並行計算

EP02: 作業系統和硬體基本知識

需要一間十分差的學校才能毀掉一位好學生
但是
卻要一間極好的學校才能救起一位差學生

Dennis J. Frailey

本集內容

- 多處理機系統、對稱多處理機系統、多核心系統
- 雙執行模式
- Interrupt 和 Trap、Interrupt導向
- 叫用系統（System Calls）、叫用系統機制
- 計時器（Timer）
- 讀取-解碼-執行（Fetch-Decode-Execute）週期
- 特殊機器指令

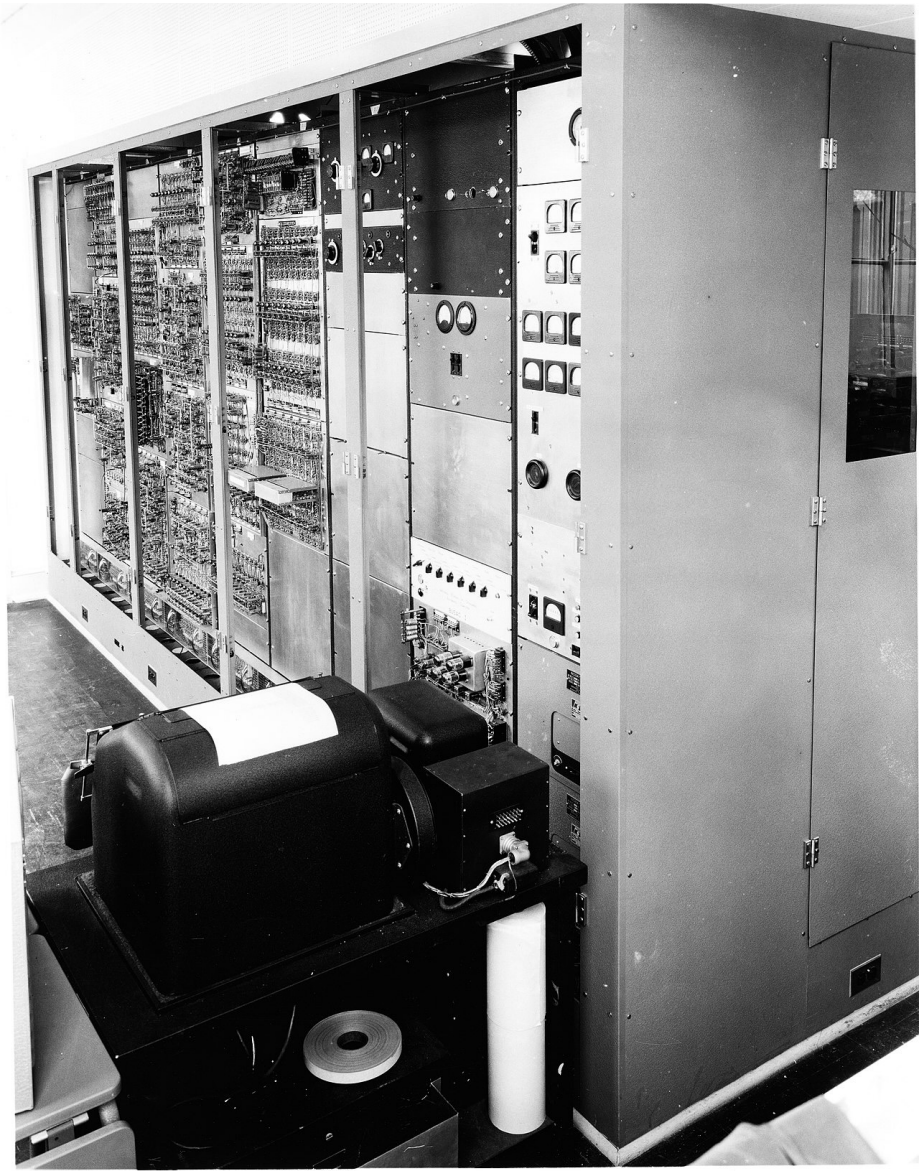
多處理機系統

- 多處理機系統（Multiprocessor Systems）通常也叫做平行系統（Parallel Systems）或緊密連接系統（Tightly Coupled Systems）有若干個CPU。
- 它的優點:
  - 增加生產力：可以完成更多的工作
  - 較大的系統經濟效益：因為系統中有共用的資源，一個多處理機系統會比很多個單處理機系統更經濟
  - 增加可靠度：一個處理機的問題不會使整個系統「當機」

多處理機系統的早期發展: 1/16

- 多處理機系統中有多於一個CPU。
- 多處理機系統在1950年代早期的真空管時代就已經出現。
- 在1948年，美國國家標準技術研究院 （National Institute of Standards and Technology、簡稱NIST；以前叫做National Bureau of Standards）決定以最快速度造一台具有有限計算能力的電腦來滿足當時國家標準局的需求，這台機器叫做SEAC（*Standards Eastern Automatic Computer*），它使用747個真空管，在1950年4月展示、同年6月啟用。

多處理機系統的早期發展: 2/16

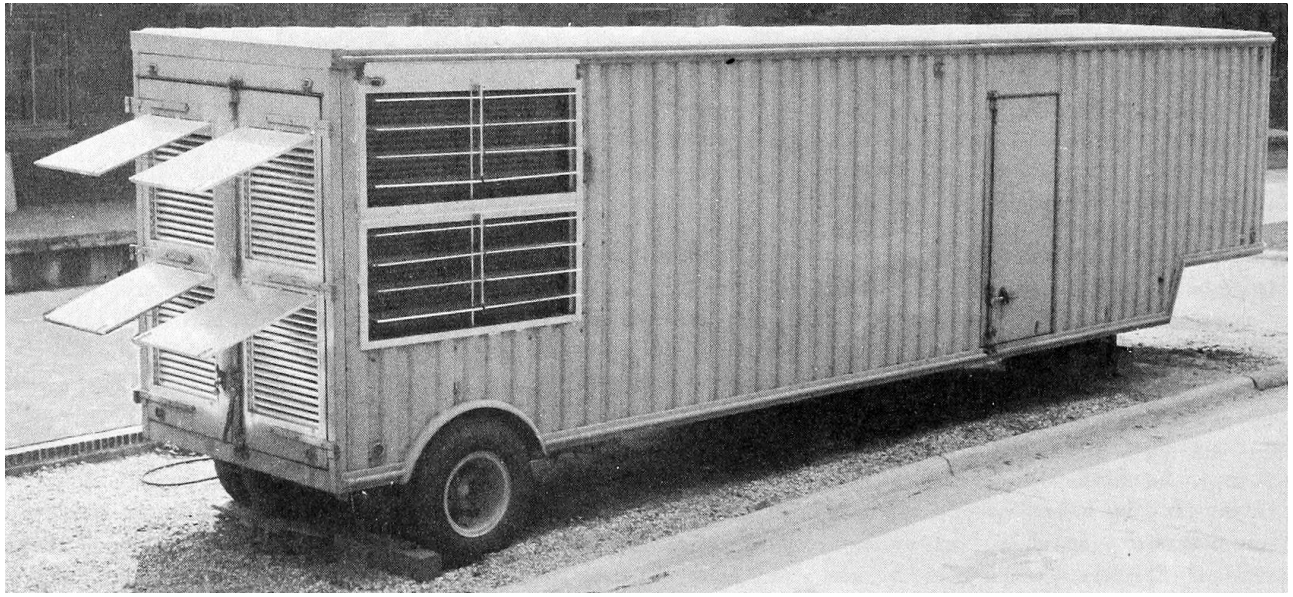


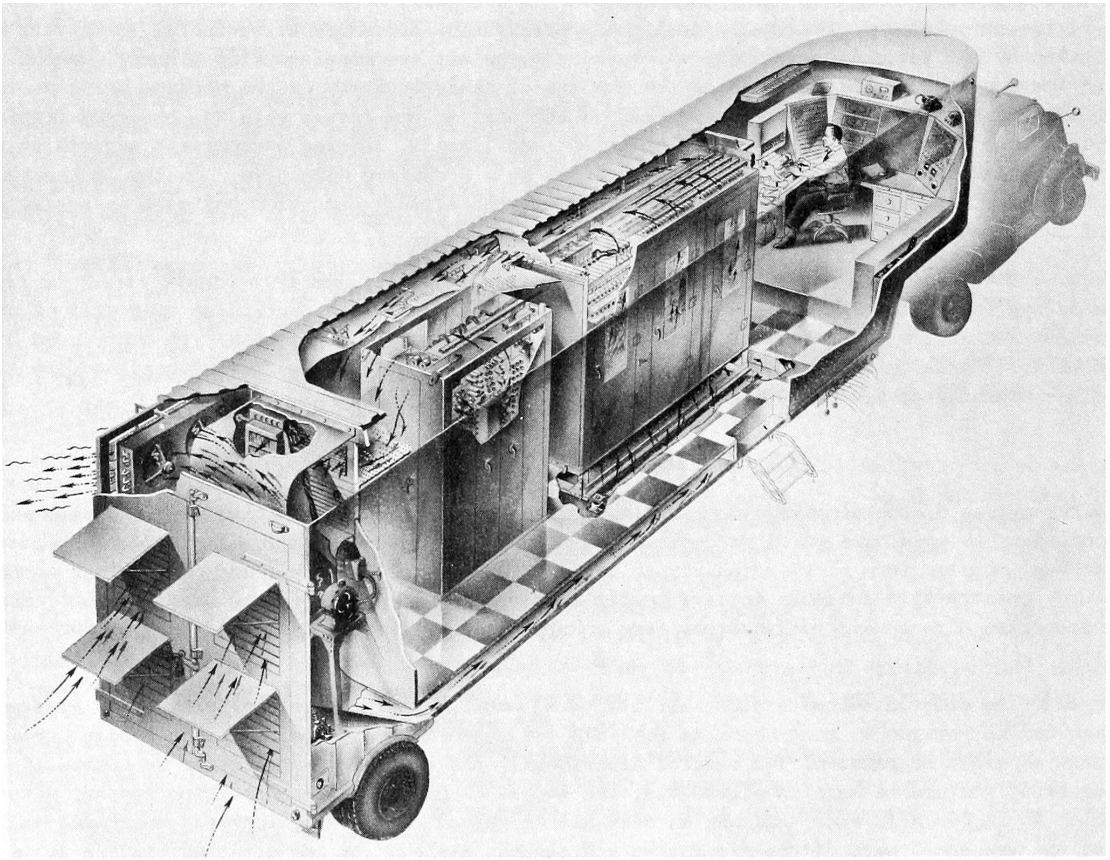
SEAC（1950）

多處理機系統的早期發展: 3/16

- SEAC的繼任者是DYSEAC，仍然使用真空管（一共900個），有512個45-bit的word的記憶體。
- 這是為美國陸軍通信兵造的電腦，總重18噸、整個系統裝在一台拖車上（可以說是世界上第一台可以移動的電腦）。
- DYSEAC在1954年4月通過檢驗、同年5月交貨（給美國陸軍通訊兵、只造了一台），這是發表SEAC四年之後。
- 有意思的是，它可以隨時外接許多儀器，包含一台SEAC。據稱程式指令中有一個指示該指令得在哪一台電腦上執行的bit。
- 因此這可以說是雙CPU平行系統的雛型。

多處理機系統的早期發展: 4/16





DYSEAC （1954）

您可以在下面連結中找到SEAC和DYSEAC的資料:

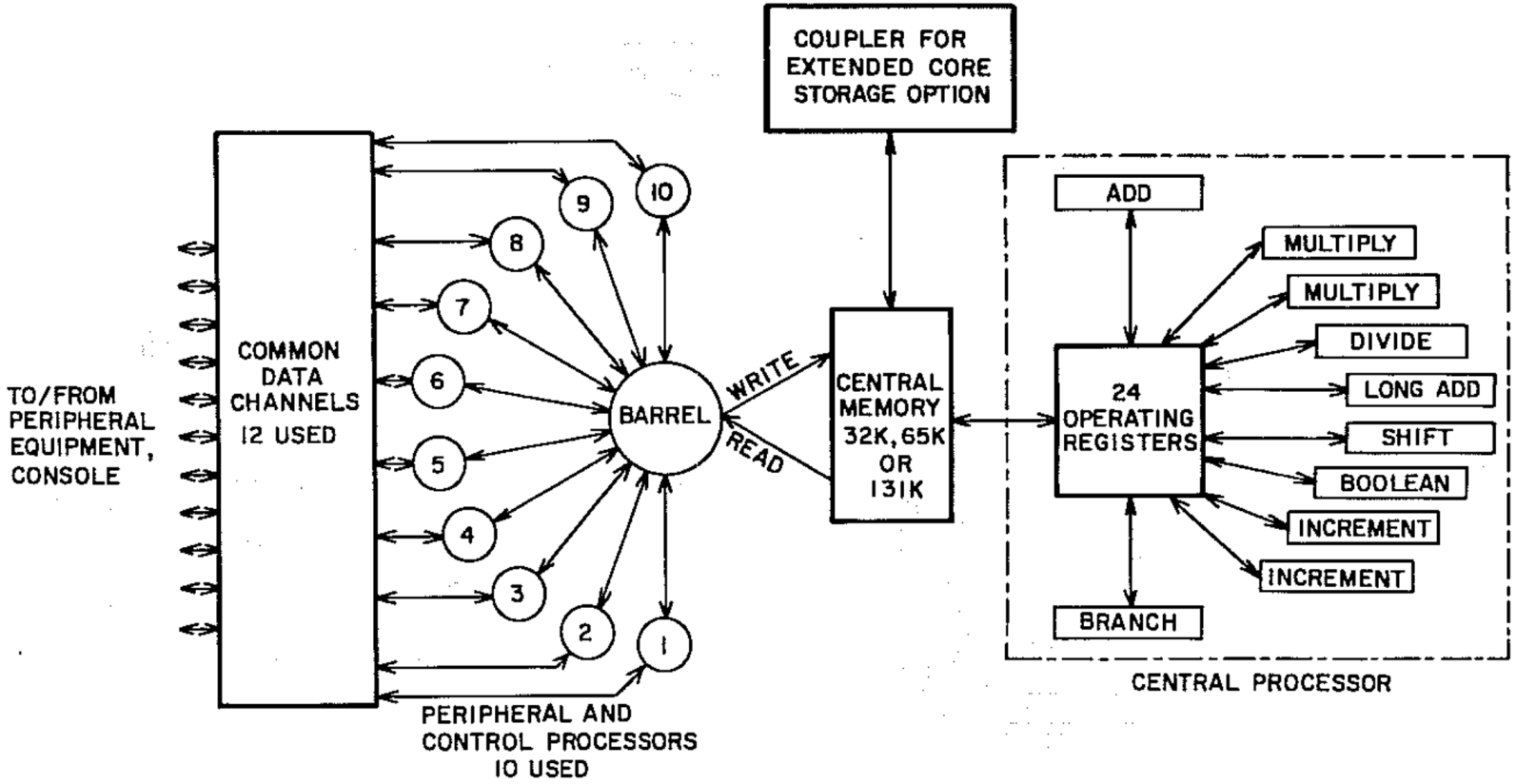
https://www.govinfo.gov/content/pkg/GOVPUB-C13-ccb5220bc32ba6b3c2962c50ed19bf4e/pdf/GOVPUB-C13-ccb5220bc32ba6b3c2962c50ed19bf4e.pdf

https://nvlpubs.nist.gov/nistpubs/circ/1955/circ551-scan1.pdf

多處理機系統的早期發展: 5/16

- Control Data Corporation（CDC）在1964年9月推出世界第一台超級電腦CDC 6600，設計者是鼎鼎大名的Seymour Cray和James E. Thornton。
- CDC 6600是當時最快的電腦，它有一個CPU和10個PPU（Peripheral Processing Unit）。
- CPU只負責計算，其它作業（輸出輸入、控制等）是由PPU處理，頗有主-從（Master-Slave）的味道。
- CDC 6500（1964）和CDC 6700（1969）有兩個CPU；6500有兩個6400 CPU、而6700則有一個6400 CPU和一個6600 CPU。

多處理機系統的早期發展: 6/16



您可以在下面連結中找到CDC 6000 系列的資料:

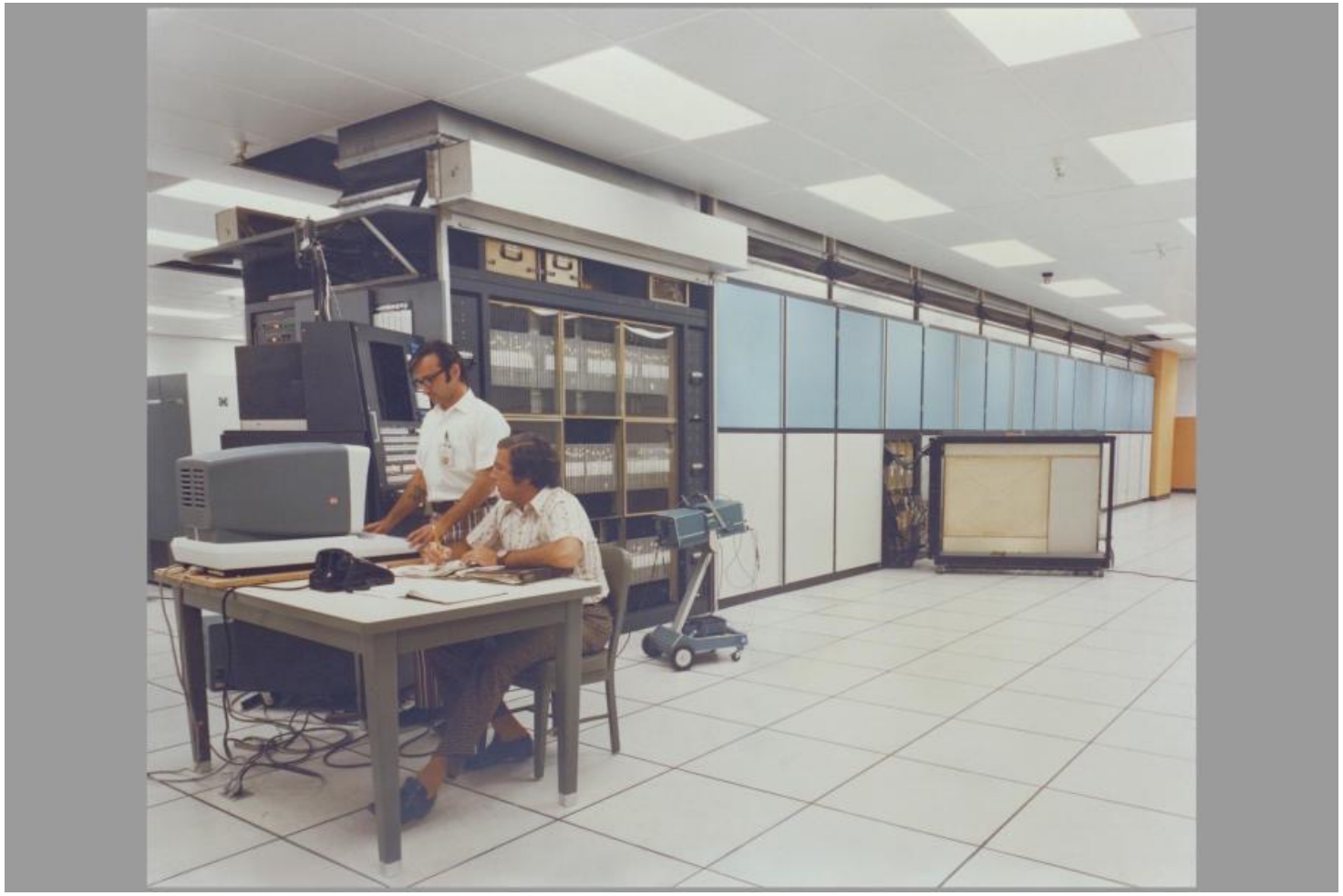
http://bitsavers.trailing-edge.com/pdf/cdc/cyber/cyber_70/60100000AL_6000_Series_Computer_Systems_HW_Reference_Aug78.pdf

多處理機系統的早期發展: 7/16

- 向量處理機（vector processor）是一個有一組可以很有效地同時處理一個相當大的一維陣列的CPU。
- 在1970年代到1990年代初，這是很多超級電腦（譬如Cray）的標準手法，但由於在微處理機上成本高昂，很快就沒落。
- 1966年推出的ILLIAC IV（Illinois Automatic Computer)有四個處理機、每一個都有可以同時處256個元素的一維陣列。

多處理機系統的早期發展: 8/16

ILLIAC IV (1966)
四個處理機、每一個都可以同時處理256個元素的一維陣列



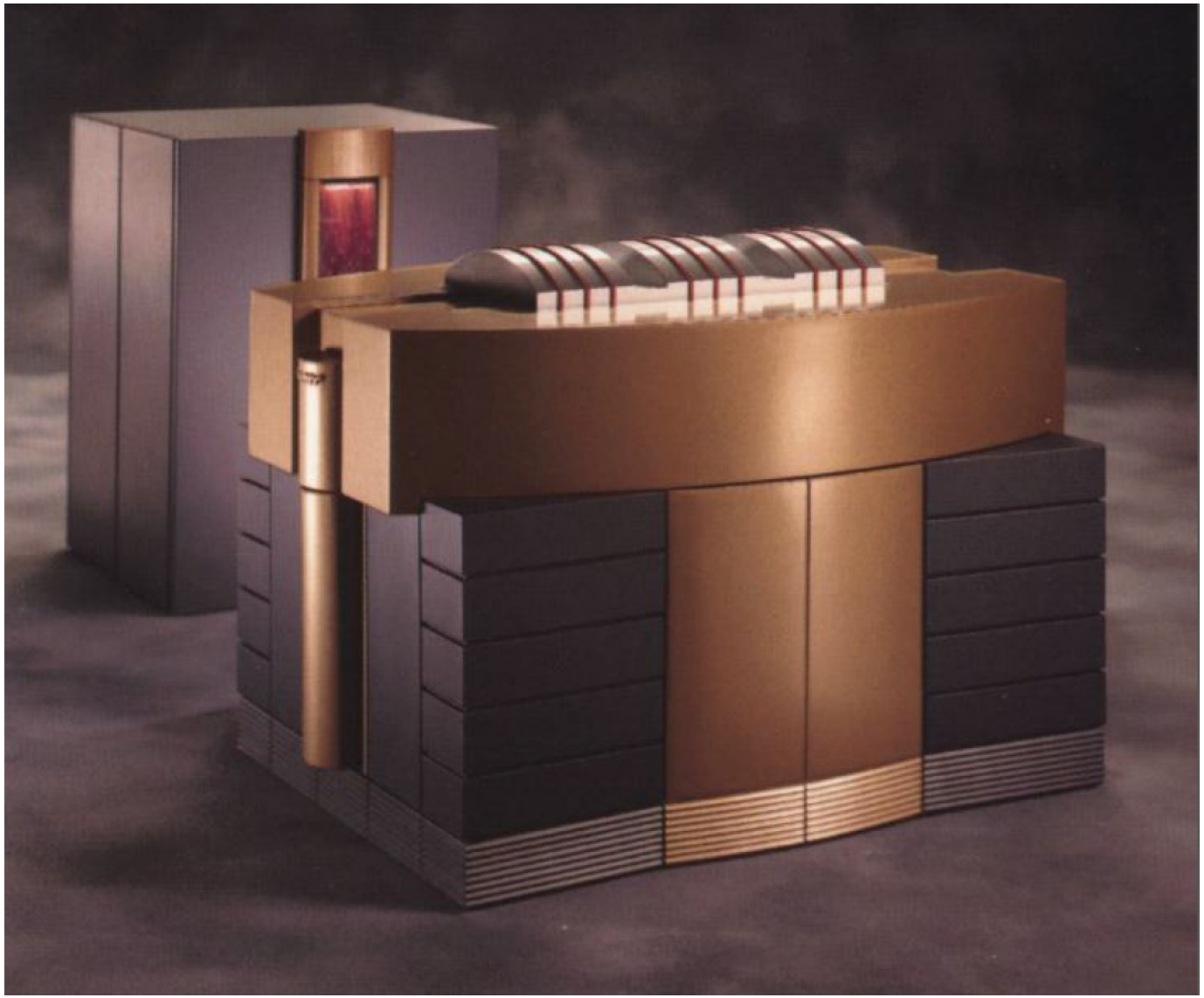
多處理機系統的早期發展: 9/16

- 第一台成功實現向量處理機的超級電腦是Cray-1。
- 這是Seymour Cray離開CDC之後成立Cray Research的第一台使用向量處理機的超級電腦。
- Cray-1只有一個CPU，但後來有多CPU機型。

Cray-1

多處理機系統的早期發展: 10/16

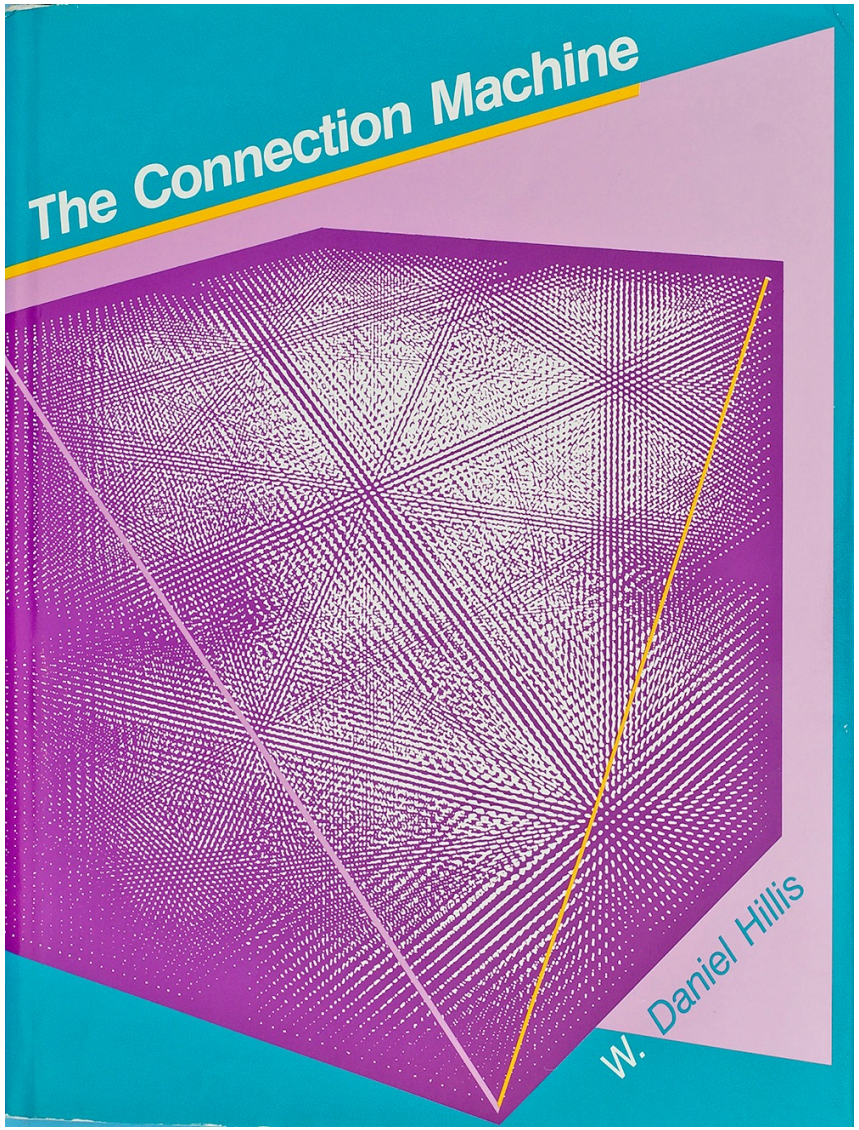
- Cray T90是Cray生產的最後系列、具有向量處理能力的超級電腦，可以從4個到32個CPU。
- Cray X1固然也有向量處理能力（最多可以有64個CPU），但記憶體改成分散式佈置。



Cray T932 (1995)

多處理機系統的早期發展: 11/16

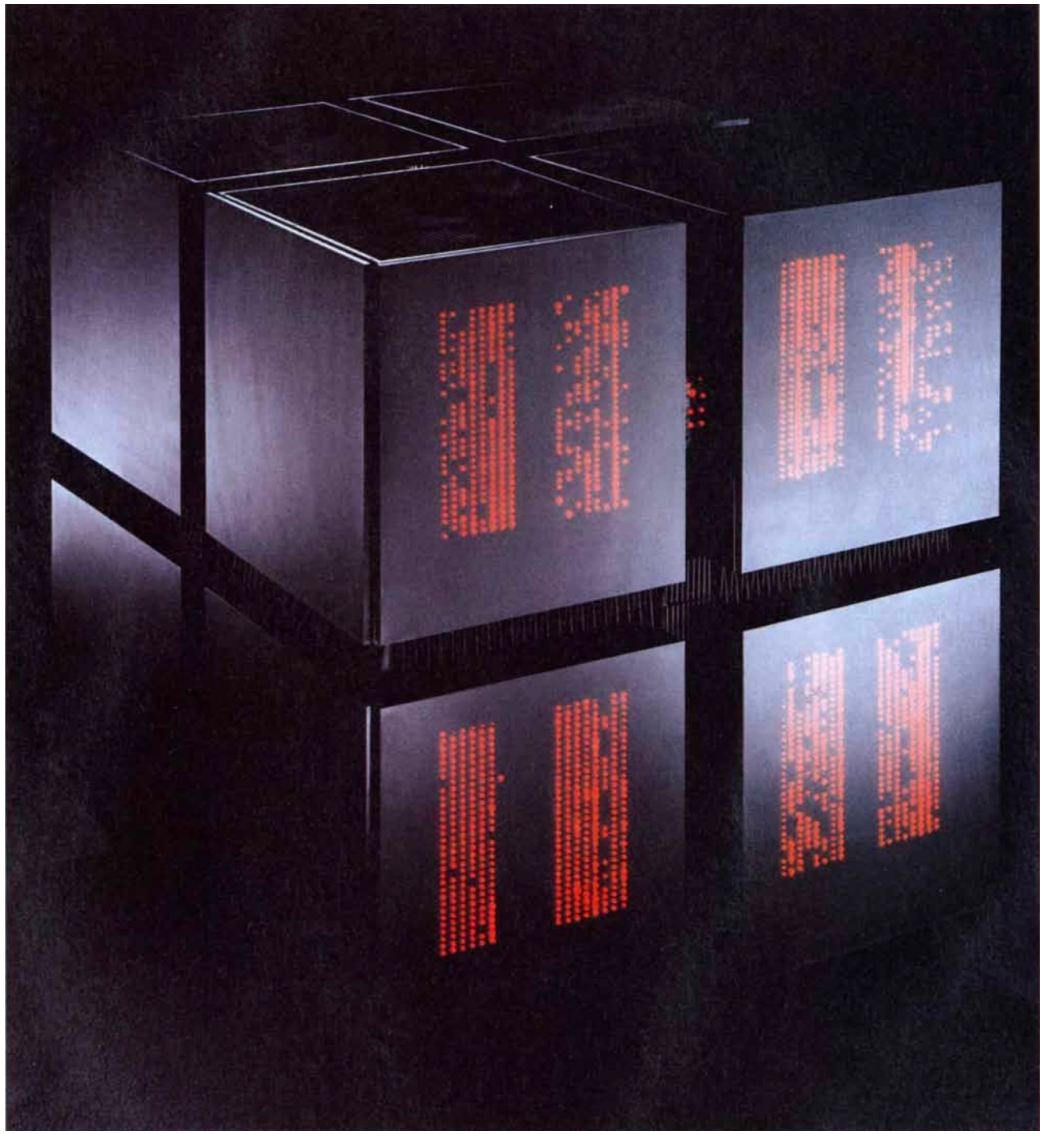
- 1980年代最有意思的多處理機無疑就是Connection Machine。
- Connection Machine原自Daniel Hillis在麻省理工學院（MIT）的博士論文。
- Daniel Hillis和Sheryl Handler在1983創辦Thinking Machine公司生產Connection Machine。



Daniel Hillis的ACM傑出博士論文獎的論文

多處理機系統的早期發展: 12/16

- Connection Machine有8 = 2x2x2個立方體組成，每一個有16塊垂直排列的電路板，電路板上有32個特殊晶片、每一個晶片中有16個處理器（每一個都只有1 bit）和少量的記憶體，共8x16x32x16 = 65536個處理機。
- 立方體的燈是晶片的工作狀態。
- 1985年第一台Connection Machine問世，在計算能力上和當時最快的Cray-2相當，但卻便宜許多。



Connection Machine (1985)

多處理機系統的早期發展: 13/16

- 這一台Connection Machine是裝在美國國家安全局（National Security Agency）的CM-5，大概是用做密碼學的研究。
- CM-5不再用以往的結構而改用SPARC和新的連接方式，更新的CM-5E又改用更快的SuperSPARC。

Connection Machine (CM-5), 1991-1997

多處理機系統的早期發展: 14/16

- 今天，很多桌機、筆電、甚至平板都會有雙核心設計，一台手機的運算速度可能不亞於初期的大型電腦。
- 因為微處理機技術急速發展，目前最快的超級電腦幾乎都用Intel、或AMD或類似結構、並且大量使用GPU。
- 目前最快的超級電腦是HP的Frontier（由Cray生產），它是第一台做到exascale的超級电腦（一億億次浮點運算）。

一億 = 10^9，一億億 = 一億 x 一億 = 10^9 x 10^9 = 10^{18}

多處理機系統的早期發展: 15/16

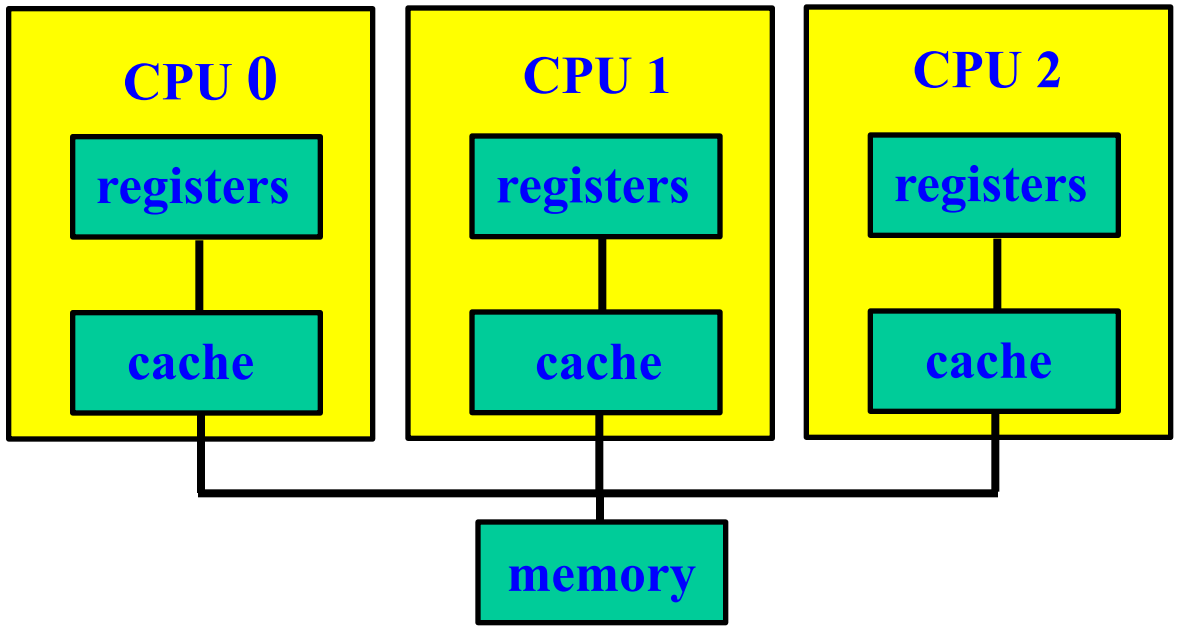
- Frontier（由Cray生產）會在年底之前讓位給El Capitan。
- El Capitan仍然由Cray生產。

多處理機系統的早期發展: 16/16

- 在一般應用上，超級電腦的架構有點殺鷄用牛刀，因此了解一般應用的多處理機和多核心架構、甚至於如何運用這些系統的知識有其必要。
- 接下來的討論會著重在一般系統結構的概念、它的使用方式和注意事項甚至於可能面臨的困難，這是本課程的重點所在。

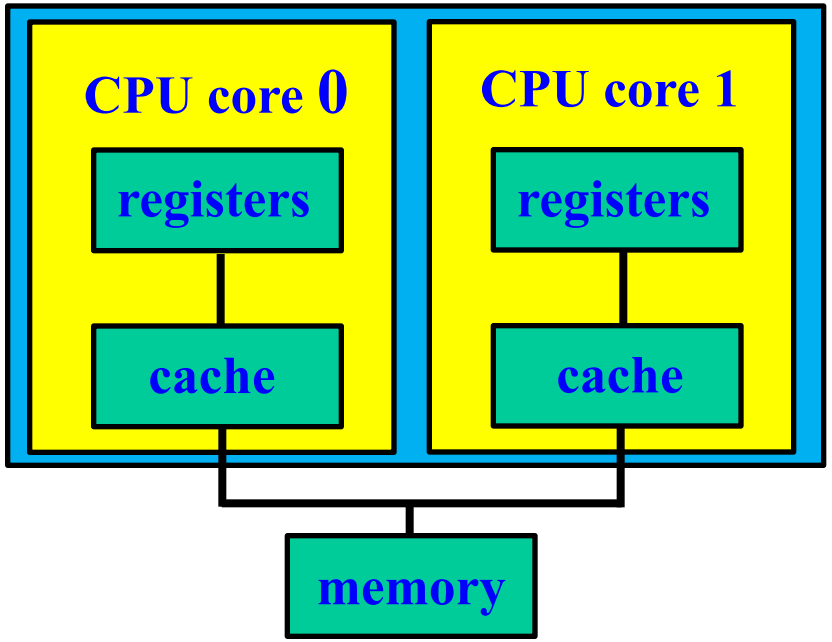
對稱多處理機系統

- 在對稱多處理機系統（Symmetric Multiprocessing — SMP）下，所有系統工作都由同一個OS控制。
- 所有處理機都在同一層次，並沒有誰是主、誰是從（master-slave）的關係。



多核心（Multicore）系統

- 這是在同一個晶片中有多處理機的情況。
- 這是效率較高的系統，因為各核心之間的溝通比各獨立CPU之間的溝通來得快。
- 此外，用電量也較低。

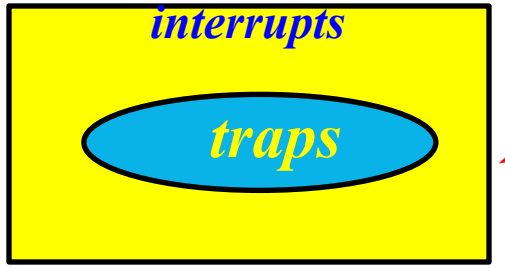


雙執行模式

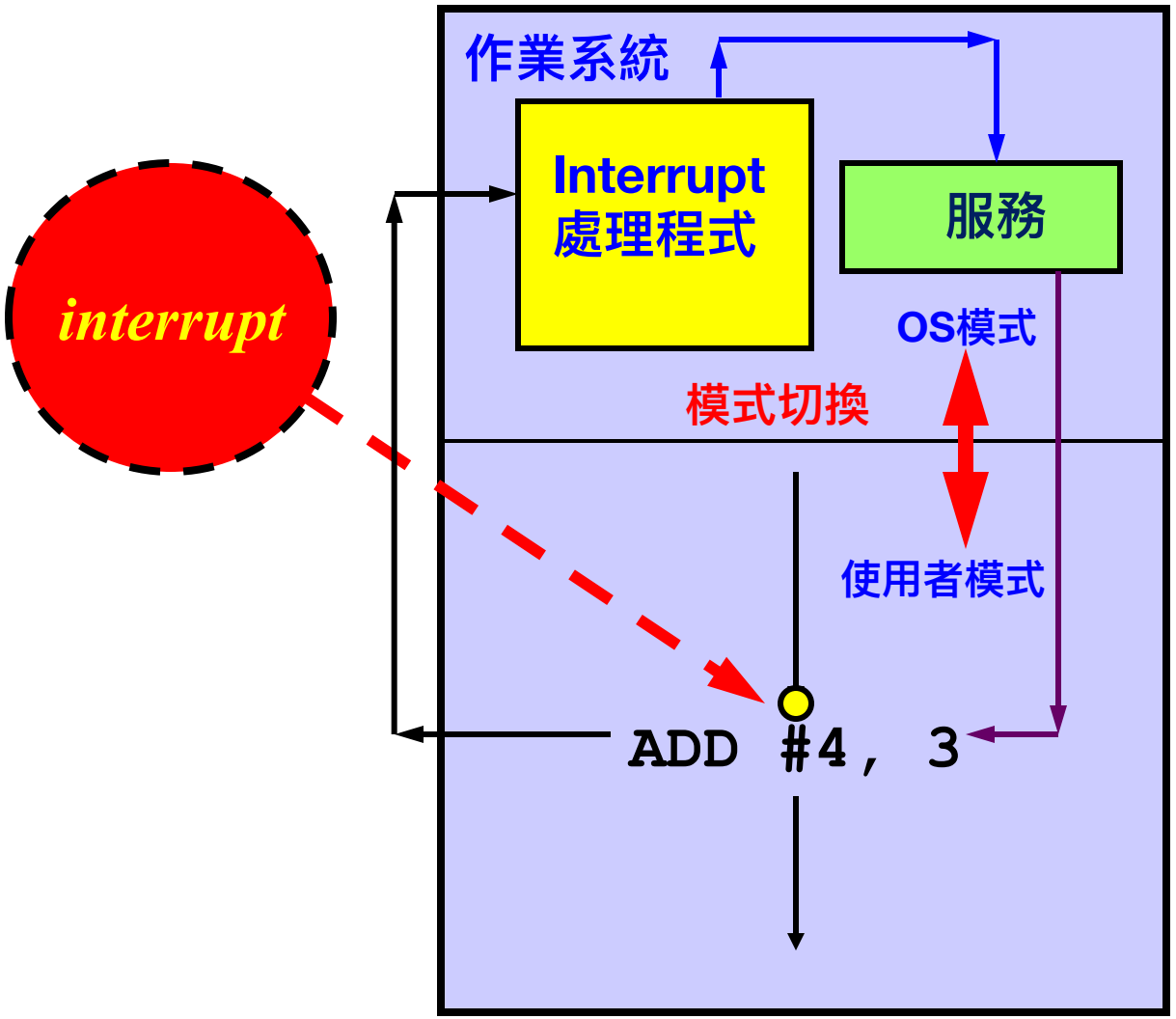
- 近代CPU有雙執行模式（Dual-Mode）：使用人模式（user mode）和作業系統模式（supervisor mode）。這兩個模式由一個模式數元（mode bit）決定。
- 作業系統在作業系統模式下執行，所有使用人程式在使用人模式下執行。
- 某些可能會損害作業系統的指令（譬如輸出輸入、切換執行模式等）是所謂的特權指令（privileged instruction），這些特權指令通常只能在作業系統模式下執行。
- 當CPU切換到作業系統（或使用人程式）時，執行模式就切換到作業系統（或使用人）模式，這是作業系統得完成的工作。
- 有些CPU可能用若干數元（bit）表示使用人模式。在使用2個數元時，00_2是作業系統模式，01_2、10_2、和11_2就可以給三個使用人程式使用。

Interrupt 和 Trap

- 一個需要作業系統處理的事件就是一個中斷（interrupt）。這些事件包含有：輸出/輸入完成、按下鍵盤的鍵、程式請求服務、程式中用0除、程式使用不在被配置記憶體內的位址、等等。
- Interrupt可能由硬體產生（譬如終端機斷線、輸出/輸入完成），也可能由軟體產生（譬如程式中用0除、程式要求系統服務）。
- 若一個Interrupt是由軟體產生（譬如用0除），這個Interrupt通常叫做Trap；所以，所有的Trap是所有Interrupt的部份集合。
- 近代的作業系統是interrupt導向（interrupt driven）的；當作業系統進入電腦系統之後，不論是否有程式執行，作業系統本身都在閑置狀態，有了interrupt就立刻處理。
- 作業系統閑置 ≠ CPU閑置



何謂Interrupt導向?

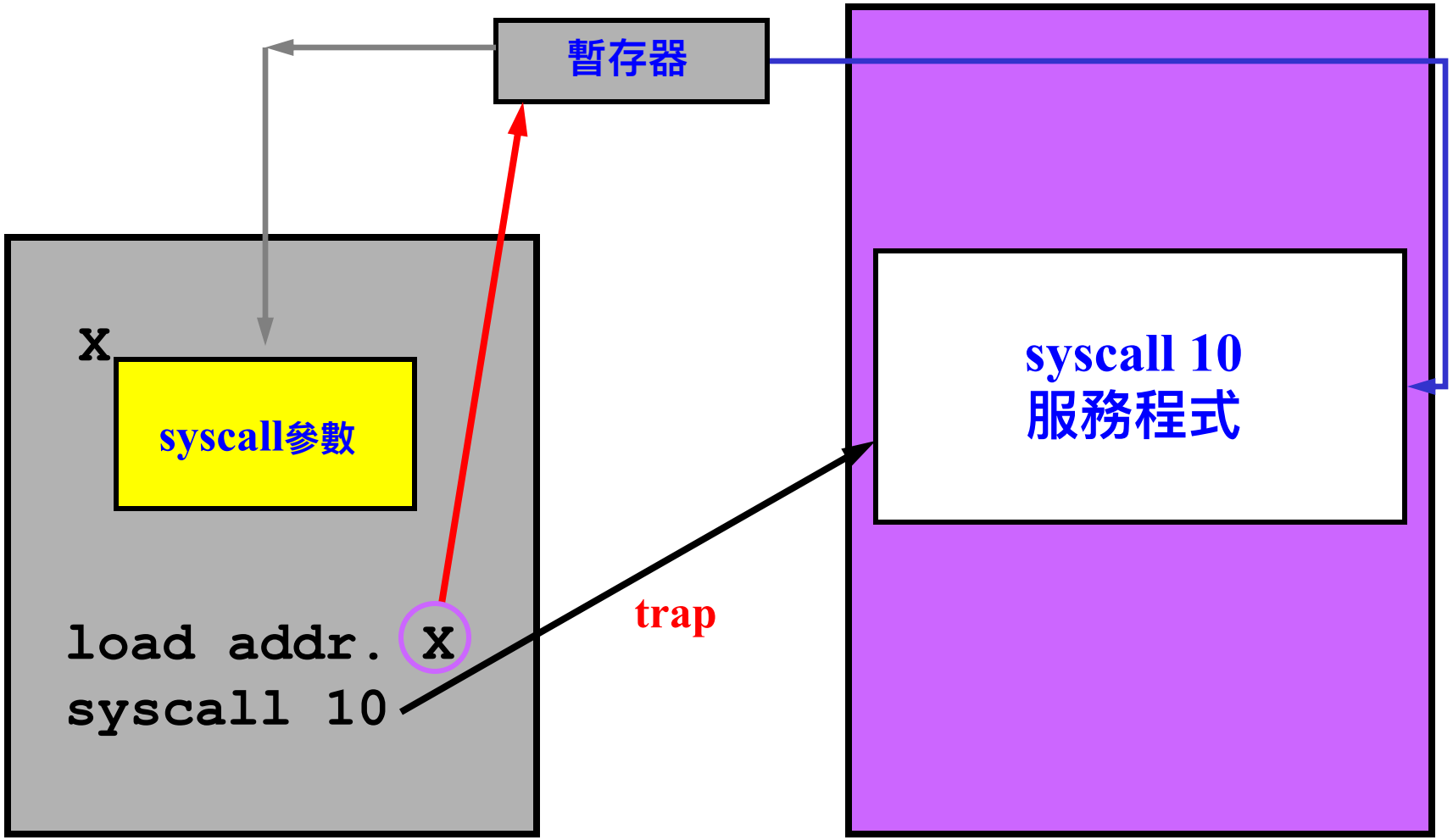


- Interrupt出現就起動作業系統
- 控制權轉移到作業系統（在OS模式下執行）
- 暫停正在執行的程式
- Interrupt處理程式分析interrupt的來源、種類、成因等等，交由合適的服務程式處理
- 作業系統處理完interrupt後，會繼續執行某個被暫停的程式（可能不是因為interrupt發生而進入作業系統時被暫停的那一個）
- 在繼續執行該被暫停的程式之前，若該程式是使用者程式，CPU執行模式得切換到使用者模式

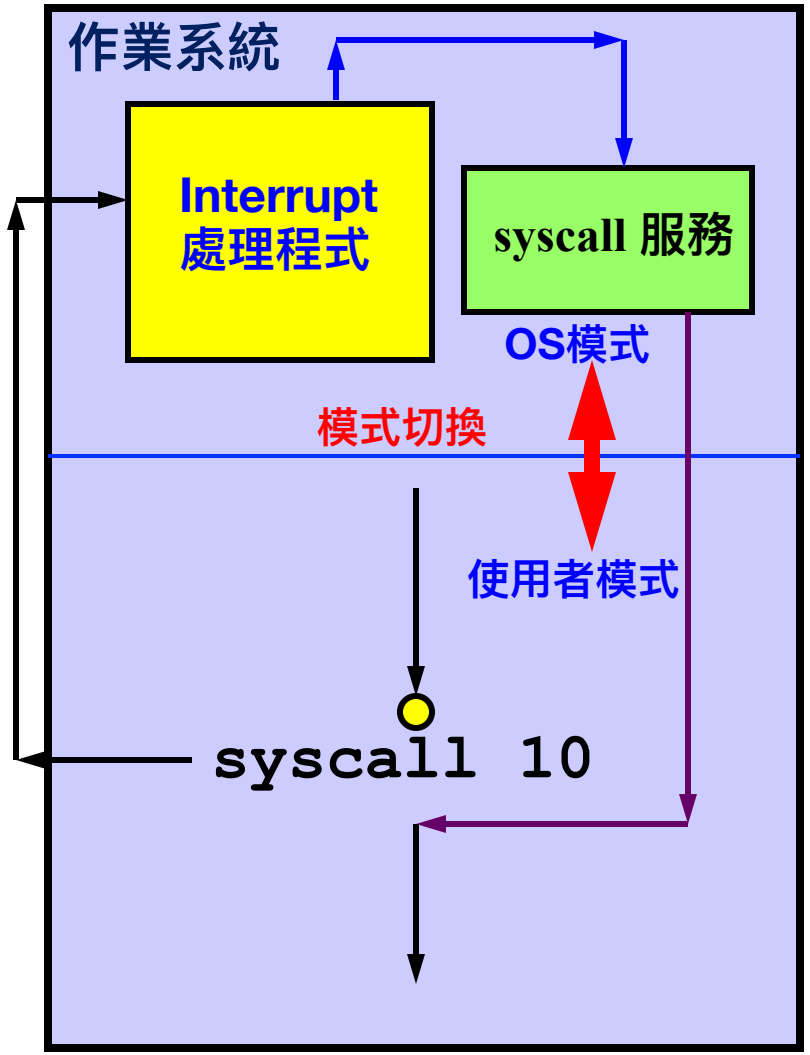
叫用系統

- 叫用系統（system calls）是作業系統提供服務的一個介面（interface）。
- 執行叫用系統時會產生一個interrupt（事實上是一個*trap*，因為是由軟體程式產生），於是使用叫用系統的程式就會被暫停執行。
- 典型的叫用系統服務有：
  - Process控制：譬如產生和摧毀process
  - 檔案管理：譬如開檔和結檔
  - 設備管理：讀取或寫入資料
  - 資訊維護：譬如取得日期或時間
  - 通訊：譬如收發訊息

叫用系統機制: 1/2



叫用系統機制: 2/2



- 叫用系統會產生一個trap
- 正在執行的程式會被暫停
- 控制權交給作業系統
- Interrupt處理程式分析該interrupt，之後控制交給對應的syscall服務程式
- 處理完後，作業系統會讓一個被暫停的程式繼續執行，但未必是原本叫用系統服務的那一個程式
- 其它細節和Interrupt的相同

計時器（Timer）

- 作業系統必須控制CPU的使用方式，不能讓一個使用者程式長期佔用CPU而不叫用系統的服務（譬如輸出和輸入）。
- 系統一般有兩種計時器： 實時時鐘（Real Time Clock）和區間計時器（Interval Timer）。實時時鐘和牆上的鐘和手錶無異。
- 區間計時器是個倒數的，當值降到0時會產生一個Interrupt。
- 當一個使用者程式開始（或繼續）執行前，OS在區間計時器上設定一個很小的時間值（time quantum），然後讓程式執行。同時，區間計時器開始倒數，一旦到0時，interrupt就中斷該程式的執行而把控制權交給OS、讓OS採取適當的處理。
- 這樣，一個一直佔用CPU只做計算而沒有其它行為的程式就無法長期佔用CPU。

讀取-解碼-執行的週期: 1/13

- CPU執行一道機器指令時會有幾個階段：讀取（fetch）、解碼（decode）和執行（execution）。
- 我們可以把這三個階段再細分如下：
  - 讀取（fetch）：從記憶體中把下一道指令抄入CPU
  - 解碼（decode）：分析該指令並且決定運算碼（operation code）和運算元（operand）
  - 載入運算元（load operands）：把運算元的內容從記憶體或暫存器（register）抄入CPU
  - 執行（execute）：執行運算碼指定的工作並且得出結果
  - 儲存（save）： 把得來的結果存入暫存器和記憶體

讀取-解碼-執行的週期: 2/13

- 考慮下面這一道假設性的指令

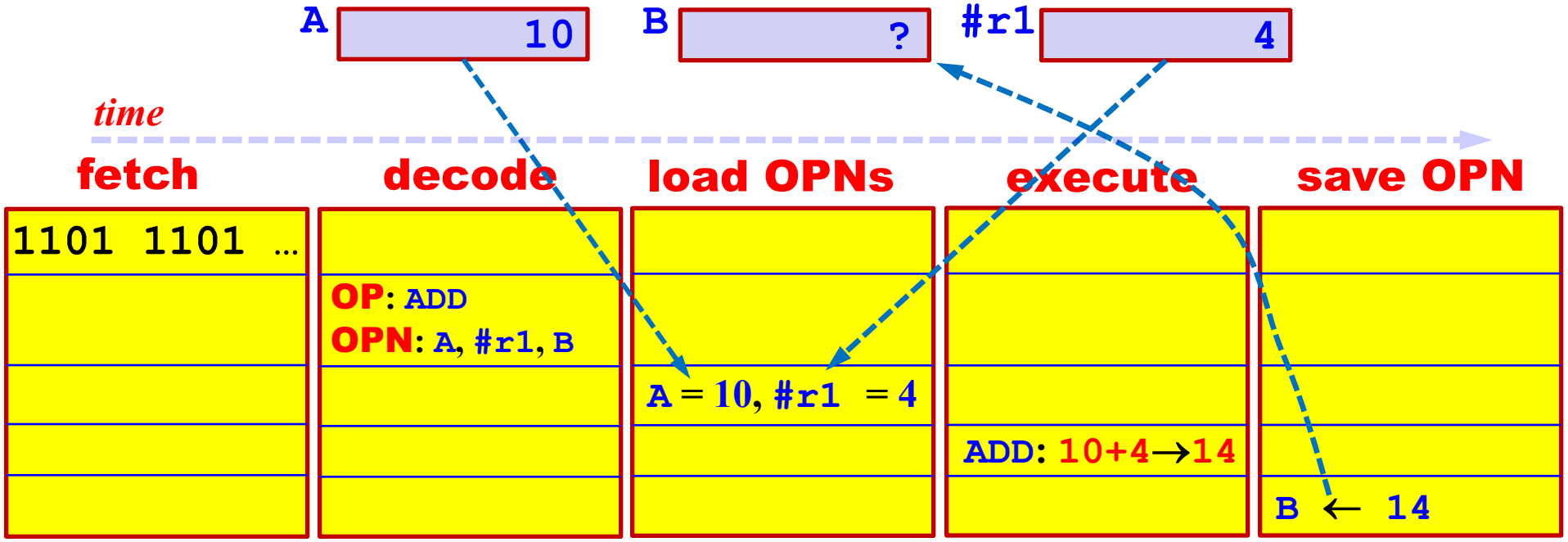
```
ADD   A, #r1, B
```

- 它把在記憶體A處的值和在暫存器1（#r1）的值相加，把結果存入記憶體B處
- 假設這道指令的二進位表示如下，此地前八位是運算碼

```
1101 1101 … …
```

讀取-解碼-執行的週期: 3/13

- 下面自左而右顯示CPU執行該指令的各階段



- 在任何時刻，CPU中只有一個階段在工作、其餘四個階段都閑置
- 我們何不在CPU解碼時把下一道指令抄入CPU呢?

讀取-解碼-執行的週期: 4/13

- 接下來我們看下面的五道指令:

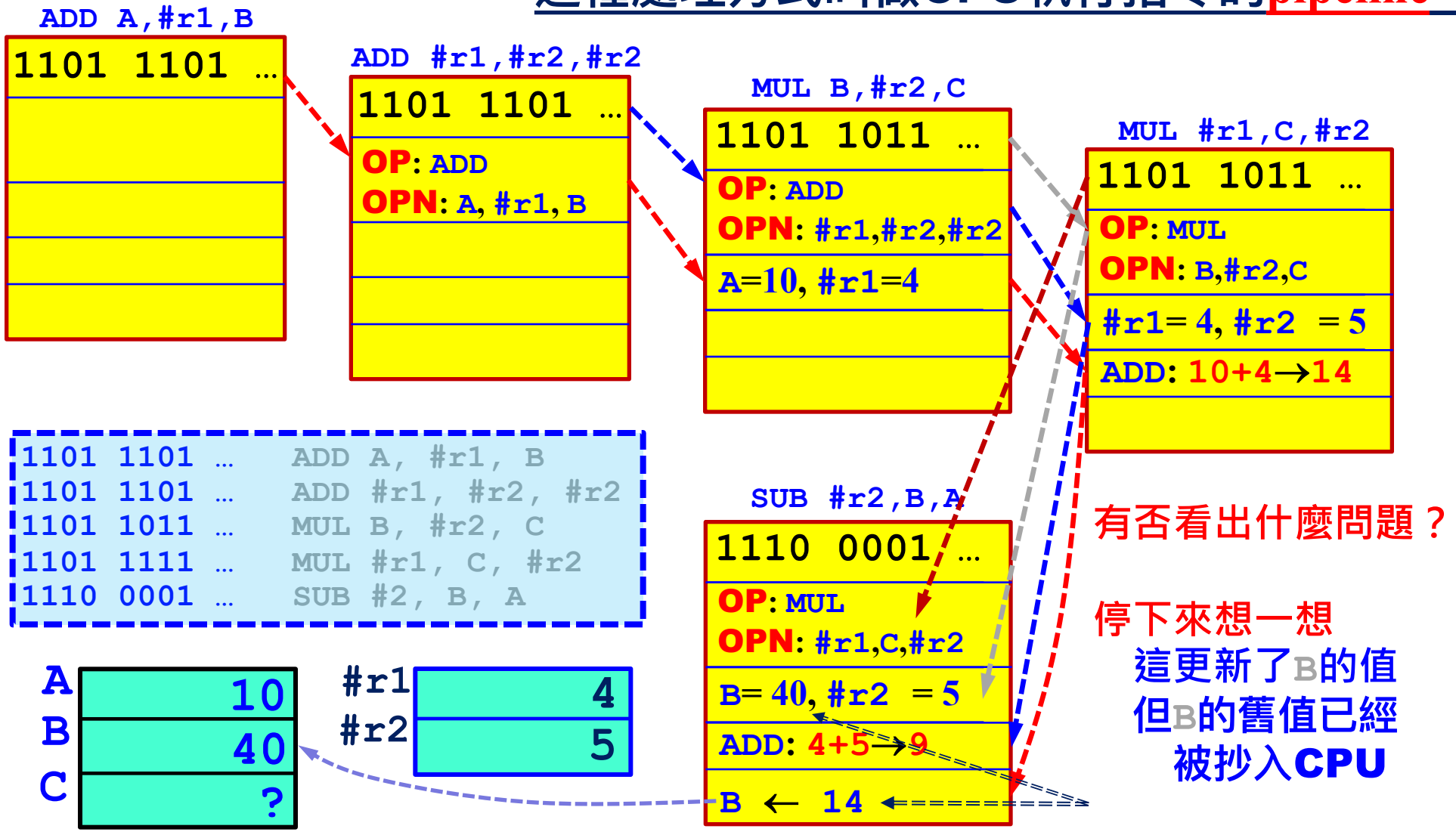
```
1101 1101 …   ADD A, #r1, B
1101 1101 …   ADD #r1, #r2, #r2
1101 1011 …   MUL B, #r2, C
1101 1111 …   MUL #r1, C, #r2
1110 0001 …   SUB #r2, B, A
```

- 每一道指令都使用前兩個運算元運算、並且把結果存入第三個運算元。此地ADD、SUB和MUL分別表示（整數）加、減和乘。

	A	B	C	#r1	#r2
initial	10	40	?	4	5
ADD_1	10	**14**	?	4	5
ADD_2	10	14	?	4	**9**
MUL_1	10	14	**126**	4	9
MUL_2	10	14	126	4	**504**
SUB	**490**	14	126	4	504

讀取-解碼-執行的週期: 5/13

這種處理方式叫做CPU執行指令的pipeline（流程）



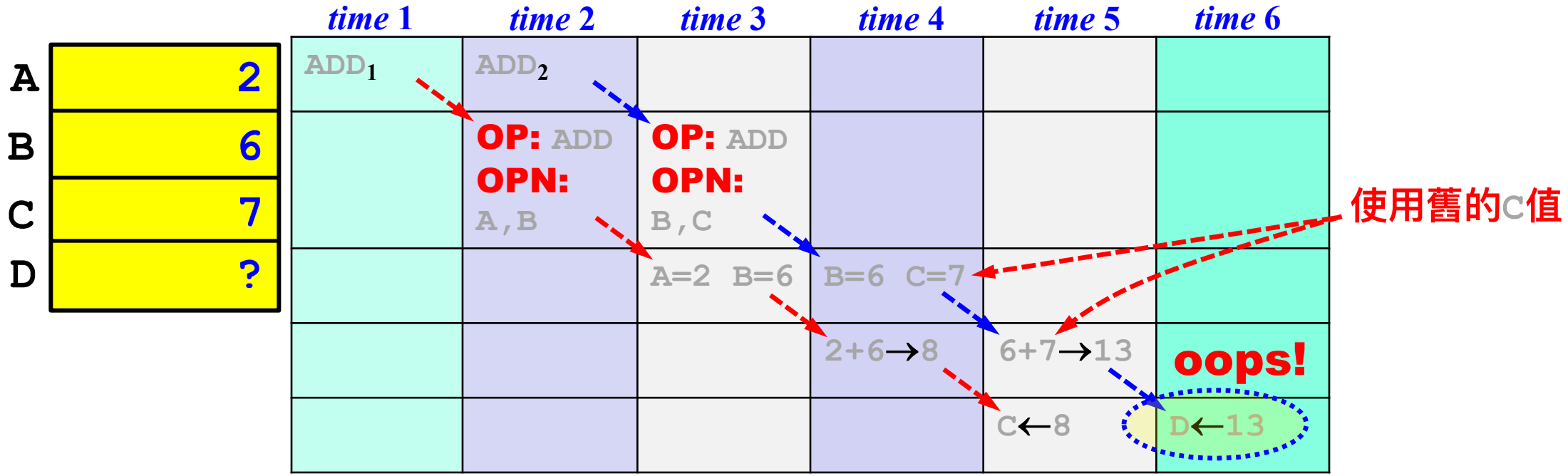
讀取-解碼-執行的週期: 6/13

- 再看下面兩道指令，我們期望 D = 14

```
ADD   A,B,C
ADD   B,C,D
```

第一道指令的結果被第二道指令使用（C=A+B=2+6=8）

這是否有一點並行共享的味道? （D=B+C=6+8=14）



資料衝突（Data Hazards）：這指的是一道指令依賴已經被抄入**pipeline**的指令的結果

第二道指令中的 C 使用到第一道已經被抄入pipeline的指令的結果

讀取-解碼-執行的週期: 7/13

- 有些指令對並行運算非常重要
- Compare-and-Swap (CS)指令就是個好例子

下面各步驟是由單一指令完成

```
int CS(int *p, old, new)
{
    if (*p != old)
        return FALSE;
    *p = new;
    return TRUE;
}
```

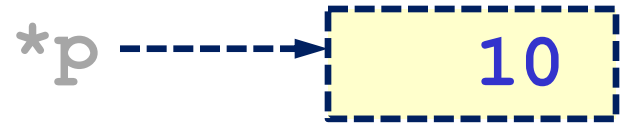
若 *p 和 old 的值不同
    傳回 FALSE
    而且不更新
不然， 用 new 更新 *p 的值
    並且傳回 TRUE

```
done = FALSE;
while (!done) {
    value = *p;
    done = CS(p, value, value+x);
}
```

因為交錯執行，這個process可能在執行完
    value = *p 之後就被暫停
另一個process可能更改 *p
    到執行 CS 時， *p 和 value 的值可能不相同!
只有 *p 和 value 的值相同時，才會用 value+x 的.
    值更新*p

讀取-解碼-執行的週期: 8/13

- 假設 `*p = 10`



- 考慮兩個process P_1 和 P_2:

	Process P_1		Process P_2
	`int x;`		`int y;`
	…		…
1	`x = *p;`	3	`y = *p;`
2	`… = CS(p,x,x+1);`	4	`… = CS(p,y,y+2);`
	…		…

- 若 P_1 和 P_2 在只有一個CPU而且也沒有pipeline的機器上執行。
- 至少有兩個不同的交錯執行的可能：
  - **1-2-3-4：** 一個process在另一個開始之前就已經完成
  - **1-3-2-4：** 一道 `CS` 指令在另一個process兩道敘述之間執行（交錯執行）

讀取-解碼-執行的週期: 9/13

- 一個process完成後另一個才開始

Process P_1	x	Process P_2	y	*p
				10
x = *p	10			10
CS(p,x,x+1)	10			11
		y = *p	11	11
		CS(p,y,y+2)	13	13

這是正確結果

*p 的更新是序列、而非並行

讀取-解碼-執行的週期: 10/13

- 若 CS 在另一個process的 *p 設定敘述和 CS 之間執行（亦即交錯執行）

	Process P_1	x	Process P_2	y	*p
1					10
2	x = *p	10			10
3		10	y = *p	10	10
4	CS(p,x,x+1)	10		10	11
5		10	CS(p,y,y+2)	10	11

因為 *p 和 y 不相同，process P_2 在這一次循環中無法更新

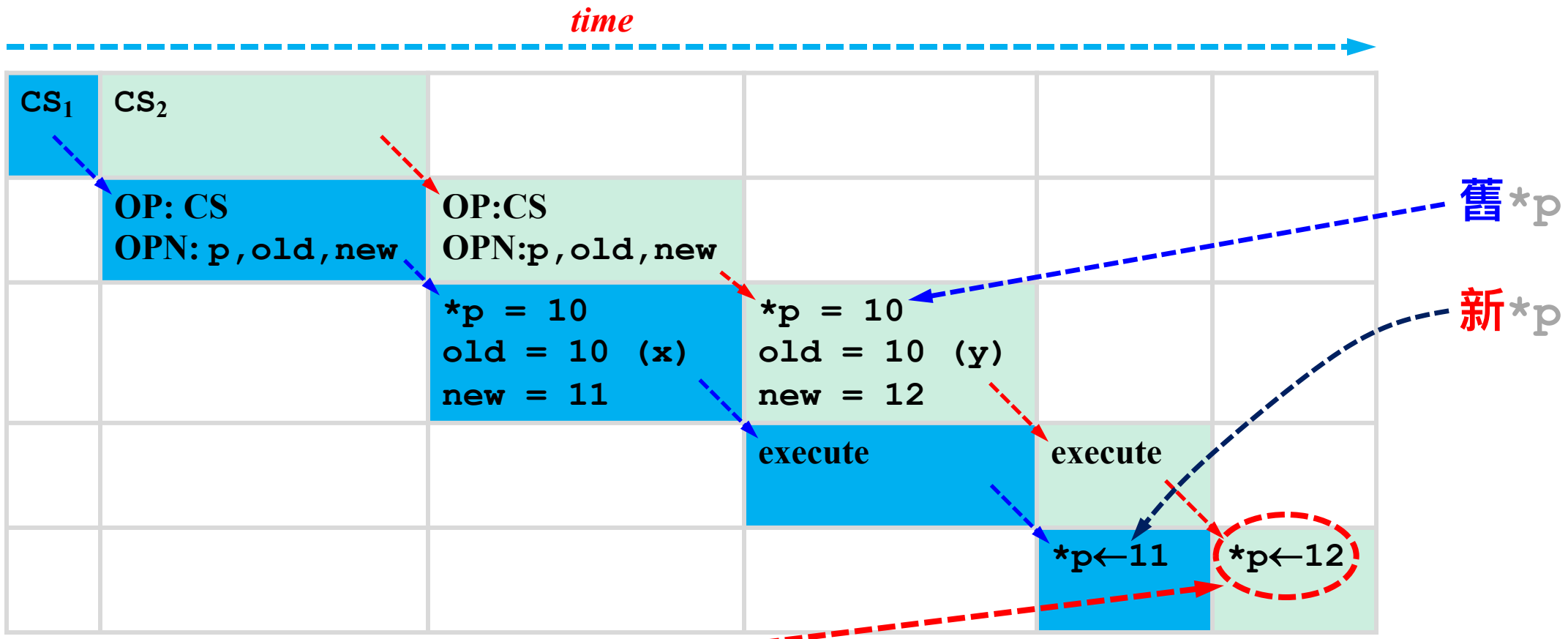
被 P_1 更新

P_2 必須再次嚐試更新

但是，若上述情況一再重覆， P_2 根本沒有機會更新

讀取-解碼-執行的週期: 11/13

- 若兩個 CS 在有pipeline的CPU中執行



這看起來不正確
因為它使用舊的 *p = 10、而不是更新後的 *p = 11

讀取-解碼-執行的週期: 12/13

- 若兩道 CS 指令在不同的CPU或核心中執行

CPU 1	CPU 2
CS1	CS2
OP: CS **OPN:** p,old,new	**OP:** CS **OPN:** p,old,new
*p = 10 old = 10 new = 11	*p = 10 old = 10 new = 12
execute	execute
*p = 11	*p = 12

結果是11還是12？

這取決哪個CPU比較快

若CPU 1比較快，

於是先把*p更新成11、

然後較慢的CPU 2再把

*p更新成12

反之，結果為11

讀取-解碼-執行的週期: 13/13

- 為了克服上述的問題，近代CPU引入新類型的機器指令。
- 這些叫做不能分割（atomic）的指令，也叫做原始或元指令
- 不能分割的指令基本上就是等於在執行時不用pipeline功能

特殊機器指令: 1/3

- Atomic指令在執行時不容許有交錯執行或被其它指令分割。當一道atomic指令被CPU偵測到並且執行時：
  - 所有在CPU內各階段的其它指令都會被暫停、等到這一道atomic指令執行完畢後才會被繼續，而且有些指令（譬如用到atomic指令產生的結果的指令）可能會重頭來過（至少得再次抄入它所使用的各運算元）。
  - Atomic指令在執行時不能被interrupt打斷，必須從頭做到尾、一氣呵成。
  - 如果有若干道atomic指令進入CPU或核心，它們都會被一個接一個地順序執行，但順序為何則是由硬體決定。這種情況通常在有若干個CPU或核心的系統中出現。

特殊機器指令: 2/3

- 一道atomic指令通常表示：
  - 當它在執行時，在CPU內它就是唯一的執行者（其它的都停頓），而且還不能被interrupt打斷。
  - 當CPU執行一道atomic指令時，所有其它在CPU內各階段的指令都得停頓、使得這一道atomic指令是唯一在執行的指令。而且還不容許interrupt發生！
  - 當一道atomic指令執行完後其它指令才能繼續，而且一些指令和運算元還得重新抄入CPU。
- 您會在*Computer Architectures* 這門課中學到更多，此地只是提供一個直觀上、而且滿足往後討論的說法。

特殊機器指令: 3/3

- 我們會在後面講解同步（synchronization）時討論到atomic指令TS（Test-and-Set），前面講到的CS（Compare-and-Swap）是另一個常見的例子。
- 若沒有atomic指令的幫助，當若干道指令同時更新一個共用的記憶體內容時，就會產生競爭狀態（race conditions）。我們會在不久的將來講解競爭狀態，這是一個在並行計算中令人頭痛、又很難解決的難題。
- 特權指令（Privileged Instructions）：這些指令，一般而言，只能在作業系統模式下執行，使用者程式用到這些指令時就會產生一個trap，從而程式會被作業系統終止執行。

我們學到了什麼?

- 為了保護作業系統，在電腦發展初期就有雙執行模式
- Interrupt是一個很有效的、在需要時讓作業系統介入的方式
- 為了不讓一直用CPU做計算的程式佔用CPU，於是有了區間計時器
- CPU執行指令時有「讀取-解碼-執行」階段。但是，為了提昇CPU效率，這三個階段可以同時處理若干條指令，這是CPU的pipeline。
- 然而，CPU的pipeline卻可能產生一些問題（譬如資料衝突 — Data Hazards）；不但如此，若干CPU或核心同時更新一個共用的記憶體時問題更大。
- 於是不能分割（atomic）指令於焉而生，這對並行計算的同步（synchronization）作業影響極大。
- 最後，我們回顧特權指令。

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