並行計算

EP05: 執行線（Thread）

在學習時你想你了解了，
到要寫下來時更加確定，
要教他人時就會更肯定，
但只有在寫程式時才能真正體認自己充分了解。

Alan J. Perlis

本集內容

- 什麼是執行線（thread）？
- 使用Thread的好處
- 使用人（User）Thread和系統（Kernel）Thread
- 複線作業（Multithreading）模型
- 多核心程式寫作基本觀念
- 取消Thread執行（Thread Cancellation）
- Thread專用和Thread安全
- 協同函數（Coroutine）和纖維線（Fiber）
- 簡單的歷史回顧
- 再說本質上的順序性（Inherently Sequential）

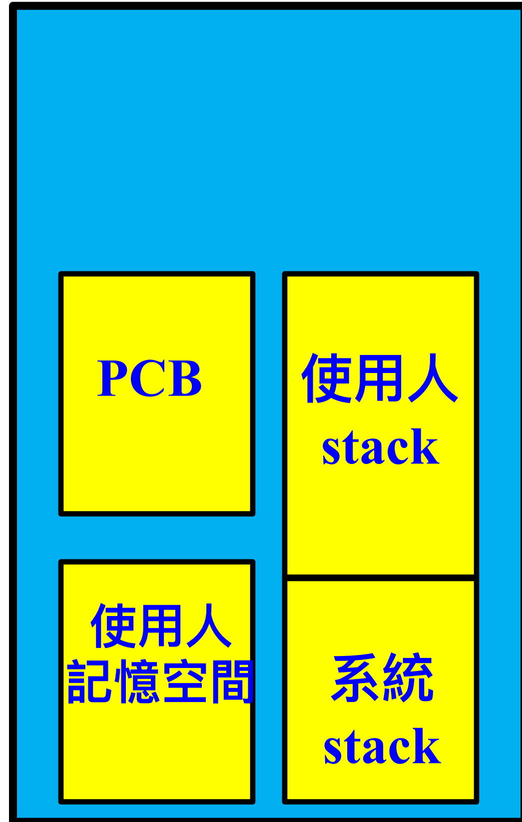
請看影片的說明，那兒有取得投影片和程式的連結

什麼是Thread（執行線）

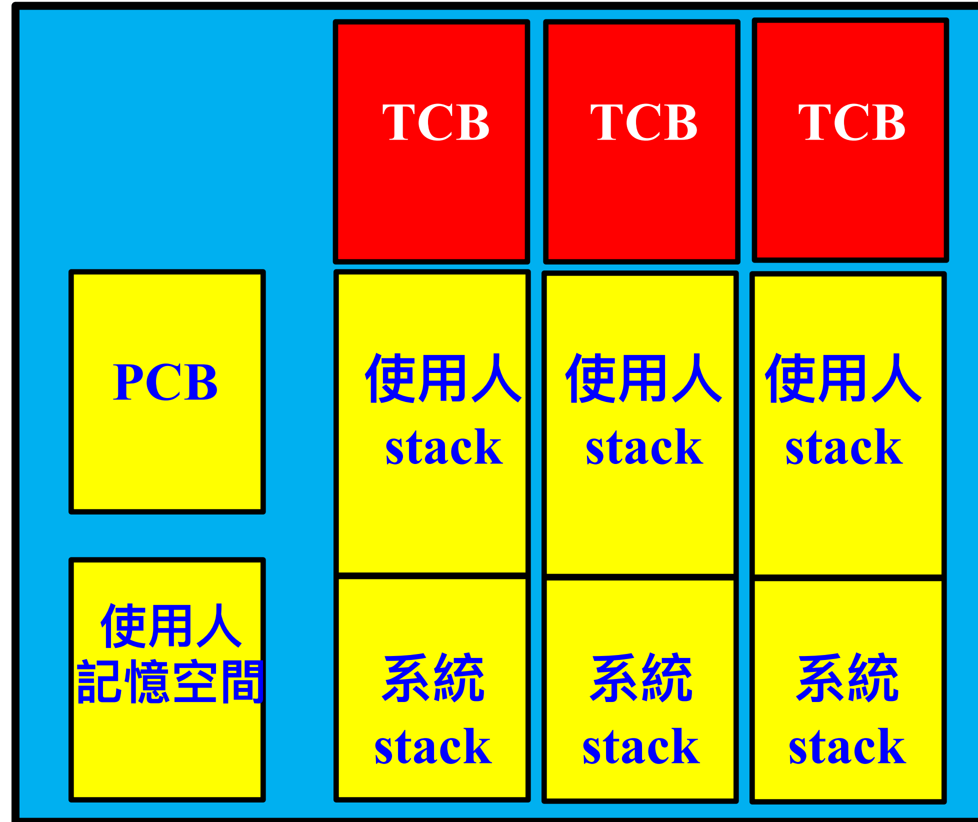
- Thread（執行線）是一個基本的可以在CPU/Core中執行、由某個調度程式（Scheduler）管理的執行單位。這個調度程式可能是作業系統中的CPU調度程式（CPU Scheduler）。
- Thread由process產生。
- 一個Thread擁有thread ID（TID）、program counter、一組暫存器和stack。這一點和process相同。
- 但是，因為thread是由process產生，所以由某一個process產生的所有thread共享該process擁有的code和data區域，以及其它作業系統的資源（譬如記憶體、檔案等等）。
- 因為thread需要的系統資源比較低，thread有時會稱為輕量process（Light Weight Process、LWP），而process則叫做重量process（Heavy Weight Process、HWP）。

單線和複線Process

單線（Single-Threaded）Process



複線（Multithreaded）Process



使用Thread的好處

- **反應能力**：如果一條**thread**（基於任何理由）不能執行了，包含它的**process**的其它**thread**仍然可能可以執行。
- **共享資源**：在**約定**的情況下，一個**process**產生的所有**thread**會**共用**許多系統資源（譬如檔案、記憶體等等）。
- **經濟效益**：因為一個**process**使用很多系統資源，產生或摧毀一個**process**、為它配置記憶體和資源、甚至於**切換執行環境**（**context switch**）都相當費時；但**thread**使用資源少，因此從事這些作業也相對省時間。
- **使用多處理機架構**：在多處理機系統中，若干個處理機可以同時執行同一個**process**中的若干**thread**（包含該**process**本身）。更重要的是，**不需要修改程式**就可以做到。

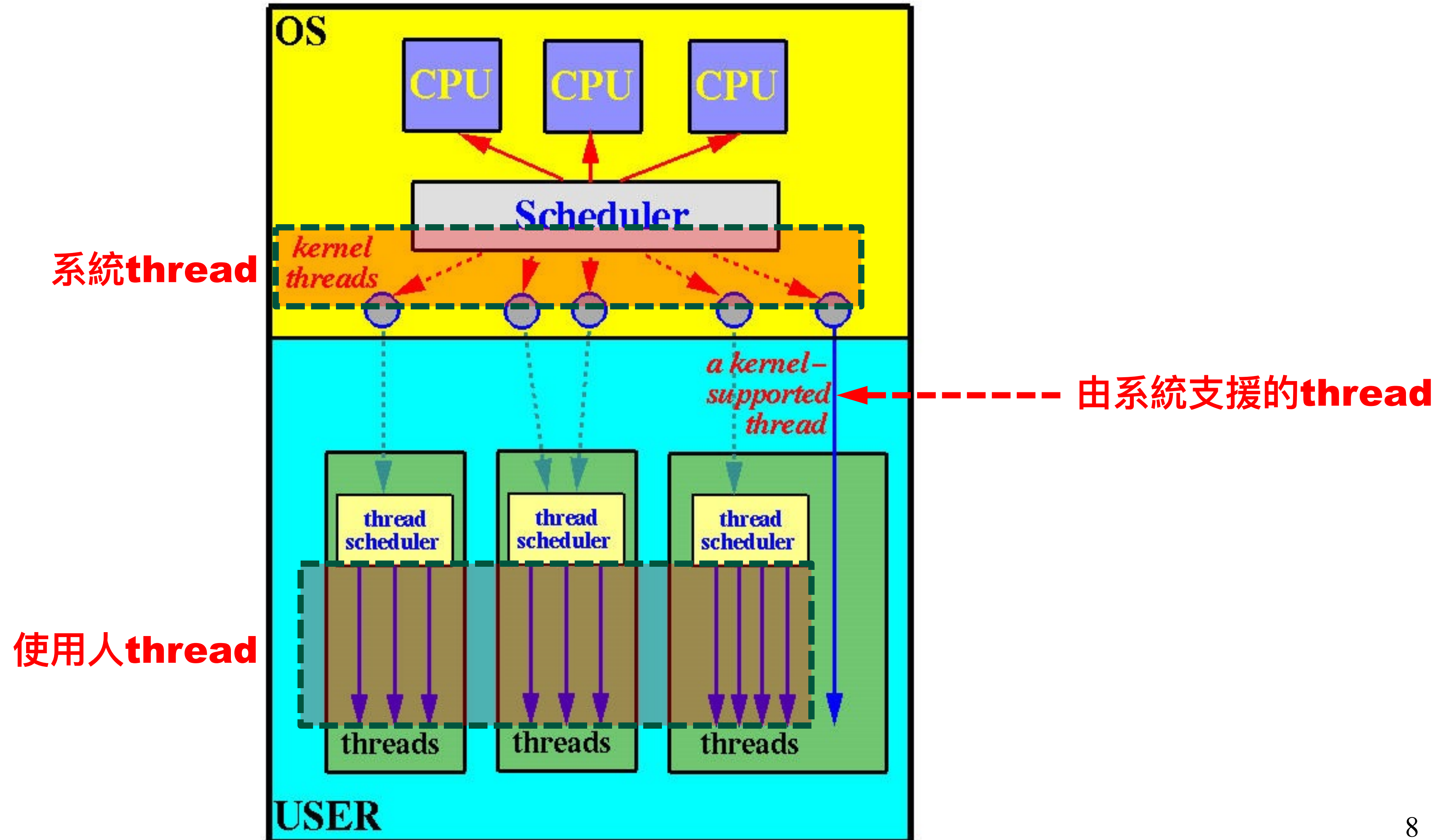
使用人Thread和系統Thread: 1/3

- 使用人Thread
  - ✓ 在使用人的層面支援使用人thread（user thread），系統不知道使用人thread的存在。
  - ✓ 一般而言，使用人thread是由程式庫支援，可以產生、停止一個thread；可以和另一個thread匯合；甚至於調度（schedule）使用人thread。
  - ✓ 因為使用人thread不需要系統介入（不需要透過叫用系統）處理，使用人thread效率會比較高。
  - ✓ 然而，因為系統只認得包含了產生這些使用人thread的process，一旦一個process被暫停執行，它產生的使用人thread也都不能執行。換言之，當產生這些使用人thread的process失去了CPU使用權，它產生的使用人thread也就無CPU可用。

使用人Thread和系統Thread: 2/3

- 系統Thread
  - ✓作業系統支援系統thread。系統負責產生、停止一個thread、和另一個thread匯合、也負責調度（schedule）系統thread。
  - ✓從事上述工作時都得叫用系統、在OS內完成，因為這些額外的系統負擔，系統thread的效率（和使用人thread相比）是較低的。但因為thread所需資源較少，所以系統thread的效率仍然比process高。
  - ✓另一方面，當一個系統thread由於某種原因不能執行時，同一個process的其它系統thread仍然有機會執行，CPU調度程式（scheduler）可以挑另一個系統thread執行。
  - ✓在多處理機環境下，作業系統可能會把這些系統thread安排到不同的處理機上執行。

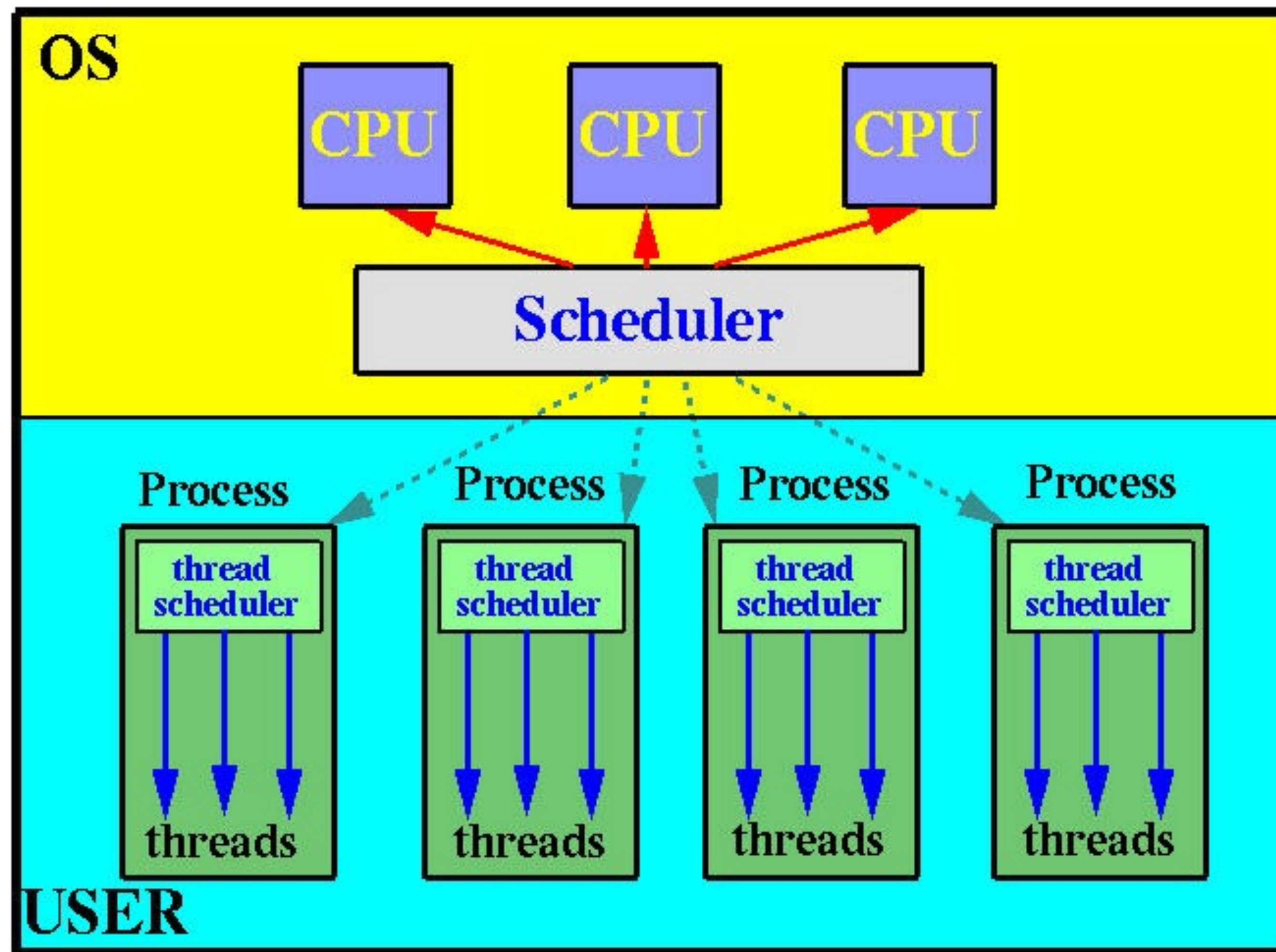
使用人Thread和系統Thread: 3/3



複線作業（Multithreading）模型

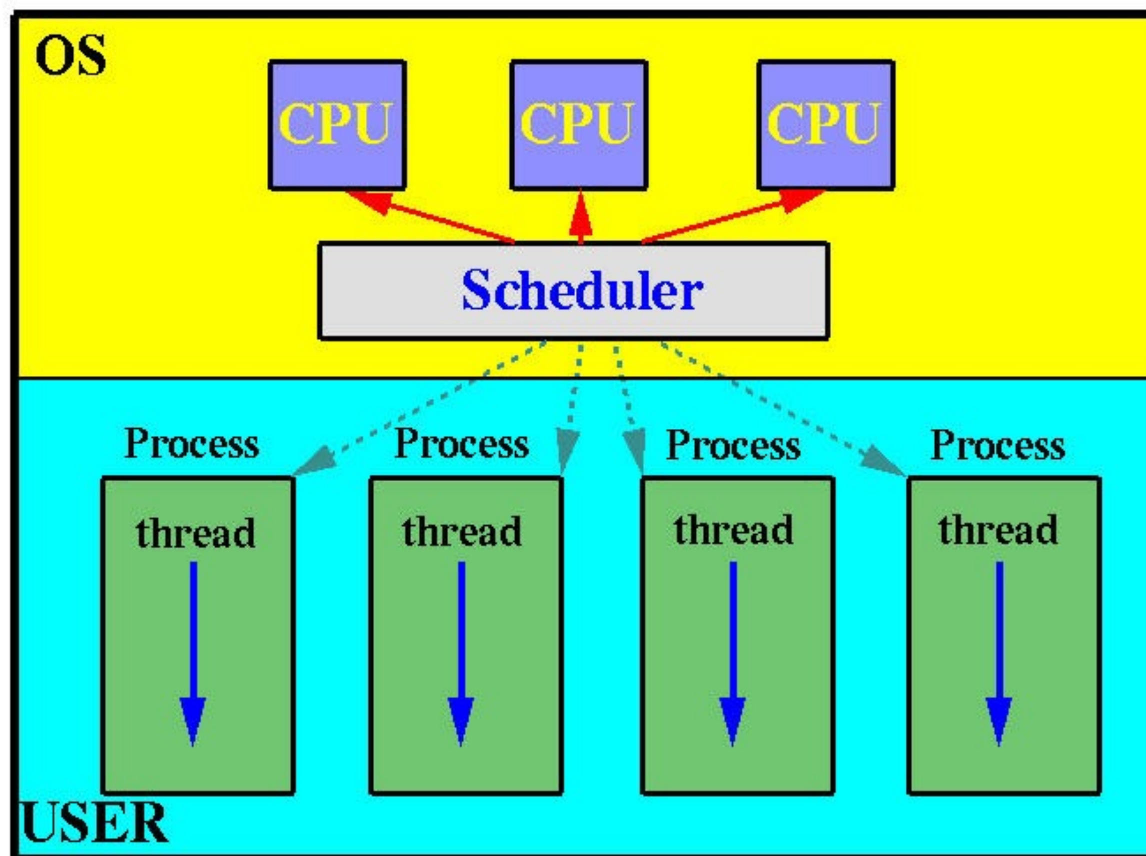
- 不同的系統可能會用不同的方式支援thread，下面是三種常見的方式：
  1. 多-對-1 (Many-to-One):一個系統thread（或process）給若干個thread使用，所以這是個使用人thread模型。
  2. 1-對-1 (One-to-One): 每一條使用人thread都對應著一條系統thread；一些UNIX/Linux系統和Widows用此方式。
  3. 多-對-多(Many-to-Many): 若干條使用人thread對應著若干條系統thread。

多-對-1（Many-to-One）模型



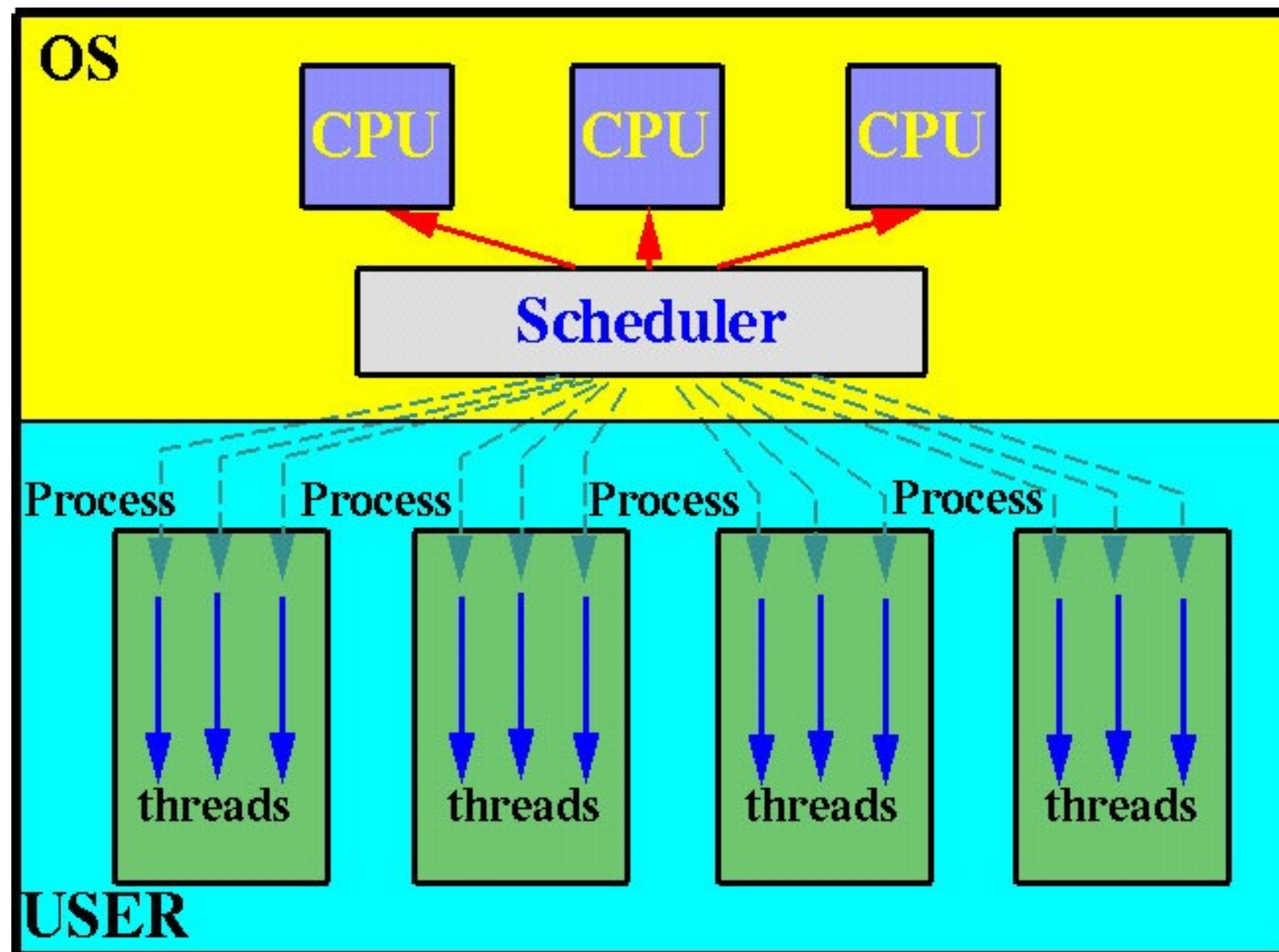
每一個process都有若干使用人thread，它們都對應到一條（給process用的）系統thread。
若process被暫停（失去CPU使用權），在它之下的所有使用人thread也都不能執行。

1-對-1（One-to-One）模型: 1/2
一個極端的個案：傳統Unix



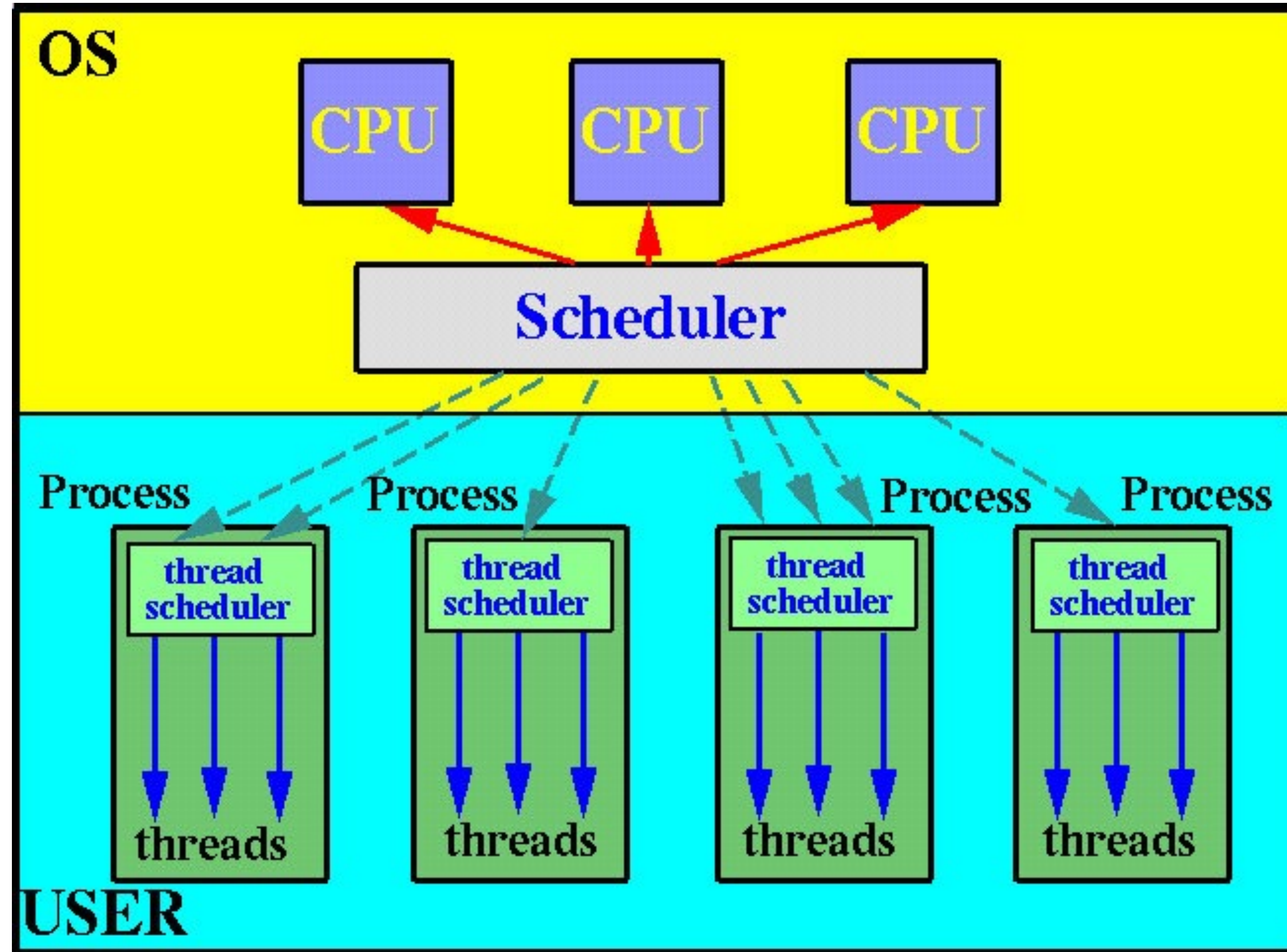
每一個process都只有一條使用人thread，它對應到一條系統thread。

1-對-1（One-to-One）模型: 2/2



每一個process都有若干條使用人thread，每一條使用人thread都對應著一條系統thread。若一條系統thread不能執行，它對應的使用人thread就不能執行。

多-對-多（Many-to-Many）模型



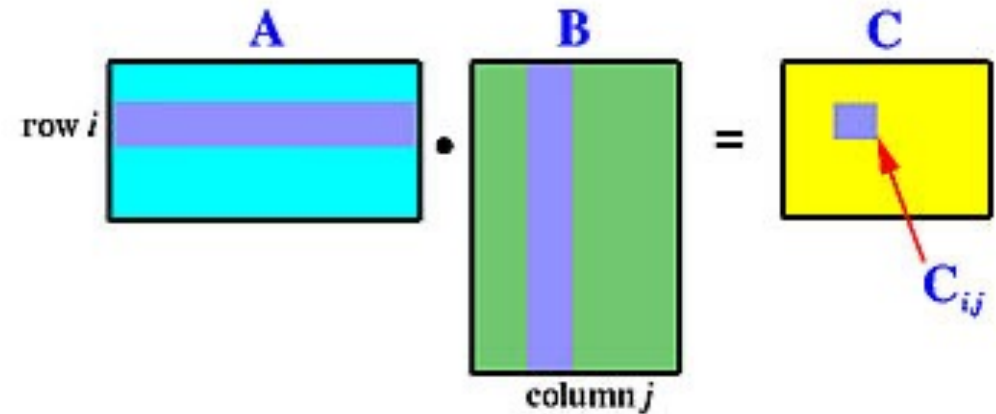
每一個**process**都有若干條**thread**，它們對應到若干條系統**thread**（數目未必相同）。
若一條系統**thread**不能執行，在當時對應著該系統**thread**的使用人**thread**就不能執行。

多核心程式寫作基本觀念: 1/6

- 在單核心（core）的CPU之下，CPU調度程式每次只能選一條thread在唯一的CPU中執行。
- 若一個CPU 有若干個核心（core），CPU 調度程式可以選若干條thread、每一條thread都佔用一個核心執行。
- 支援多核心的系統要比想像中複雜得多，寫作程式時也如此。
- 在多核心CPU 系統下工作的程式必須注意到五項課題：**工作分配、工作量均衡、資料劃分、資料獨立性、測試與偵錯。**

多核心程式寫作基本觀念: 2/6

- **工作分配**：因為每一條**thread**可以在不同的核心下執行，我們得分析問題，**把程式的工作分析出來、使得可以分散到不同核心上並行地執行**。
- 矩陣相乘是一個極好的例子。
- 不幸的是，有一些問題**本質上就是順序**（inherently sequential）的，很難被轉換成平行（或並行）。



$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

對每一個C_{ij}都可以產生一條**thread**

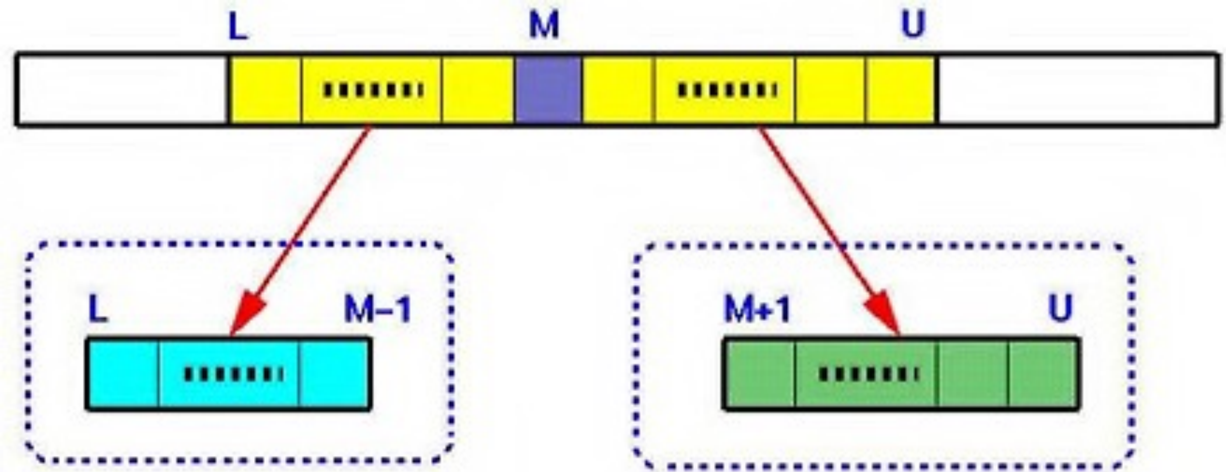
- ✓ 每一個C_{ij}都需要n個乘法
- ✓ C有$n \times n = n^2$個元素，總共需要$O(n^3)$ 個乘法，所以總乘法數目是 $O(n^3)$).。
- ✓ 如果我們用 n^2 條**thread**（每一個C中的元素一條），每一條**thread**只要個 n 乘法算出，所以用複線做法比不用時快 $O(n^2)$ 倍，雖然我們需要n^2條**thread**。

多核心程式寫作基本觀念: 3/6

- 工作量均衡：儘可能讓分配給每一條thread的工作量佔總工作量的貢獻比率大致相同。
- 若一條工作量不顯著的thread常常被執行，它就常常佔用一個核心，於是其它從事重要計算工作的thread能被執行的機會就降低，整個程式的效率也就跟著降低。

多核心程式寫作基本觀念: 4/6

- **資料劃分**：需要處理的資料可以分成若干段落、**每一個段落都可以獨立處理**。
- 矩陣相乘是個好例子。
- 快速排序（Quicksort ）是另一個好例子。當我們用某個中間值把輸入陣列切割後，中間值左（或右）邊的值都比中間值小（或大）；換言之，中間值已經在它最終的位置。於是，左邊和右邊就可以分開並且獨立地排序。
- **程式習題2**中的**並行合併排序**則是第三個好例子。



把a[L..U] 分割成 a[L..M-1] 和 a[M+1..U]之後，我們可以產生兩條thread，一條處理左段、另一條處理右段。每一條thread都只排交給它的那一段、而不會碰到另一段。因此，產生thread的方式就是個二元樹狀結構（binary tree）；程式習題2的合併排序也完全相同。

多核心程式寫作基本觀念: 5/6

- 資料獨立性：要注意到被若干thread共用的資料；譬如說，有沒有若干thread同時更新一個共用的變數等等。事實上，如果若干thread會用到同一筆資料，這筆資料在各thread之間就不再有獨立性，而是某些thread會依賴另一些thread更新該筆資料後的新值。
- 若這種情況發生，就很可能會得到意想不到的結果。因此，各thread之間就得同步（synchronization）、使得在任何時刻都只會有一條thread會更新一個共用變數的值。
- 寫作複線作業程式時，同步（synchronization）是一項有相當難度、也不容易做得好的課題。

多核心程式寫作基本觀念: 6/6

- 測試與偵錯：複線作業程式的行為是動態（dynamic）的，特別是有共享的資源時就更是如此。在這次測試中出現的錯誤在下一次測試中末必會發生；或者是這次測試沒有出現錯誤、但在下一次測試中就跑了出來；或者是每次都出錯、但錯誤卻都不同。
- 有些錯誤在整個程式的生命期都不會出現，也有可能在完全出乎意料的時刻出現。
- 有一些偵錯的課題（譬如競爭狀況 — 同時更新一個共享的資料，或系統死鎖）沒有很高效率的解法，我們能夠做的就是在設計和寫作程式時不要把這些錯誤放進去。
- 所以，測試和偵錯是一項藝術，而且寫程式時就需要很仔細地規劃和設計。**往後我們會陸續討論這些課題。**

取消Thread執行（Thread Cancellation）：1/2

- 取消thread執行指的是中止正在執行的thread；也就是說，在一條thread完成執行之前就中止它。這條被取消的thread叫做目標執行線（target thread）。
- 有兩種取消的方式：
  - ✓非同步取消（asynchronous cancellation）：目標thread立即停止。
  - ✓延遲取消（deferred cancellation）：目標thread固然會停止執行，但會容許它週期性地檢查自己是否應該進入已結束（terminated）狀態，讓系統收回給它的資源。這樣就給目標thread有序地結束的機會。
  - ✓目標thread可以讓自己進入結束狀態的地方叫做取消點（cancellation point）。

取消Thread執行（Thread Cancellation）：2/2

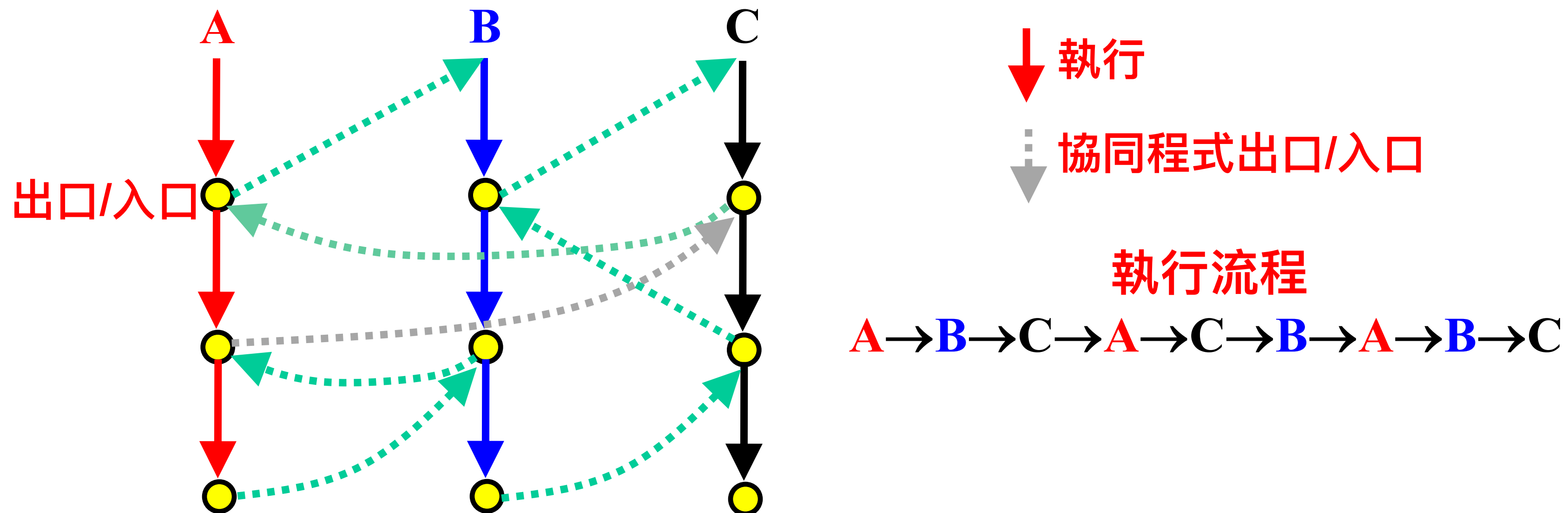
- 在非同步取消下，如果一條目標thread擁有一些系統中的共用資源，目標thread被取消時系統不一定能收回這些資源，因為可能有其它thread正在使用。
- 在延遲取消下，目標thread可以自己決定何時終止執行，因此收回這些資源不是個問題。
- 很多系統對process 和thread使用非同步取消，譬如Unix中用`kill`命令或`kill()` 叫用系統對process執行非同步取消。
- POSIX Threads（Pthreads）支援延遲取消。

Thread專用和Thread安全

- Thread專用（Thread-specific）資訊指的是那些thread用來做自己作業的資訊。
- 對thread專用資料支援不足就可能會發生問題；譬如說，雖然每一條thread 都有自己的stack，但它們都共用產生它們的process的堆積（heap），請回顧第三集中有關一個process記憶空間劃分的說明（第3講第16頁）。
- 如果兩條thread同時叫用malloc()在heap中配置記憶體（第3講第6-7頁）？或者是，兩條thread同時使用printf()顯示資訊（第三講第32-34頁）？
- 一個程式庫的函數若同時被若干thread叫用時都能正常工作，這個程式庫（或函數）就是thread安全的。

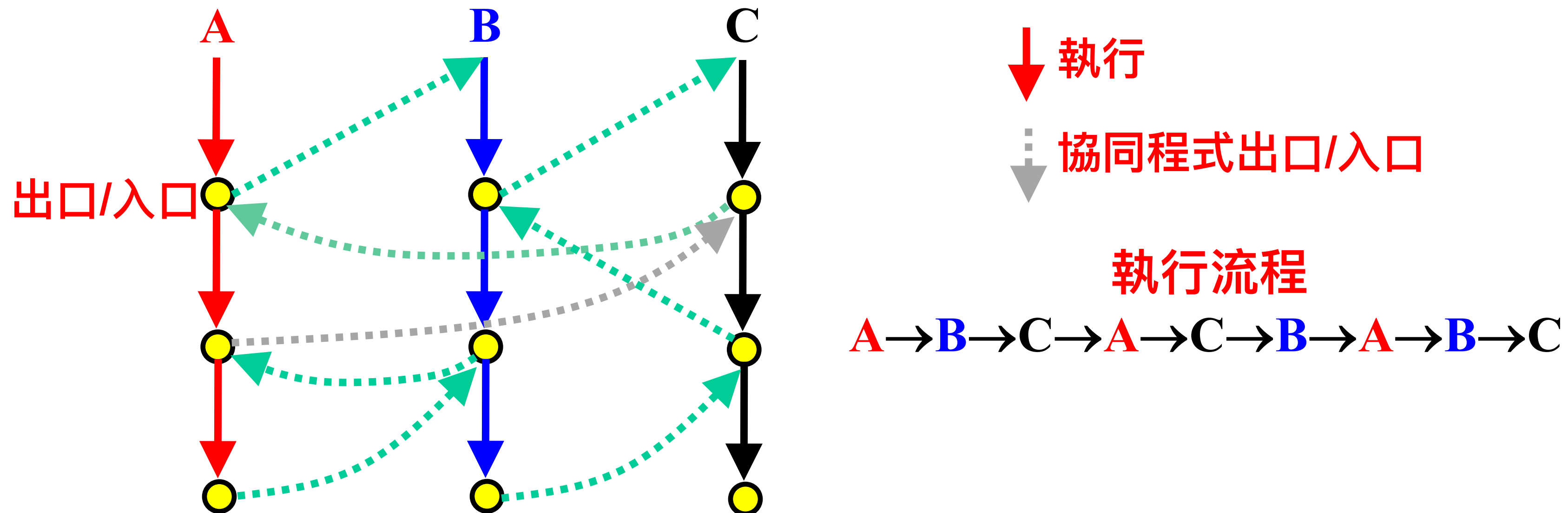
協同函數和纖維線: 1/3

- 一般叫用函數時，函數總是從該函數的第一條指令（入口）開始執行，雖然離開該函數時可能有若干出口。
- 一個協同函數（coroutine）有若干個入口、也有若干個出口。於是「叫用」一個協同函數時，會從上一次離開該函數的出口的下一條指令開始執行。



協同函數和纖維線: 2/3

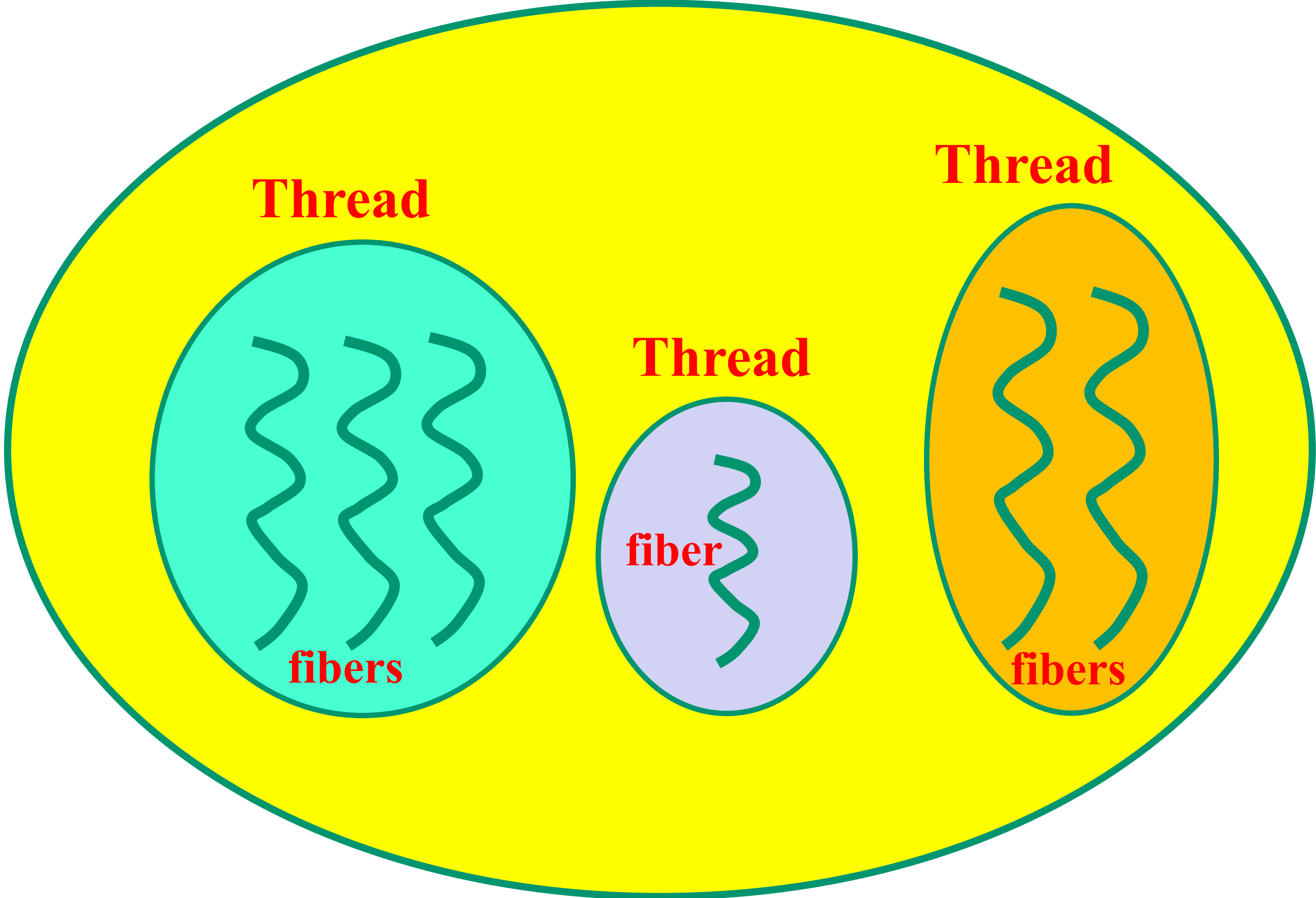
- 從上一頁的出口/入口的方式看來，它是否有一點像調度程式（scheduler）呢？
- 沒錯！一個出口相當於被調度程式暫停，而入口則是被調度程式選中繼續執行，當然這就是從被暫停時的下一道指令開始。
- 協同程式幾乎就是調度程式進行環境切換（context switch）的翻版。



協同函數和纖維線: 3/3

- 執行纖維線（簡稱纖維線 — fiber）就是輕量的thread，這就像thread是一條輕量的process；但是，纖維線必須人工地被應用程式調度，也就是沒有調度纖維線的調度程式。
- 纖維線是由thread產生，它和其他被該thread產生的纖維線共享該thread的資源。
- 一條纖維線在產生時擁有一個stack、一組暫存器、它的資料區域（或是它的局部記憶體）。
- 纖維線透過一種合作（co-operative）方式調度。
- 合作調度（co-operative scheduling）是使用某種「自願」的方式放棄CPU而由下一條纖維線使用。一些有纖維線的語言使用YIELD或類似的函數，讓叫用者讓出CPU使用權。
- 纖維線比執行線（thread）簡單，而且用類似協同程式的方式作業。

Processes ⇒ Threads ⇒ Fibers



簡單的歷史回顧（Process和Thread）：1/4

- **Process**這個詞最遲在**1960**年代中期之前就已經被廣泛使用。
- **Multiprogramming**的出現也差不多是**1960**年代前後，Edsger W. Dijkstra在荷蘭**Technische Hogeschool Eindhoven**（**Eindhoven University of Technology**，簡稱**THE**） 發表了**THE**作業系統（**1965**年），這可能是**Multiprogrammig**這個字正式浮上檯面的時刻，不過學術論文要到**1968**年才刊出。
- 另一方面，**IBM**在**1964**年**4**月發表**IBM System/360**，開始了電腦系統設計和相容性的全新視野。
- **System/360**的作業系統是**IBM System/360 Operating System**（**OS/360**），它在**1966**年**3**月**31**日交給客戶使用。

簡單的歷史回顧（Process和Thread）：2/4

- 事實上，**OS/360**有四套不同版本：
  - ✓ **PCP**（**Primary Control Program**）：這是一套一次只能執行一個程式的**OS**，給那些小機型用的，很快就停用。
  - ✓ **Multiprogramming with a Fixed number of Tasks**（**MFT**）：這是給中階機型用的系統，最多只能支援同時執行**15**個程式。
  - ✓ **Multiprogramming with a Variable number of Tasks**（**MVT**）：這是給大型機用的，它去掉了最多只能執行**15**個程式的限制。
  - ✓ **Disk Operating System/360**（**DOS/360**）：給小機型用的系統。
- 在**IBM**術語中，上述的**task**相當於**process**。
- 但是，**OS/360**和**DOS/360**中也支援**thread**，不過也叫做**task**。

簡單的歷史回顧（Process和Thread）：3/4

- 在OS/360和DOS/360之下，有不少集體指令（macro）可以產生task（也就是thread）並且進行同步（synchronization）；這些一直沿用至今：
  - ✓ 使用人可以用ATTACH產生task、用DETACH把已經終結的task刪除。
  - ✓ IBM用*serially reusable*（序列性可以再使用）資源表示任何時刻只容許一個task可以使用的資源，用今天的話來說就是互斥地使用。
  - ✓ 對這些互斥的資源而言，程式員可以產生一個等待線（queue），在使用一個共同資源前用enqueue（ENQ）集體指令等待使用該資源；使用完後用dequeue（DEQ） 離開該等待線、讓下一個在等待線中的task可以使用該共用資源。
  - ✓ 使用人可以定義事件（event），用WAIT（事件名稱）等待該事件發生（有沒有像叫用系統wait()呢）；若某task工作完成，該task可以用POST宣告某事件已經發生。
  - ✓ Hercules是一個很好用的IBM System/370，System/390和z/Architeture的模擬程式（http://www.hercules-390.org）。目前System/370的幾個作業系統（包含VM/370）和編譯程式都可以在網上找得到，您不妨試試在個人電腦上玩玩這些幾十年前的經典系統。

簡單的歷史回顧（Process和Thread）：4/4

- **OS/360**並沒有用目前教科書方式進行同步（synchronization），但是也用另一方式（ENQ/DEQ 和 WAIT/POST）完成我們期望的結果。
- **IBM PL/I F**（**1966**）中有相當完整的複線（multithreading） 執行功能，但編譯程式卻並沒有公開給客戶使用，因此使用人似乎只能使用集體指令解決這個問題。目前的**IBM z/OS**繼承了幾十年前的傳統，這些功能都有了、而且也相容。
- 這些被產生的task會和其它task共同被系統調度。所以，**OS/360**對thread的支援應該是前述的系統thread。**Thread**這個字是Jerome Haward Saltzer在他的1966年博士論文中提到的（第20頁腳註）、V. Vyssotsky建議的字，但那是指process而不是今天熟知的執行線。

Jerome Howard Saltzer, Traffic Control in a Multiplexed Computer System, MIT PhD Thesis (MAC-TR-30 THESIS), July 1966.

再說Inherently Sequential問題: 1/4

- 在理論計算機科學中，如果輸入個數是n，那麼**P**是所有我們可以用某個變數為n的多項式步驟能夠完成的問題的集合。
- 譬如，搜尋一個已經排序的陣列得用$O(\log_2(n))$個比較（二分搜尋法）、用比較排序會有$O(n\log_2(n))$的演算法（合併法、堆積法、等等）、矩陣相乘$O(n^3)$等等。這些都被視為「簡單」的題目。
- 如果我們有足夠的處理機（或核心），而且在理想情況下每一個thread都可以取得一個處理機執行。
- **NC**這個集合是所有用$O(n^p)$個處理機、在$O(\log_2^q(n))$步驟下可以完成的問題的集合，此地 p 和 q 是兩個和n無關的常數。

再說Inherently Sequential問題: 2/4

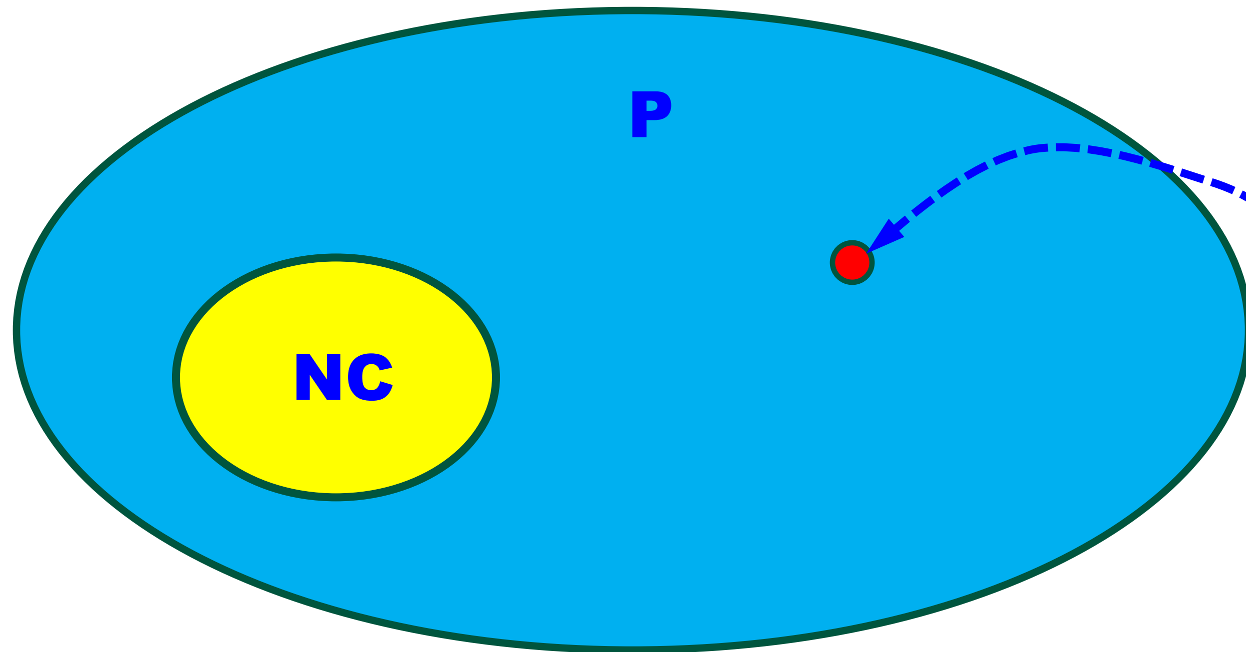
- 每一個**NC**中的問題也是**P**中的問題，因為我們只要把每一個處理機序列化就行了（譬如，把給第一個處理機的工作做完後再讓給第二個處理機繼續、等等），於是總工作量（在只有一個處理機的情況下）就是$O(n^p \log_2^q(n))$。
- 很明顯地，$O(n^p \log_2^q(n)) < O(n^{p+q})$，因為$\log_2(n) < n$。所以，每一個**NC**中的問題也都是**P**中的問題。
- **NC**是Nick's Class，這是Stephen Cook為這一類型問題的命名，此地Nick是另一位在這個領域有很廣很深研究的學者Nick Pippenger的名字。

再說Inherently Sequential問題: 3/4

- 問題是：NC是P的部分集合，但NC 是否等於P卻是個到目前都未曾解答的問題（和P是否等於NP屬於同一類）。若NC不等於P，那麼P中就至少會有一個問題，它不能用$O(n^p)$個處理機、每一個處理機用$O(\log_2^q(n))$個步驟做出來；換言之，這個問題無法很有效地用平行（當然也是並行）的方式解決。
- 您在演算法複雜度（complexity）或理論計算機科學的課程中學到,此地不打算再深入討論。

再說Inherently Sequential問題: 4/4

- 直覺地說，本質上為順序性（inherently sequential）的問題就是難以（或不可能）被平行化的問題，它很可能不在NC中。



若P = NC，則本質上為順序性的問題不存在，每一個在P中的問題都可以平行化。

若P /= NC，則至少有一個在P中的問題不在NC中，這是個難以（或無法）被平行化的問題。

我們學到了什麼?

- 了解thread的意義和不同系統下支援thread的方式
- 使用人thread和系統thread
- 複線作業模型
- 複線作業在多處理機（或核心）下的程式寫作基本概念
- 其它和thread相關的專有名詞
- 協同程式（Coroutine）和執行纖維（Fiber）
- 歷史回顧
- 本質上順序性（Inherently Sequential）的意義

結束，謝謝收看！
期望您再次觀看下一集

請看影片的說明，那兒有取得投影片和程式的連結