本文衍生自維基百科

程式人《十分鐘系列》

計算機結構與作業系統裏

資工系學生們經常搞錯的那些事兒！

陳鍾誠

2017 年 1 月 5 日

今天

- 我在計算機結構課程口試了一些學生

這學期的計算機結構

- 我是採用 nand2tetris 這門網路線上課程
- 要求學生們要寫完《硬體部分的作業》
- 也就是從 nand 閘一直設計到 CPU 與整台電腦

但是有很多同學

- 沒辦法完成所有的作業！

所以我就

- 要他們讀一本計算機結構的書之後來讓我口試。

結果發現

- 有些同學念了書之後，對很多概念不清楚。
- 一看之下才發現，他們念的書和文章也交代得很不清楚。

哪裡不清楚呢？

請容我舉一個例子

虛擬記憶體

- 是基於對位址空間的重定義，即把位址空間定義為「連續的虛擬記憶體位址」，以藉此「欺騙」程式，使它們以為自己正在使用一大塊的「連續」位址。

問題是

- 看完上述描述之後，你知道甚麼是虛擬記憶體了嗎？

其實上面有點斷章取義

不過對於初學者而言

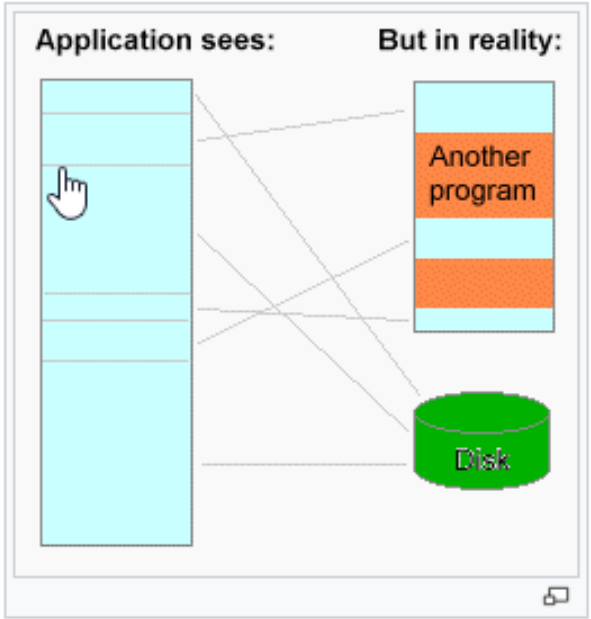
- 真的很容易誤解！

讓我們看看完整版

虛擬記憶體 [編輯]

維基百科，自由的百科全書

虛擬記憶體是電腦系統記憶體管理的一種技術。它使得應用程式認為它擁有連續的可用的記憶體（一個連續完整的位址空間），而實際上，它通常是被分隔成多個實體記憶體碎片，還有部分暫時儲存在外部磁碟記憶體上，在需要時進行資料交換。與沒有使用虛擬記憶體技術的系統相比，使用這種技術的系統使得大型程式的編寫變得更容易，對真正的實體記憶體（例如RAM）的使用也更有效率。



注意：**虛擬記憶體**不只是「用磁碟空間來擴充功能實體記憶體」的意思——這只是擴充記憶體級別以使其包含硬碟機而已。把記憶體擴充功能到磁碟只是使用虛擬記憶體技術的一個結果，它的作用也可以通過覆蓋或者把處於不活動狀態的程式以及它們的資料全部交換到磁碟上等方式來實作。對虛擬記憶體的定義是基於對位址空間的重定義的，即把位址空間定義為「連續的虛擬記憶體位址」，以藉此「欺騙」程式，使它們以為自己正在使用一大塊的「連續」位址。

現代所有用於一般應用的作業系統都對普通的應用程式使用虛擬記憶體技術，例如文書處理軟體，電子製表軟體，多媒體播放器等等。老一些的作業系統，如DOS和1980年代的Windows，或者那些1960年代的大型電腦，一般都沒有虛擬記憶體的功能——但是Atlas，B5000和蘋果公司的Lisa都是很值得注意的例外。[1]

那些需要快速存取或者反應時間非常一致的嵌入式系統，和其他的特殊應用的電腦系統，可能會為了避免讓運算結果的可預測性降低，而選擇不使用虛擬記憶體噢。

問題是

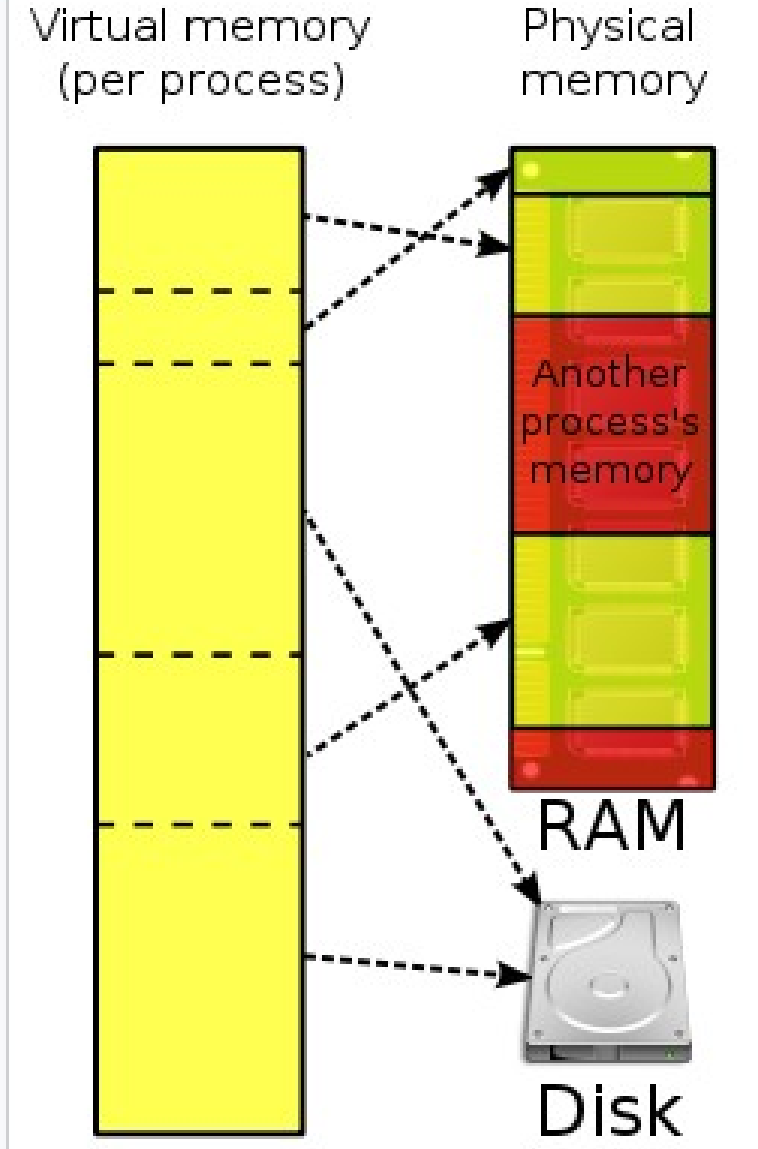
- 初學者看到之後，會懂嗎？

這讓我想起

- 我自己大學時候也搞不清楚
- 反覆唸了好幾遍課本才有點懂

為甚麼不直接說

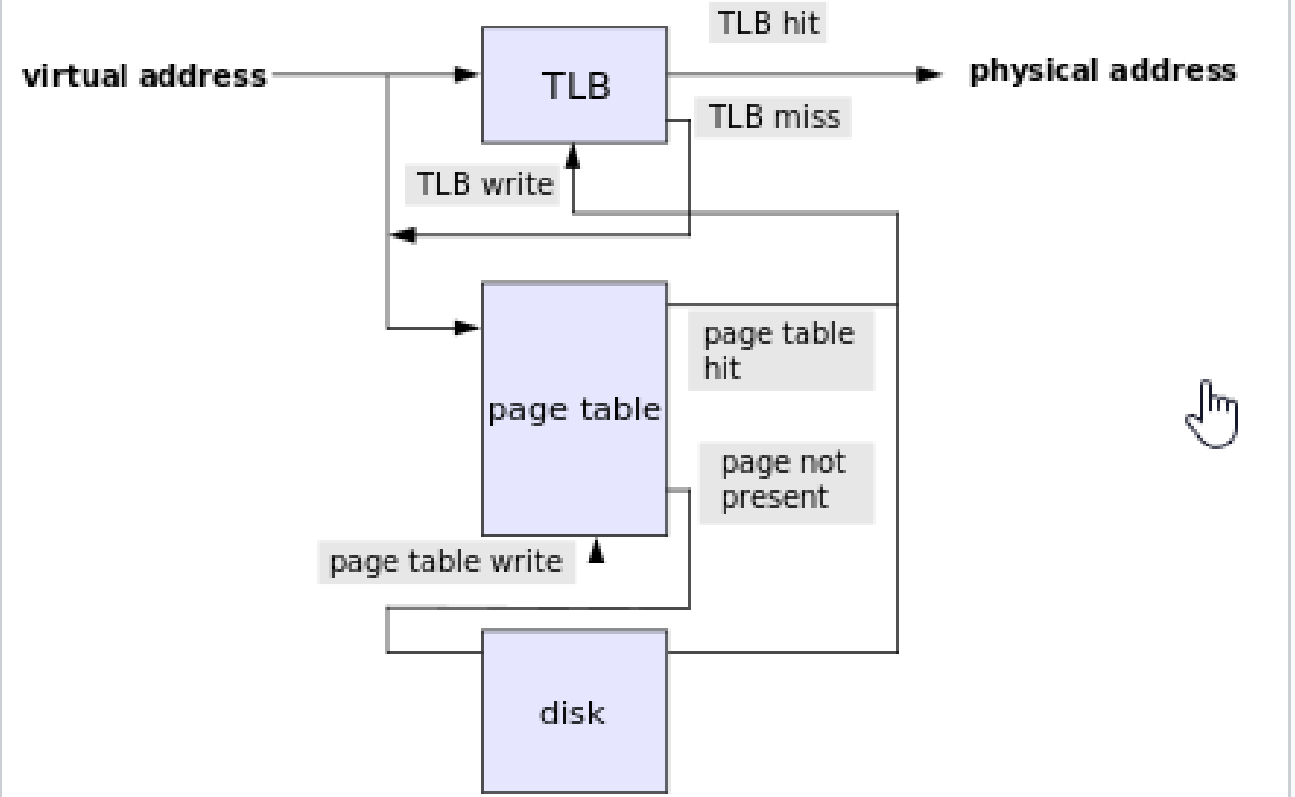
- 虛擬記憶體，其實就是把硬碟當成超大的延伸記憶體使用。
- 只是需要在 CPU 的硬體上，加上一些電路，才能完成這個功能。



Virtual memory combines active RAM and inactive memory on DASD[NB 1] to form a large range of contiguous addresses.

而那些延伸電路

- 主要就是用來存取分頁分段表的電路
- 特別是 TLB (Translation lookaside buffer)



Actions taken upon a virtual to physical address translation. Each translation is restarted if a TLB miss occurs, so that the lookup can occur correctly through hardware.

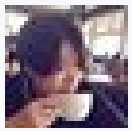
我們為了精確描述

- 有時反而會把一個簡單的概念講得很複雜，這樣讀者反而會被誤導！
- 特別是在讀原文書，英文又不夠好的情況下。

不過上面的描述

- 太過精簡可能會有不精確的情況

所以請參考網友回覆：



Jim Huang Page 16: 虛擬記憶體不是專門為了做 swap memory 的用途，注意到 1970 年代的 PDP-11 裡頭的 virtual memory 的設計致使「可定址的虛擬記憶體可能比實體記憶體來得小」

收回讚 · 回覆 · 15 · 9小時

吳孟徽 心有戚戚焉...

不過virtual memory那樣解釋怪怪的。我認為它主要是用來加大定址空間，避開physical memory的局限。慢速儲存媒介並不是必要，就算沒有硬碟，它也可以運作。

再舉一個例子

- 很多對硬體比較有注意的同學，會聽到《超頻》這件事情。
- 但是在一知半解下，會把《超頻》想得非常神秘。

但超頻其實就是

- 把 clock 線的頻率調高
  - 內頻： CPU 的 clock 頻率
  - 外頻：周邊設備的工作頻率

內外頻都可以調高，讓電腦跑得更快

這就是超頻！

（只是要小心太快會導致錯亂或燒掉）

但問題是

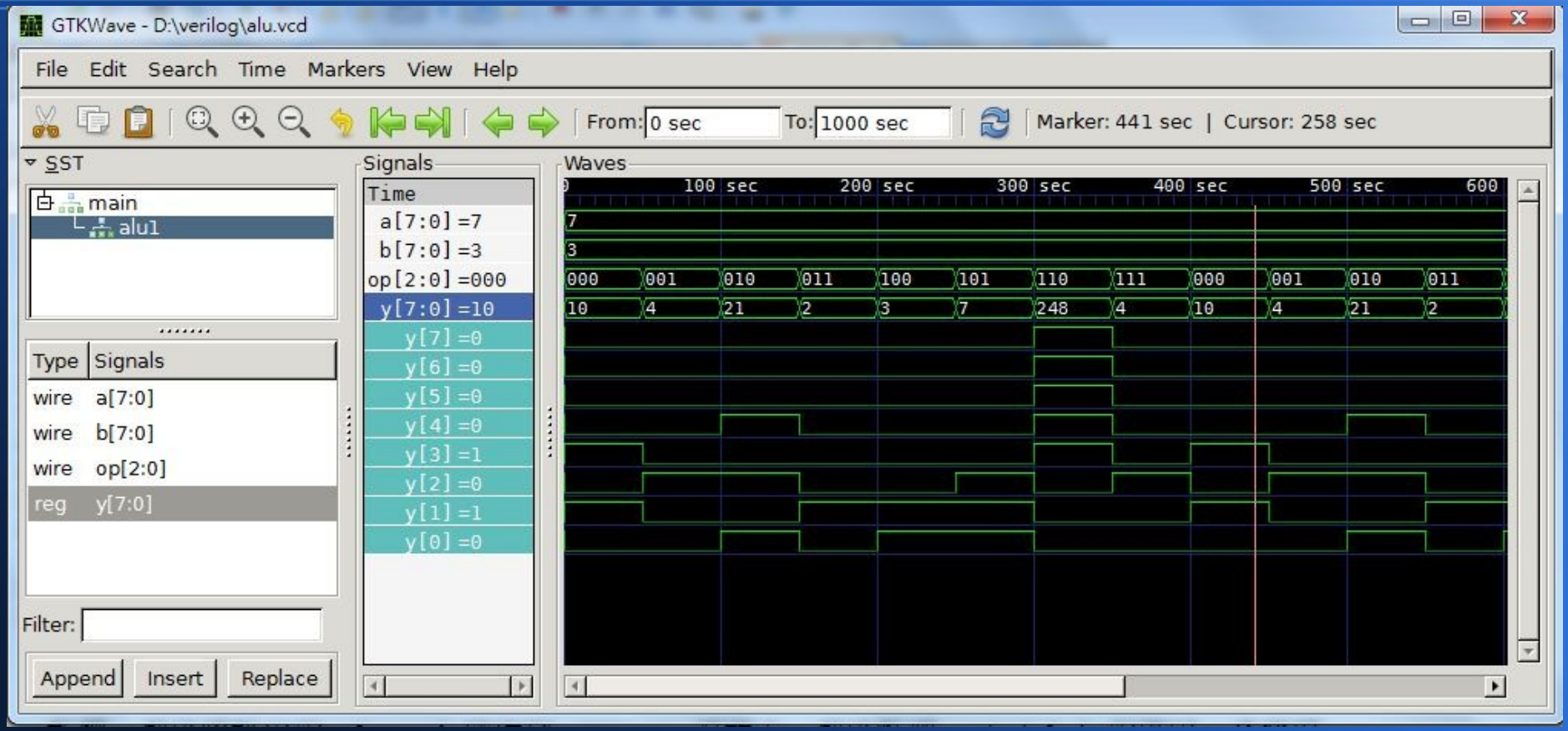
- 同學們常常對《數位邏輯》裡的 clock 線沒有概念
- 導致對超頻的理解常常充滿幻想 …
- 其實我在大學的時候也是這樣 . . .

如果同學們知道

- 邊緣觸發正反器中 clock 線的用途
- 應該就可以理解 clock 的同步概念
- 然後就可以輕易的理解上述的超頻現象了

但是 clock 的概念

- 對熟悉電路波形圖的人或許很簡單



但對資工領域的某些學生卻很陌生！

有了《邊緣觸發》的概念

- 才比較能理解《管線處理器》 pipeline 結構到底是怎麼回事！
- 但是課本通常不會談兩者的關聯性。
- 以及《暫存器牆》是如何用來區隔管線各個階段的。

相反的

- 課本通常會從《精簡指令集》RISC開始談起。
- 於是學生開始搞不懂《精簡指令集》與管線之間的關係。

其實管線架構

- 才是現代處理器的重點
- 而不是精簡指令集。
- 雖然指令集精簡之後管線的設計會比較容易。

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

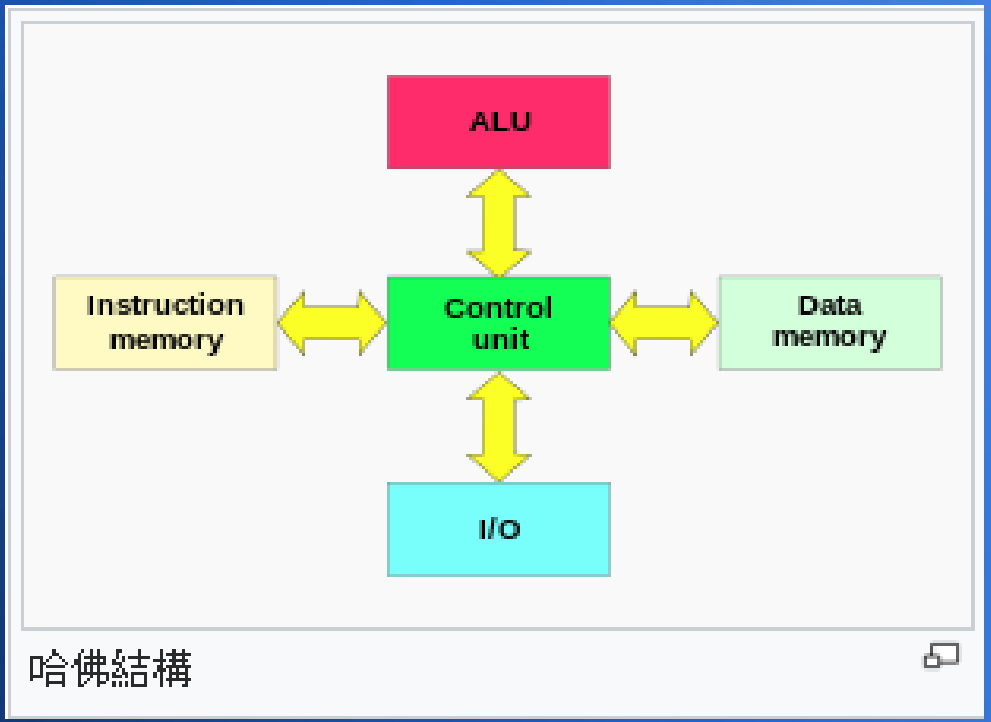
RISC機器的五層管線示意圖（IF：讀取指令，ID：指令解碼，EX：執行，MEM：記憶體存取，WB：寫回暫存器）

由於管線架構

- 要盡量避免記憶體存取（因為記憶體存取通常比暫存器慢很多）
- 這樣才能讓管線很順暢的一個接著一個，不會有泡泡產生。
- 因此管線處理器通常會採用哈佛架構。
- 就算沒有用雙記憶體，也通常會用雙快取

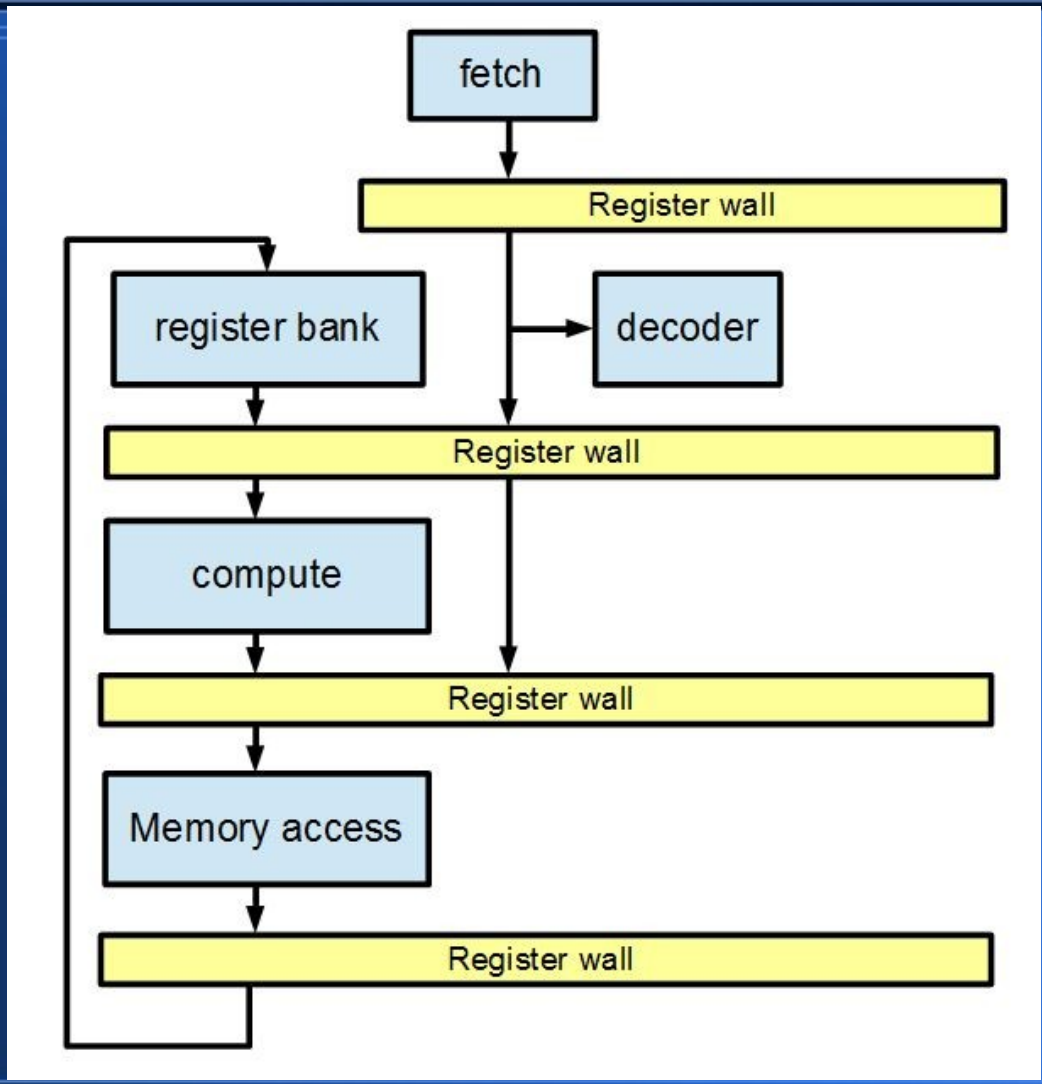
管線處理器

- 通常把指令和資料分在兩個記憶體或快取中。
- 這種指令和資料分開在兩套記憶區的模式，就偏向了《哈佛架構》。



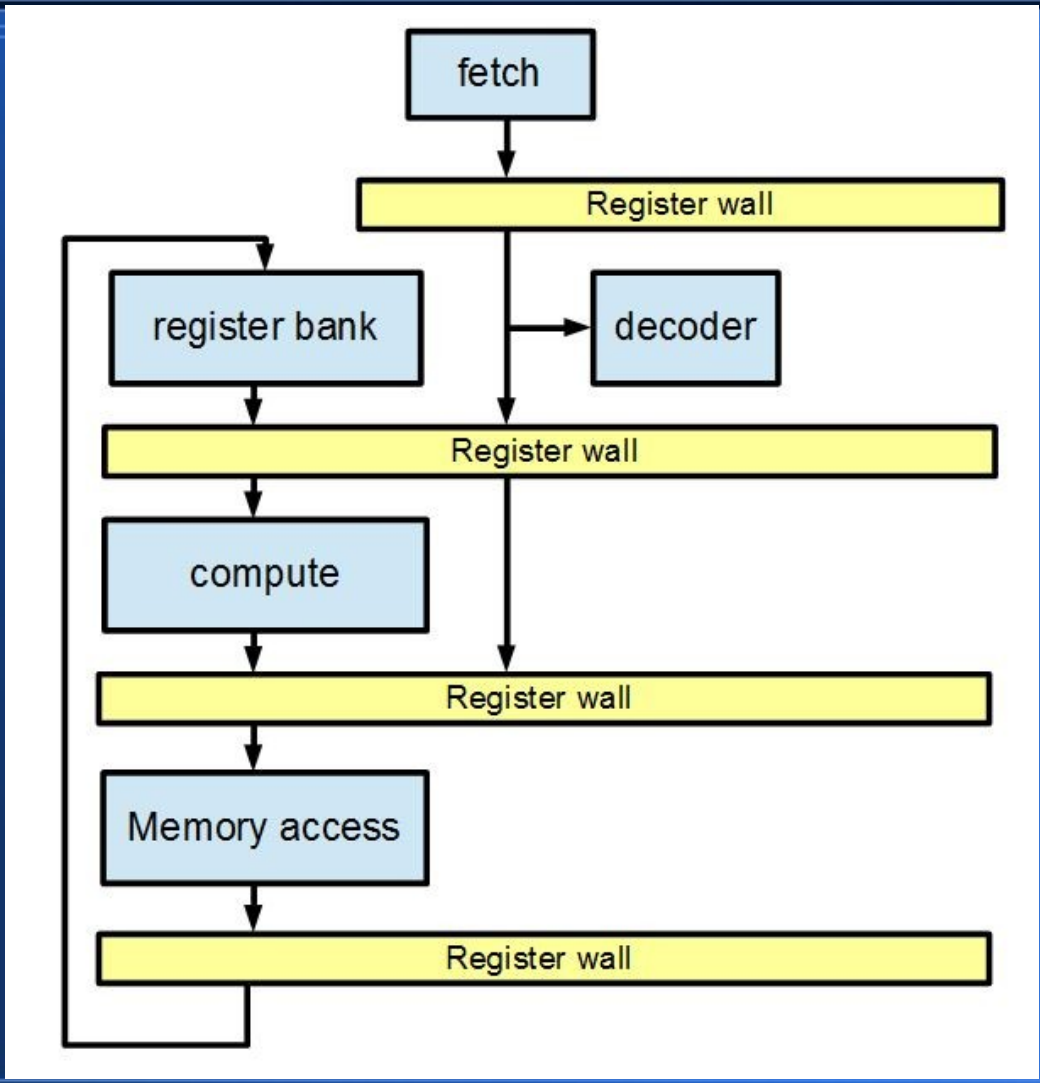
管線處理器利用暫存器牆

- 將《提取、解碼、執行、存取、寫回》等階段隔開
- 一顆 CPU 可以同時執行不同階段。
- 於是整顆 CPU 的各部分都充分的使用



暫存器牆

- 之所以可以隔絕各個階段的電路
- 正是因為《邊緣觸發》的特性所導致的。



而沒有用管線的傳統處理器

- 通常只有一小部分的電路在工作，其他部分會處於閒置狀態。
- 但是閒置還是要耗費電力的，所以會比管線處理器耗費更多的電，能源效率就比較差！

這些特點

- 才是管線處理器之所以興起的原因。
- 所以重點並不在精簡指令集，而是管線架構。

理解了管線技術

- 就能理解 MIPS 與 ARM 這類管線處理器為何在手機上有優勢
- 也能對 Intel x86 複雜指令集架構的包袱有所認識
- 其實 Intel 也想辦法引入了管線到後來的 x86 處理器當中，但卻是在管線的精簡指令上架構出複雜指令集的結構。

希望這樣的解釋

- 能讓您對管線處理器有清楚的認識

另外

- 有很多同學，會搞混《同步》這個詞彙的意義

像是

- 電路的同步與非同步
- 程式的同步與非同步

電路的同步

- 代表大家一起按照 clock 的頻率步伐來運作
- 而非同步則反之！

程式的同步呼叫

- 其實是按照順序一行一行執行下來
- 而非同步呼叫則通常會回呼 callback
- 非同步在 node.js/javascript 裏大量使用，而且通常是採用回呼的方式進行的。

網友的回覆
糾正與補充

像是這樣

檔案讀取

檔案：readfile.js

```
var fs = require('fs'); // 引用檔案物件
var data = fs.readFileSync(process.argv[2], "utf8"); // 讀取檔案
console.log(data); // 顯示在螢幕上
```

檔案讀取（非阻斷回呼型）

檔案： readfileCallback.js

```
var fs = require('fs'); // 引用檔案物件
fs.readFile(process.argv[2], "utf8", function(err, data) {
  console.log("data="+data);
});
console.log("----readFile End-----"); // 顯示在螢幕上
```

同步呼叫
Synchronous
(Blocking)

非同步呼叫
Asynchronous
(NonBlocking)

雖然兩者都是同步與非同步

- 但是意義卻差了很多！

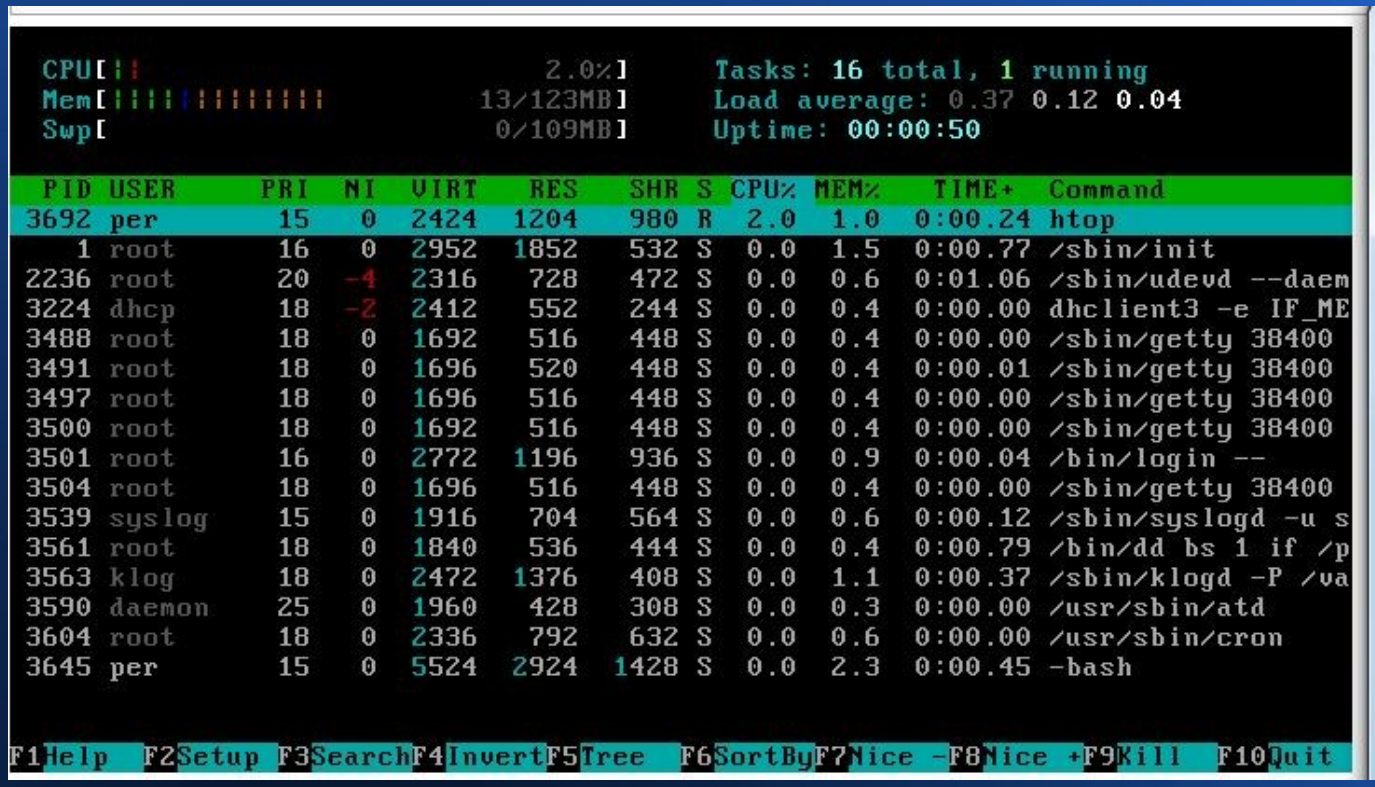
再來是作業系統裏

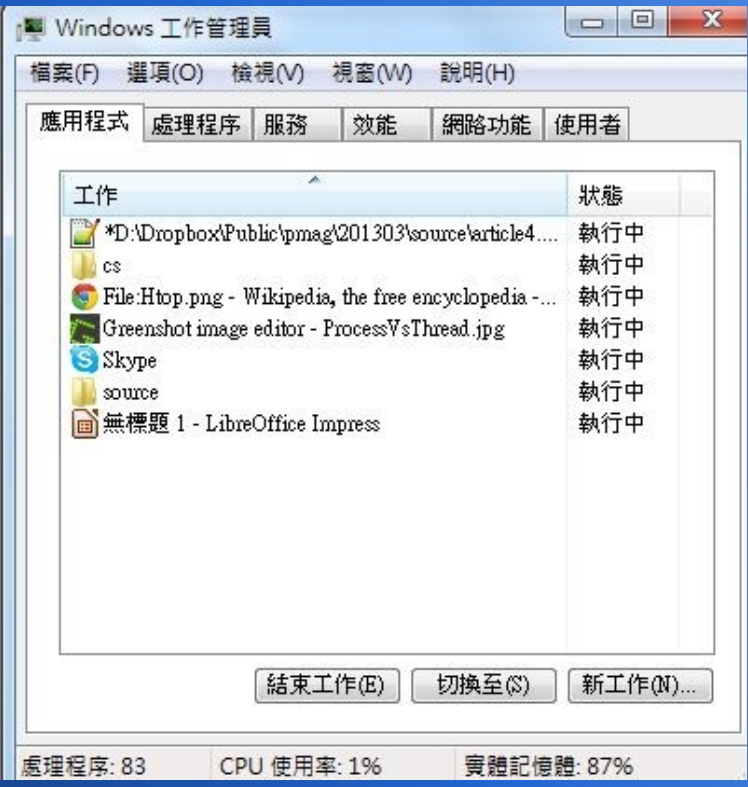
- 那個神祕的 Thread 與 Process ，也就是《執行緒》和《行程》之間的關係。

這兩者我想在怎麼用文字描述

- 都是很難懂的！
- 但是用程式和實例就比較容易懂。

Process 就是你在檔案總管裏看到一個一個執行中的程式





而 thread 則是一個程式裏可以一次執行好多個 thread 函數

```c
int main(void)
{
        const char *str1 = "Task1", *str2 = "Task2", *str3 = "Task3";

        usart_init();

        if (thread_create(test1, (void *) str1) == -1)
                print_str("Thread 1 creation failed\r\n");

        if (thread_create(test2, (void *) str2) == -1)
                print_str("Thread 2 creation failed\r\n");

        if (thread_create(test3, (void *) str3) == -1)
                print_str("Thread 3 creation failed\r\n");

        /* SysTick configuration */
        *SYSTICK_LOAD = (CPU_CLOCK_HZ / TICK_RATE_HZ) - 1UL;
        *SYSTICK_VAL = 0;
        *SYSTICK_CTRL = 0x07;

        thread_start();

        return 0;
}
```

```c
static void delay(volatile int count)
{
        count *= 50000;
        while (count--);
}

static void busy_loop(void *str)
{
        while (1) {
                print_str(str);
                print_str(": Running...\n");
                delay(1000);
        }
}

void test1(void *userdata)
{
        busy_loop(userdata);
}

void test2(void *userdata)
{
        busy_loop(userdata);
}

void test3(void *userdata)
{
        busy_loop(userdata);
}
```

Thread 和 Process

- 都可以在作業系統或執行環境的安排下交錯的執行，但只有 thread 可以共用變數。
- 單看文字敘述是很難理解兩者之區別的，但執行過這類程式的人很容易就會了解 thread 與 process 的概念！

另外像是死結的概念

```
class ThreadTest
{
    static String A = "A";
    static String B = "B";

    public static void Main(String[] args)
    {
        Thread thread1 = new Thread(AB);
        Thread thread2 = new Thread(BA);
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
    }
```

```
public static void AB()
{
    lock (A)
    {
        Console.WriteLine("AB.lock(A)");
        Thread.Sleep(1000);
        lock (B)
        {
            Console.WriteLine("AB.lock(B)");
        }
    }
}

public static void BA()
{
    lock (B)
    {
        Console.WriteLine("BA.lock(B)");
        Thread.Sleep(1000);
        lock (A)
        {
            Console.WriteLine("BA.lock(A)");
        }
    }
```

只要有了範例程式的體驗

- 通常也就可以清楚的理解了！
- 這些都是很難光看文字，就能深入體會其意義的。

希望

- 這些想法能對您有所幫助！

這就是我們

- 今天的十分鐘系列！

希望你會喜歡！

我們下回見！

Bye Bye!