的時間遠小於行程間的內文切換。

多執行緒優點

- Responsiveness(快速應答)：將一個互動程式採取多執行緒做法，可使得程式的某一部份被暫停或正在執行冗長運算時，仍然可以繼續執行，可以提高對使用者的應答速度。
- Resource Sharing(資源共享)：同一行程的執行緒共享所屬行程的記憶體及資源，會比不同行程間須透過程式設計者利用共用記憶體和訊息等方法來共享資源顯得更容易。
- Economy(經濟)：因為行程必須配置獨立的記憶體及資源，但執行緒則共享所屬行程的記憶體及資源，因此執行緒的產生比行程的產生代價較低，且執行緒的內文切換(context-switch)也較行程的內文切換簡便許多，因此可以大幅降低系統的負擔。
- Scalability(可擴展性)：在多處理器的架構下，一個多執行緒行程的不同執行緒可以在不同處理器上平行執行，充分發揮多處理器的效益。

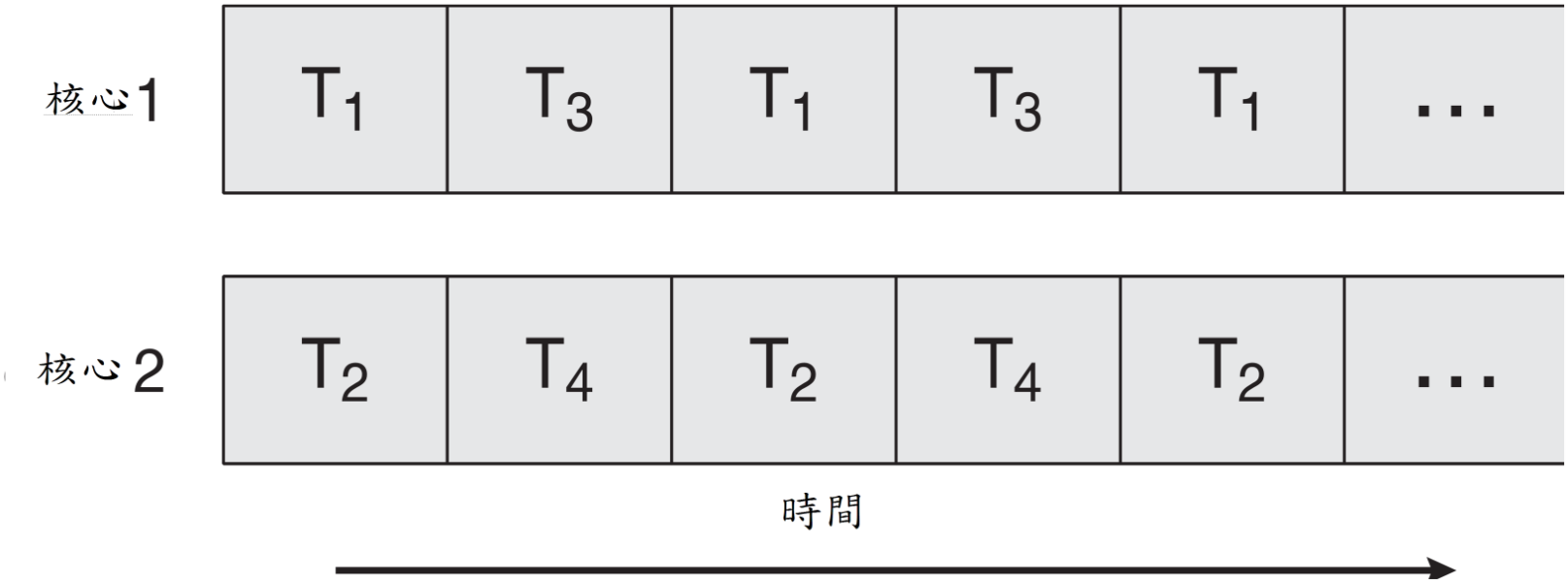
多核心程式設計(Multicore Programming)

- 多核心(multicore)或多處理器(multiprocessor)系統對程式人員造成負擔，挑戰包括了:
  - 任務確認(Identifying task):找出應用程式中可獨立且同時執行的任務
  - 工作量平均(Balance)
  - 資料分割(Data splitting):對每個任務欲處理的資料進行切割給不同核心
  - 資料相依(Data dependency):檢查任務間是否有資料相依關係，若有則須考慮任務間的同步
  - 測試與偵錯(Testing and debugging):一個多執行緒程式同時在多核心上執行時，會有許多可能的執行順序，使得程式的測試與偵錯變得更複雜。
- 平行(Parallelism)表示系統可以同時執行一項以上的工作
- 並行(Concurrency)支援一個以上的任務，並讓每個任務有進展
  - 單處理器 / 核心, 排班器提供並行

Concurrency(並行) vs. Parallelism(平行)

- 在單核心系統的並行執行 ：

- 在多核心系統的平行執行:



平行(Parallelism)的類型

- 資料平行(data parallelism) –每一個核心對分配到的部分資料進行相同的運算處理
  - 如陣列A[0]+…+A[99]加總：$core_0$ 負責A[0]+…+A[49]加總， $core_1$ 負責A[50]+…+A[99]加總。
- 任務平行(task parallelism) –分配執行緒到多個運算核心，每一個執行緒針對相同資料或不同資料執行不同的任務
  - 如$core_0$ 負責陣列A 平均數計算任務， $core_1$負責陣列A 最大值計算任務， $core_2$ 負責陣列B 平均數計算任務。
- 當執行緒的個數增加時，硬體架構對於執行緒的支援也增加
  - CPU有更多的核心和硬體執行緒的支援
  - 考慮 Oracle SPARC T4 有 8個核心,而每個核心有8個硬體執行緒

阿姆達爾定律(Amdahl's Law)

- 一個同時有依序執行單元(serial component)和平行執行單元(parallel component)的應用程式，在加入額外運算核心後所獲得的加速效果(speedup)

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

  - S是依序執行單元所佔比例
  - N表處理核心數量
- 當 S=75%及N=2的情形下，最快可加速到1.6倍
- 當N 趨近無限大時，最快可加速到1/S倍

使用者執行緒(User Threads)和核心執行緒(Kernel Threads)

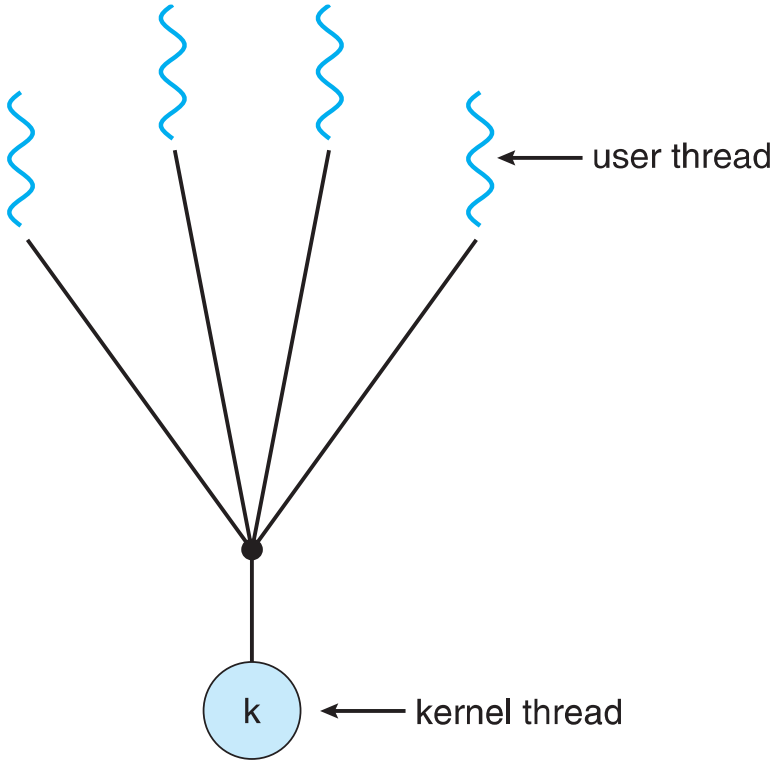
- User threads -由使用者層次的執行緒程式庫(threads library)來支援與管理每個執行緒的狀態與資訊
- 三種主要執行緒程式庫：
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel threads -由作業系統核心來支援與管理每個執行緒的狀態與資訊
- 範例 – 幾乎所有一般用途作業系統都提供核心執行緒：
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

多執行緒模式(Multithreading Models)

- 使用者執行緒與核心執行緒的三種對應關係
  - 多對一(Many-to-One)
  - 一對一(One-to-One)
  - 多對多(Many-to-Many)

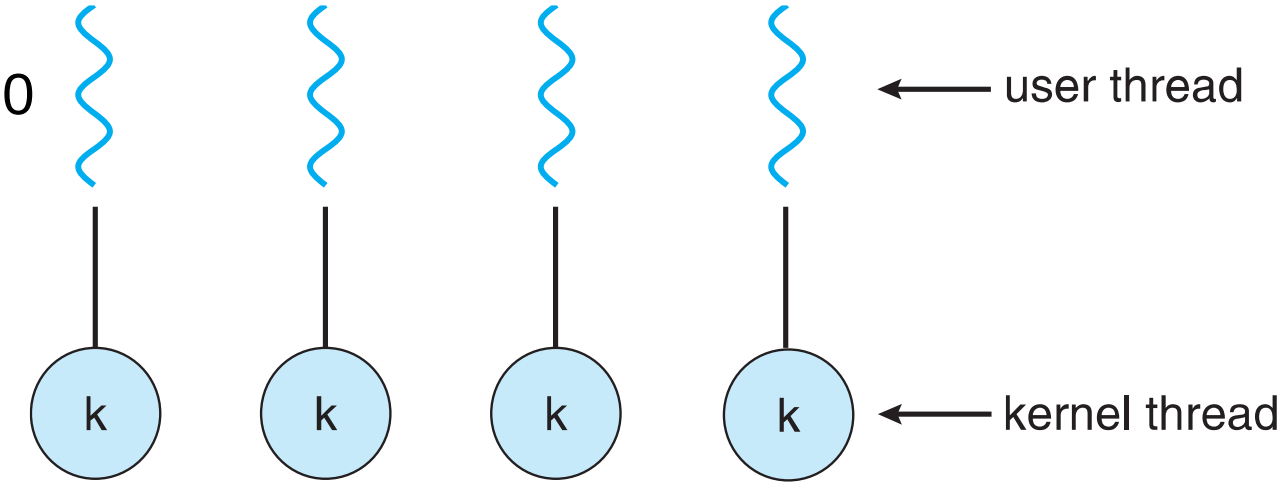
Many-to-One Model

- 許多個使用者層次執行緒對應到單一個核心執行緒
- 當一執行緒進行一需等待的系統呼叫(blocking system call)將造成整個行程的等待
- 因為一次只有一個執行緒可以在核心，同一行程的執行緒不能在多核心系統上平行地執行
- 目前很少有系統使用這個模式
- 範例：
  - Solaris Green Threads
  - GNU Portable Threads



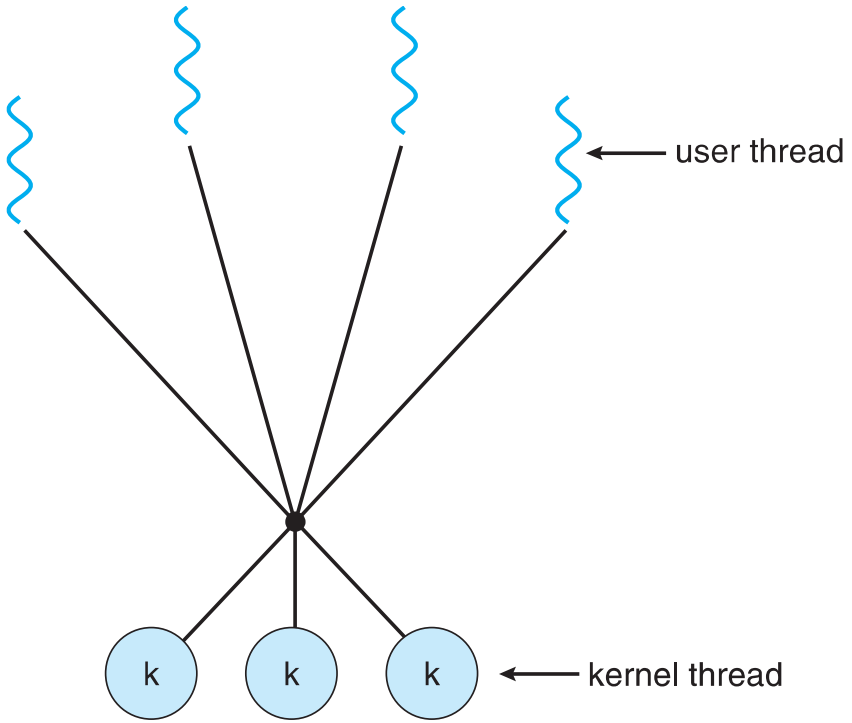
One-to-One Model

- 每一個使用者執行緒對應到一個核心執行緒
- 產生一個使用者執行緒時就會產生相對應的核心執行緒
- 提供了比多對一模式更多的並行功能(concurrency)
  - 當一執行緒進行一需等待的系統呼叫(blocking system call)時，其他執行緒仍可執行不會造成整個行程的等待。
  - 允許多執行緒在多處理器下平行執行
- 因為產生核心執行緒的額外負擔，每一個行程的執行緒個數須受到限制
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



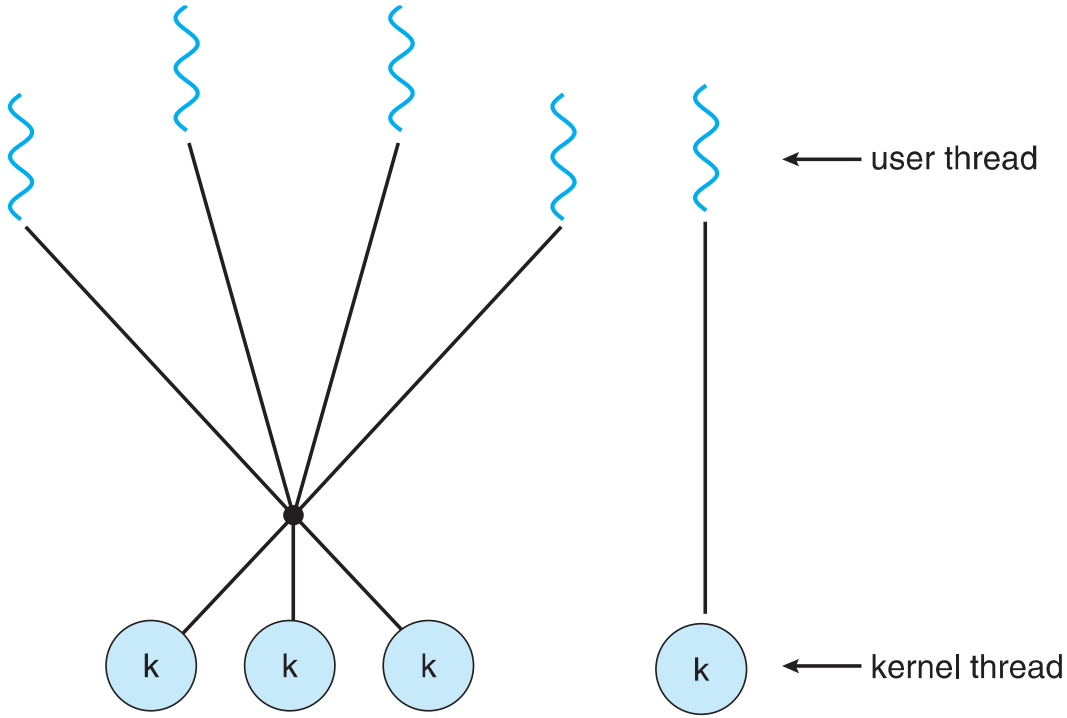
Many-to-Many Model

- 允許多個使用者執行緒對應到多個核心執行緒
- 允許作業系統產生足夠數目的核心執行緒
- 允許核心執行緒在多處理器下平行執行
- 當一執行緒進行一需等待的系統呼叫(blocking system call)時，其他執行緒仍可執行不會造成整個行程的等待。
- Solaris 第 9 版前
- Windows NT/2000 with the ThreadFiber package



雙層模式(Two-level Model)

- 和多對多模式相似，但允許一個使用者執行緒對應一核心執行緒
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 和早期版



執行緒程式庫(Thread Libraries)

- 執行緒程式庫提供程式設計者API來產生和管理執行緒
- 二種主要實作方法
  - 使用者空間(user space)程式庫
  - 作業系統支援的核心層次程式庫
- 三個常用執行緒程式庫：
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Pthreads

- 可以由使用者層次或核心層次提供
- 針對執行緒的產生和同步而設計的POSIX(IEEE 1003.1c)標準API
  - 定義執行緒的規格，而非製作方式

- 在UNIX 作業系統很普遍(Solaris, Linux, Mac OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.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 */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   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 begin control 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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Pthreads Code for Joining 10 Threads

```
#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);
```

Figure 4.10 Pthread code for joining ten threads.

Win32 API Multithreaded C Program

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

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

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

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Win32 API Multithreaded C Program (Cont.)

```
/* 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 */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

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

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

Java執行緒

- Java執行緒由JVM管理
- 通常使用底層OS提供的執行緒模式製作
- Java執行緒可以使用以下方是產生:
  - 繼承自Thread類別
  - 製作 Runnable介面的類別

```
public interface Runnable
{
    public abstract void run();
}
```

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

隱式執行緒(Implicit Threading)

- 當執行緒數目增加，程式發展人員自己處理執行緒時，程式的正確性變得更難
- 隱式執行緒的產生和管理由編譯器和執行階段程式庫完成，而不是程式發展人員
- 隱式執行緒的三種方法
  - 執行緒池(Thread Pools)
  - OpenMP
  - Grand Central Dispatch
- 其他方法包括了Microsoft Threading Building Blocks (TBB), java.util.concurrent套件

Thread Pools

- 在行程開始執行時就產生一些執行緒，然後放入執行緒池中等待工作
- 當伺服器收到請求時，只要到池中喚醒一個執行緒去處理就可以了
- 執行緒完成工作之後，會再回到池中等待
- 優點:
  - 使用現存的執行緒服務一項要求時，會比產生新執行緒快
  - 應用程式的執行緒數目受限於執行緒池的執行緒個數
  - 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
- Windows API 支援執行緒池:

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

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions – blocks of code that can run in parallel

- #pragma omp parallel
- Create as many threads as there are cores

```
#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;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” - ^{ printf("I am a block"); }
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main queue
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
  (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

執行緒相關議題(Threading Issues)

- fork( ) 和exec( )系統呼叫
- 信號處理
  - 同步(Synchronous)和非同步(Asynchronous)
- 執行緒取消
  - 非同步(Asynchronous)或延遲(deferred)
- 執行緒的局部儲存
- 排班程式活化作用(Scheduler Activations)

Operating System Examples

- Windows XP Threads
- Linux Thread

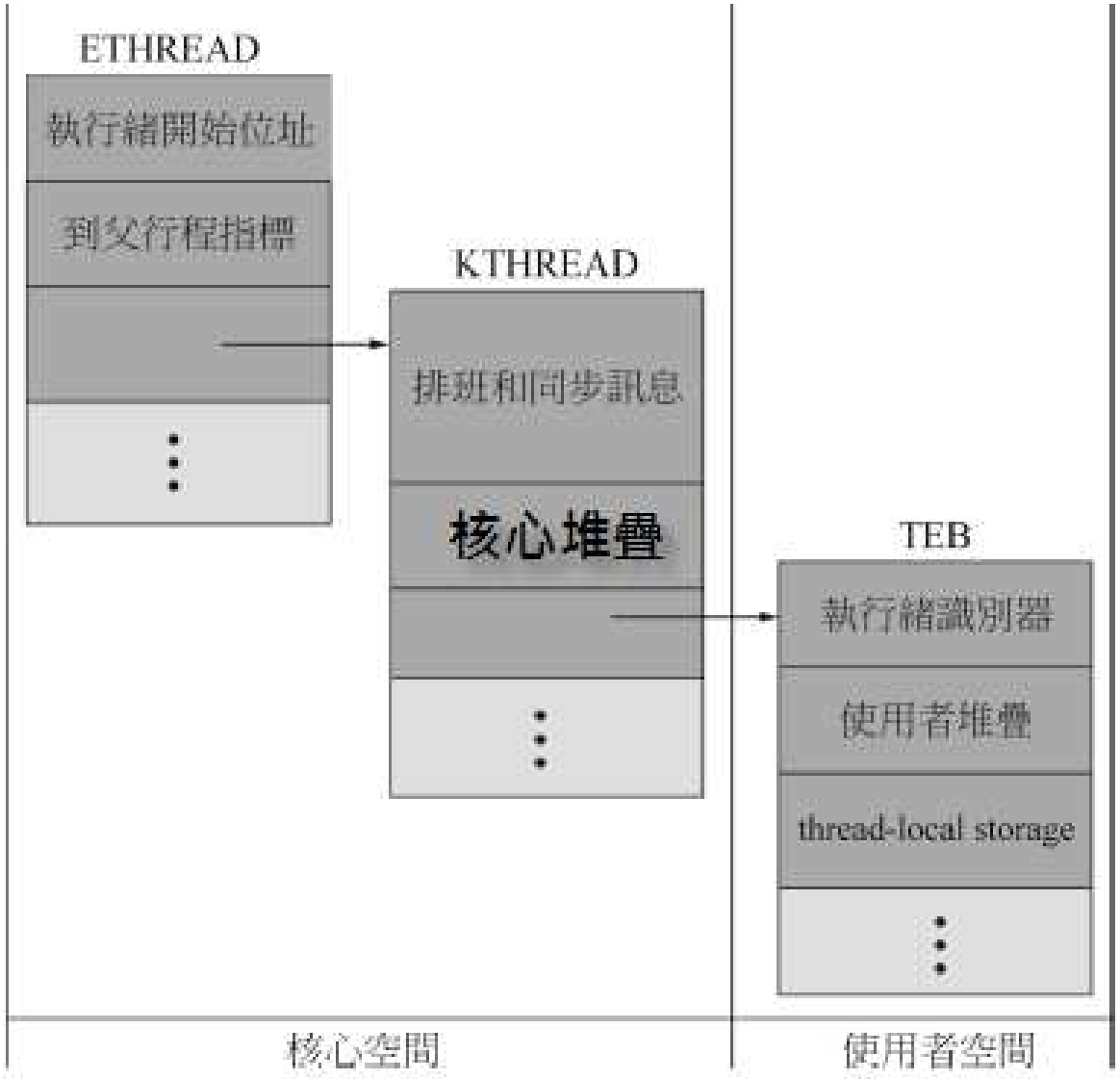
Windows Threads

- 透過Windows API
- 一對一對應，核心層次
- 每一個執行緒包含了
  - 執行緒id
  - 表示處理器狀態的暫存器組
  - 一個使用者堆疊給執行緒在使用者模式執行時使用，另一個核心堆疊給執行緒在核心模式執行時使用
  - 執行時程式庫和動態連結程式庫(DLLs)所使用的私有資料儲存區域
- 暫存器組、堆疊和私有儲存區域通稱為執行緒的內容(context)

Windows Threads

- 執行緒的主要資料結構包括了:
  - ETHREAD (executive thread block) –包括一個指向此執行緒所屬行程的指標、一個指向相對應KTHREAD的指標
  - KTHREAD (kernel thread block) – 核心空間的排班和同步資訊、核心模式堆疊、指向TEB的指標
  - TEB (thread environment block) –使用者空間的資料結構包含了執行緒id、使用者模式的堆疊、執行緒局部儲存

Windows XP Threads Data Structures



Linux Threads

- Linux 稱為任務(task)，而不是行程或執行緒。
- 執行緒的產生是經由系統呼叫clone()
- clone()讓子任務共用父任務(行程)的地址空間
  - 旗標控制行為

旗標	意義
CLONE_FS	共用檔案系統訊息
CONE_VM	共用相同記憶體空間
CLONE_SIGHAND	共用訊號處理程式
CLONE_FILES	共用一組的開啓檔案

- struct task_struct 指向行程資料結構 (共用或唯一)