

並行計算

EP01: 開場白

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

本系列的主要目標

- 這是本人英文版 *Concurrent Computing* 的中文 **精簡版**。
- 重點是在提供對並行計算的全盤理解。
- 一般而言，作業（或操作）系統課程中會有一些相關內容，但本系列的內容更廣泛、更完整、更深入。
- **預備知識**：能寫程式（當然）、一點硬體和系統操作知識、能夠做邏輯論證、敢於跳脫傳統程式寫作窠臼、等等。
- 程式寫作方面使用 **Unix**（或 **Linux**、**macOS** 等）。
- 請看本影片說明欄，該處有下載投影片、程式等內容的連結。

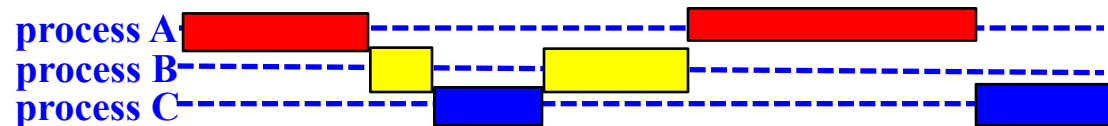
本集內容

- 交錯執行、平行、並行的差異
- 歷史上的重要事件
- 前庭和後庭
- 把程式送到後庭執行： `&`
- `ps` 和 `top` 命令
- 管道 (**pipe**)： `|`
- **Unix**的額外命令
- 重導I/O

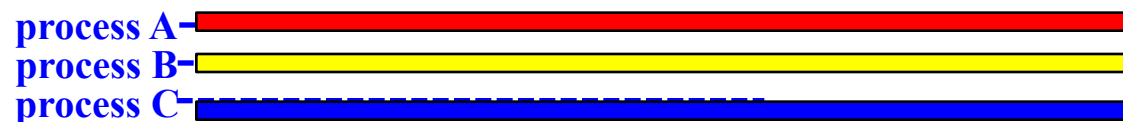
並行 vs. 平行: 1/3

- 正在執行的若干 **process** 是在

❖ **交錯執行 (interleaved)** : 這表示每一個 **process** 在任何時刻都有所進展, 但並非所有都在執行。



❖ **平行執行 (parallel)** : 這表示所有 **process** 都同時在執行。



❖ **並行執行 (concurrent)** : 這表示這些 **process** 是在交錯執行或平行執行。

並行 vs. 平行: 2/3

- **平行**執行程式的**process**或**thread**數目和**CPU**或核心 (**core**) 一樣多，每一個**process**或**thread**都佔用一個**CPU**或**core**，使得所有**process**或**thread**都能同時執行。
- 一個**並行**的程式，它的**process**或**thread**的數目通常遠大於系統中**CPU**或**core**總數，於是未必能同時執行，雖然有時候有此可能。
- 所以，**並行**的觀念比**平行**的觀念更加廣義。
- **為什麼?** 程式執行時，如果系統有足夠多的**CPU**或**core**使程式所有**process**或**thread**能同時行，於是程式就在**平行**狀況；但若**CPU**或**core**不足，系統會讓**某些****process**或**thread**執行很短時間後把**CPU**或**core**交給其它**process**或**thread**使用。於是，在任一時刻，每一個**process**或**thread**在執行上都會**有些進展**，在它們到達終點前是做一下、停一下、再做一下（斷斷續續）的。

並行 vs. 平行: 3/3

- 這個世界的活動其實是相當**並行**化的。
- 您一邊看球賽一邊吃東西（**平行**）。
- 您一邊聽音樂、一邊和同伴聊天、一邊發短訊（**平行或並行**）。
- 您一邊開車一邊發短訊，這肯定是**並行**，因為您真的無法一心兩用，必須看看路況、打幾個字、再看看路況、打幾個字，於是開車（看路）和發短訊是交錯進行的（所以是**並行**）。
- 您想想電腦系統中有哪些事是並行的呢？算是習題吧！

一些歷史事件的說明: 1/6

- 一切得從作業系統設計開始。
- 如果程式正在做輸出輸入 (I/O)，程式指令的執行必須停止、直到輸出輸入完成後才能繼續。**為什麼?**
- 若系統中只有一個程式，於是這個程式在進行輸出輸入時，**CPU**就在停頓的狀態。
- 幾十年前，**I/O**設備的速度很慢，雖然**CPU**的速度也不快，然而**CPU**卻很貴。所以，一旦程式在做**I/O**時，就等於在浪費（買**CPU**的）錢。

一些歷史事件的說明: 2/6

- 於是，為什麼不在一個程式做I/O時讓另一個程式佔用CPU執行呢？
- 因此，當程式1在做I/O時，程式2可以執行；程式2做I/O時，程式1可以執行。於是在任何時刻都有所進展。這不是並行的觀念嗎？
- 若我們可以在系統中執行兩個程式，執行更多個程式應該也不是問題。但是，**CPU的能力是有限的，系統無法承受太多個程式並行執行的負擔。**

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

一些歷史事件的說明: 3/6

- 在1960年代，作業系統可以執行若干個程式（復程式作業，**multiprogramming**）。
- 若一個系統可以同時執行若干個程式，為什麼我們不讓同一個程式分割成幾個部分（**process**或**thread**）而並行地執行呢？
- 於是，系統可以同時執行若干個程式，而每個程式也可以分成若干可以並行執行的**process**或**thread**，並行程式寫作於焉而生。
- 這就是本系列影片（講堂）的主題。

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

一些歷史事件的說明: 4/6

- 在**1960**年代，若干高階程式語言提供了並行程式寫作的的能力。
- **IBM PL/I F**和**ALGOL 68**是最先能提供這種能力的語言。
- 然後，**Concurrent Pascal**、**Modula 2**和後來的**Modula 3**，以及**Ada**、**Concurrent Euclid**、**Turing Plus**，等等都有此能力。**Java**是後進，更新的**C++**標準也支援並行處理。
- 在**1990**年代，並行程式寫作正式起飛。

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

一些歷史事件的說明: 5/6

- 您可能對並行計算的能力很興奮，但不要忘了它的代價。因為，**寫作一個好的、有並行能力的程式並不容易。**
- 因為，在此情況下**process**和**thread**必須相互溝通，溝通如果做不好，問題就大了。譬如說，若我打電話給您請提供若干資訊，若您沒能及時得到我的請求，我該繼續嗎？再說，等您看到我的請求時，要回覆嗎？因為請求可能已經過期。這可是大問題。
- 這就是**同步 (synchronization)** 問題，一旦到了這一步，您很可能就會放棄（因為很難），不過我建議您撐下去。😬

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

一些歷史事件的說明: 6/6

- 不但同步很難，把一個程式正確地切成很多部分也不容易，因為不正確切割的並行程式可能比一個不切割的程式的效率更差。
- 於是，在學習並行程式寫作時最好忘掉您之前學過的東西、讓您有個全新的開始。
- 一個並行程式的行為是動態的。這指的是您在執行一個並行程式時似乎沒有問題，但換了另一台機器或是寄給朋友在不同系統下執行時卻問題百出；也有可能是在您系統中執行出了問題，但在他人系統上卻毫無困難也不出現問題。
- 沒有任何偵錯程式（**debugger**）可以解決這個動態型的問題。我們在未來會學到很多。

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

讓我們試一試並行的滋味：1/7

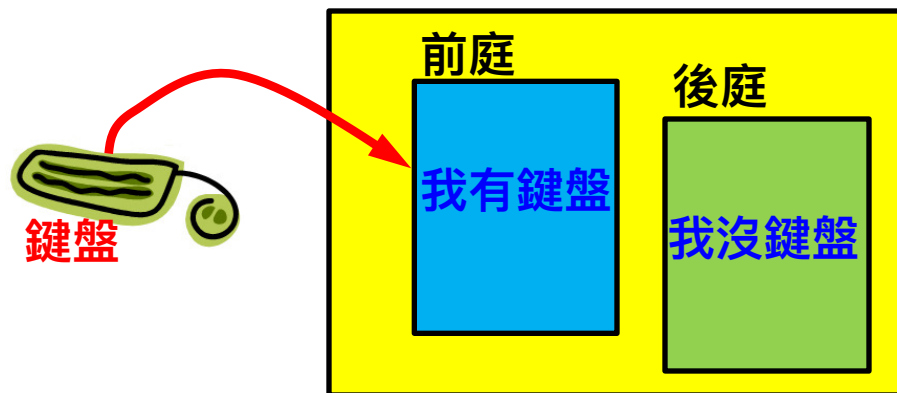
- 事實上，您每天都在這樣做的。
- 您在電腦前坐下來，打開幾個視窗、每一個視窗都執行一個應用程式（譬如**Chrome**、電郵、**Photoshop**或**Lightroom**、**PowerPoint**等等）。
- 這些程式都在執行，若視窗夠多、多到超過您電腦中的**CPU**和**core**的數目，這些程式就不可能同時執行，而變成並行了。
- 讓我們看一個很簡單的例子。

讓我們試一試並行的滋味: 2/7

- **Unix**命令列上可以用一個運算**&**。
- 這個**&**不是**C**或**C++**中的數元（或位元 — **bit-wise**）運算，而是把一個程式送到後庭（**background**）去執行。
- 當您用**Unix**命令列執行一個程式時，它就變成一個**process**（日後會解釋它）。
- 這個**process**從鍵盤取得輸入，此時程式會停止執行直到得到輸入後才會繼續。
- 正因為如此，一旦程式執行了，鍵盤成為該程式的標準輸入**stdin**，您就不能用鍵盤下達其它命令了，因為鍵盤的輸入都會被執行的程式讀取、而不會送到系統。

讓我們試一試並行的滋味: 3/7

- 把一個程式（更正確的說法是**process**）送到**後庭**（**background**）執行，簡單地說就是切斷了鍵盤做為**stdin**和該**process**之間的連繫，這樣鍵盤就可以下達操作的命令，而在**後庭**執行的**process**的**stdin**的輸入必須來自它處（譬如檔案）。
- 若**process**可以使用鍵盤做**stdin**輸入的就是在**前庭**（**foreground**）執行；打開一個命令列視窗後，只會有一個使用**stdin**讓鍵盤做為它的**前庭**程式。



讓我們試一試並行的滋味: 4/7

- 在要執行的程式名稱後面加上`&`、再按**Return**，就把該程式送到後庭執行，於是命令列就可以用來下達下一條命令：

```
a.out &
```

- 上面這道命令把`a.out`送到後庭執行，命令列的提示立即出現。
- 我們可以同時把若干個程式送到後庭執行，只要每一個程式名稱後面都加上`&`就行了。

```
a.out & dumb-prog & smart
```

- 於是`a.out`和`dumb-prog`在**後庭**執行，而`smart`則在**前庭**、它的`stdin`是鍵盤。
- 所有的這些程式都是**並行**執行的。

讓我們試一試並行的滋味: 5/7

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

#define LIMIT (20) // run this number of iterations

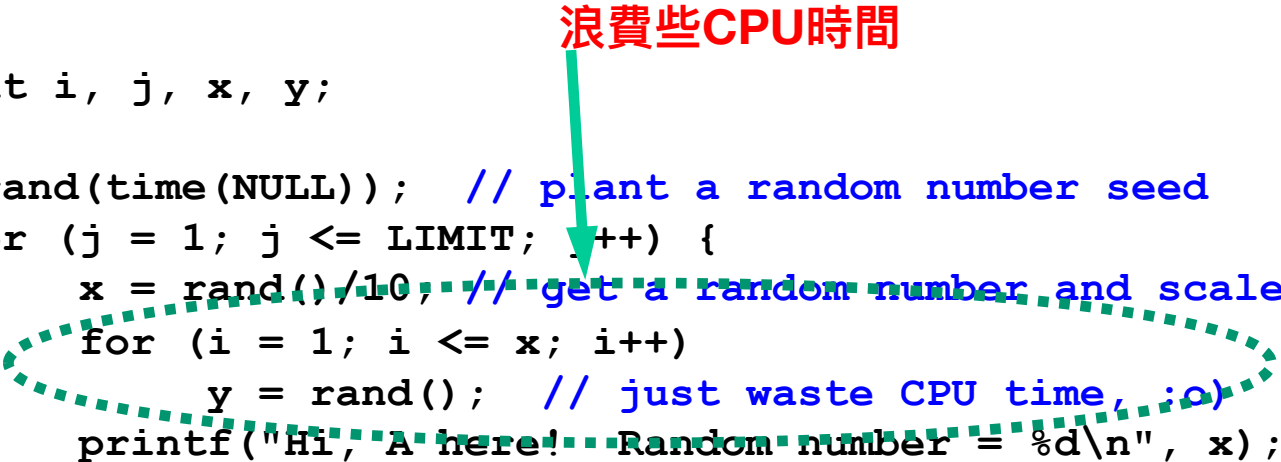
int main(void)
{
    int i, j, x, y;

    srand(time(NULL)); // plant a random number seed
    for (j = 1; j <= LIMIT; j++) {
        x = rand()/10, // get a random number and scale
        for (i = 1; i <= x; i++)
            y = rand(); // just waste CPU time, :o)
        printf("Hi, A here! Random number = %d\n", x);
    }
    printf("A completes\n");
}
```

procA.c

procA.c

浪費些CPU時間



讓我們試一試並行的滋味: 6/7

procB.c

```
#include <stdio.h>
#include <stdlib.h>pro

#define LIMIT (20)

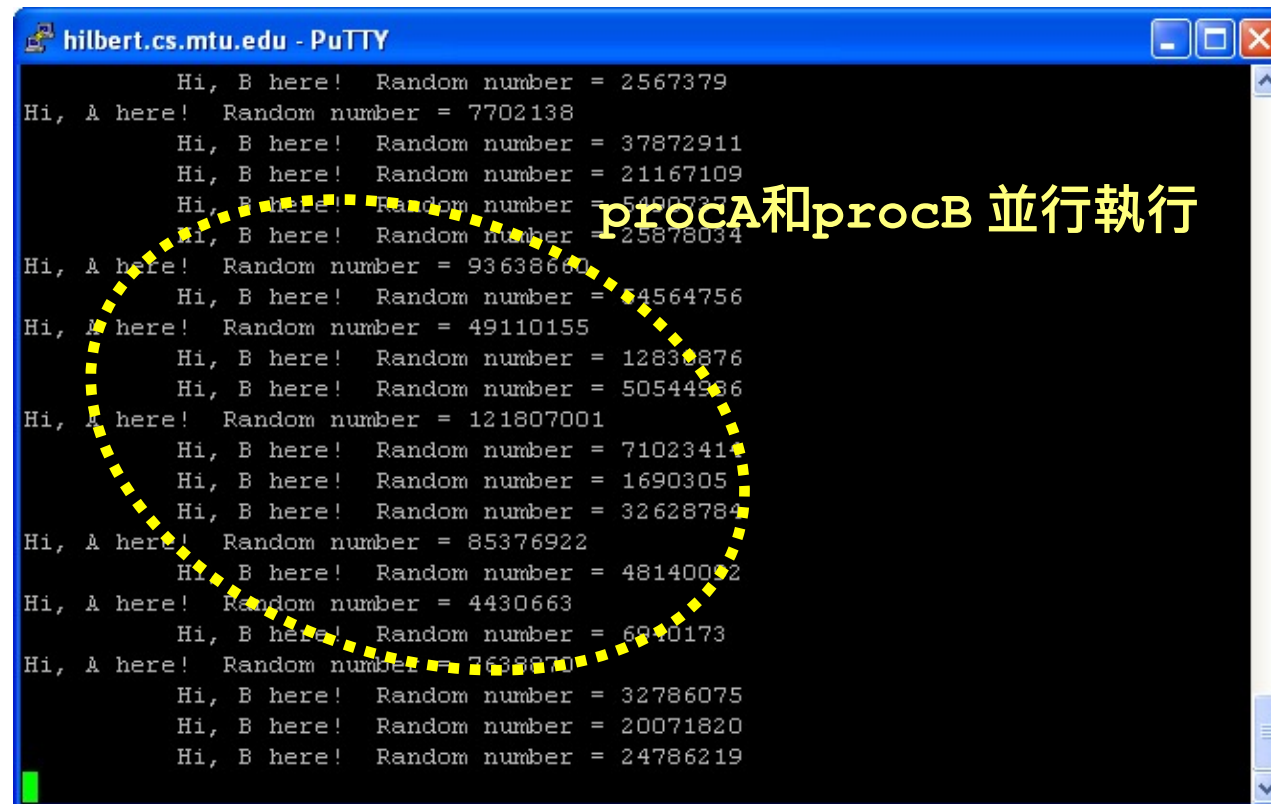
int main(void)
{
    int i, j, x, y;

    srand(time(NULL));
    for (j = 1; j <= LIMIT; j++) {
        x = rand()/30; // scaled differently
        for (i = 1; i <= x; i++)
            y = rand();
        printf("  Hi, B here!  Random number = %d\n", x);
    }
    printf("          B completes\n");
}
```

亂數的比例和procA.c的不同，
此地只有procA.c的1/3，所以procB.c
輸出的速度比較快。

讓我們試一試並行的滋味: 7/7

- 用procA & procB執行procA和procB。
- 哪個程式在前庭? 哪個程式在後庭?



```
hilbert.cs.mtu.edu - PuTTY
      Hi, B here! Random number = 2567379
Hi, A here! Random number = 7702138
      Hi, B here! Random number = 37872911
      Hi, B here! Random number = 21167109
      Hi, B here! Random number = 5498727
      Hi, B here! Random number = 25878034
Hi, A here! Random number = 93638660
      Hi, B here! Random number = 54564756
Hi, A here! Random number = 49110155
      Hi, B here! Random number = 12830876
      Hi, B here! Random number = 50544936
Hi, A here! Random number = 121807001
      Hi, B here! Random number = 71023414
      Hi, B here! Random number = 1690305
      Hi, B here! Random number = 32628784
Hi, A here! Random number = 85376922
      Hi, B here! Random number = 48140092
Hi, A here! Random number = 4430663
      Hi, B here! Random number = 6840173
Hi, A here! Random number = 7638870
      Hi, B here! Random number = 32786075
      Hi, B here! Random number = 20071820
      Hi, B here! Random number = 24786219
```

procA和procB 並行執行

我們再挖深一些

- 既然程式在執行時會變成**process**，那麼我產生了多少個**process**？
- **Unix**的 `ps` (**process status**) 命令就做這件事，它報告各**process**的狀態。
- 若 `ps` 不帶任何參數，它就只報告下命令人的**process**。
- 若加上合用的參數，`ps` 可以報告系統中所有**process**的狀態。

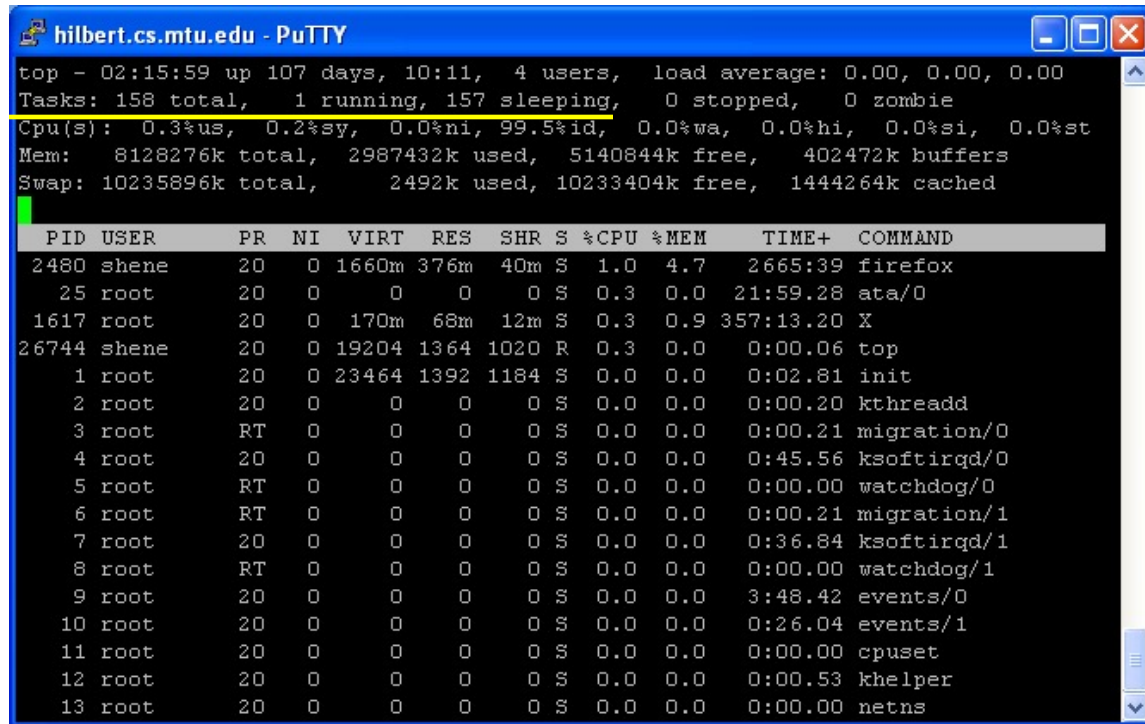
```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3331/Basics(48) ps
  PID TTY          TIME CMD
 28749 pts/2    00:00:00 tesh
 28821 pts/2    00:00:04 prog
 28822 pts/2    00:00:03 junk
 28823 pts/2    00:00:03 testing
 28825 pts/2    00:00:00 ps
HILBERT:/home/csdept/shene/CS3331/Basics(49)
```

這些是各個**process**的ID，也就是**PID**

這些是對應的程式名稱

誰吃CPU最兇？

- **Unix**的**top**命令是一個系統的監督工具，它可以顯示並且更新系統中資源的使用狀態，通常會以使用**CPU**的百分比自高到低的方式排列。



```
hilbert.cs.mtu.edu - PuTTY
top - 02:15:59 up 107 days, 10:11,  4 users,  load average: 0.00, 0.00, 0.00
Tasks: 158 total,  1 running, 157 sleeping,  0 stopped,  0 zombie
Cpu(s):  0.3%us,  0.2%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8128276k total, 2987432k used, 5140844k free,  402472k buffers
Swap: 10235896k total,  2492k used, 10233404k free, 1444264k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 2480 shene    20   0 1660m 376m  40m  S   1.0   4.7   2665:39  firefox
    25 root      20   0     0     0     0  S   0.3   0.0   21:59:28  ata/0
 1617 root      20   0  170m  68m  12m  S   0.3   0.9   357:13.20  X
26744 shene    20   0  19204 1364 1020  R   0.3   0.0    0:00.06  top
    1 root      20   0  23464 1392 1184  S   0.0   0.0    0:02.81  init
    2 root      20   0     0     0     0  S   0.0   0.0    0:00.20  kthreadd
    3 root      RT   0     0     0     0  S   0.0   0.0    0:00.21  migration/0
    4 root      20   0     0     0     0  S   0.0   0.0    0:45.56  ksoftirqd/0
    5 root      RT   0     0     0     0  S   0.0   0.0    0:00.00  watchdog/0
    6 root      RT   0     0     0     0  S   0.0   0.0    0:00.21  migration/1
    7 root      20   0     0     0     0  S   0.0   0.0    0:36.84  ksoftirqd/1
    8 root      RT   0     0     0     0  S   0.0   0.0    0:00.00  watchdog/1
    9 root      20   0     0     0     0  S   0.0   0.0    3:48.42  events/0
   10 root      20   0     0     0     0  S   0.0   0.0    0:26.04  events/1
   11 root      20   0     0     0     0  S   0.0   0.0    0:00.00  cpuset
   12 root      20   0     0     0     0  S   0.0   0.0    0:00.53  khelper
   13 root      20   0     0     0     0  S   0.0   0.0    0:00.00  netns
```

158個process

1個在執行

157個在休眠

使用CPU最高的是Firefox, 1%

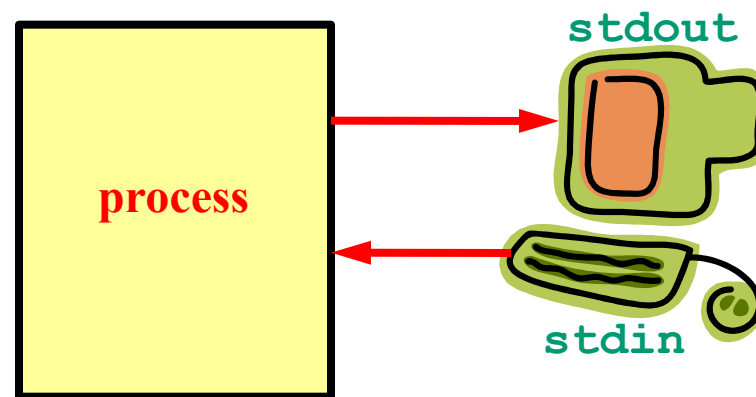
這些process都是並行的

加入一些合作: 1/10

- 上一個例子中的各個**process**各行其是而不需要其它**process**的幫忙，所以它們是相互**獨立** (**independent**) 自主的。
- 所以，它們是各自**獨立**的**process**。
- 若**process**之間必須要互相溝通來完成一項工作，這些**process**就不再獨立、而變成**合作** (**cooperating**) **process**。
- 獨立的**process**很容易處理，而合作的**process**就得有很仔細的**同步** (**synchronization**) 規劃。

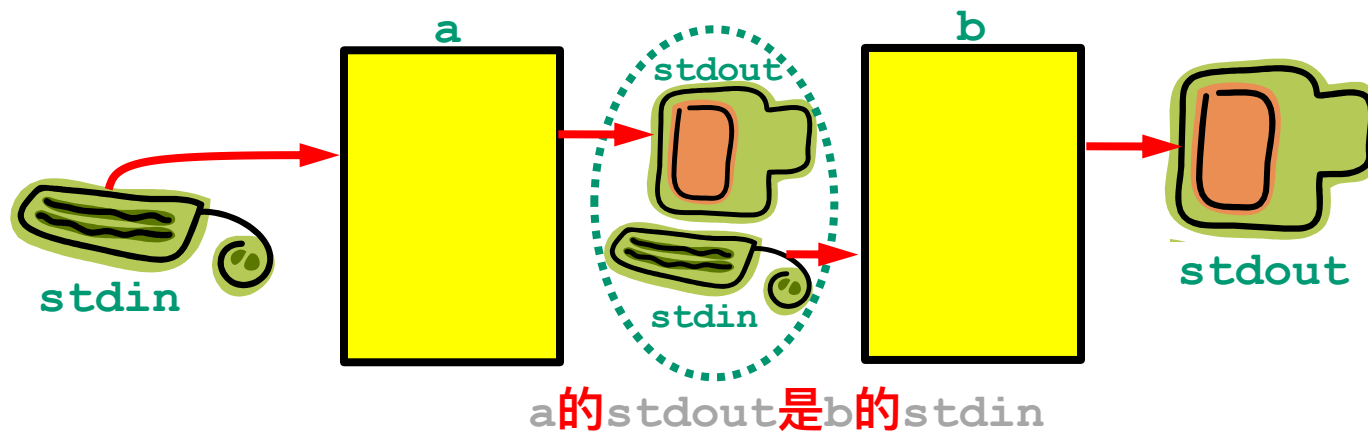
加入一些合作: 2/10

- 這是一個很簡單的合作**process**的例子。
- 我們使用**Unix**命令列的 `|` 運算。
- 在程執行時它有三個約定**I/O**: `stdin` (鍵盤)、`stdout` (顯示器、也就是執行該程式所用的視窗 — 更正確的說法是終端機**terminal**) 和`stderr` (用來輸出錯誤訊息, 未必是顯示器, 此地用不到它)。



加入一些合作: 3/10

- 若a和b是兩個可執行的程式，`a | b`表示a的stdout就是b的stdin。
- 這樣，a從它的stdin讀做資料，它的輸出透過stdout直接送到b的stdin去，而b的輸出則透過stdout顯示。我們說，把a的輸出透過**管道 (pipe)** | 送給b的輸入。



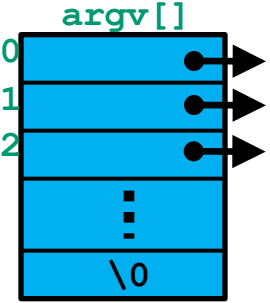
加入一些合作: 4/10

pA.c

```
#include <stdio.h>

int main(int argc, char **argv[])
{
    int    i, LIMIT;
    char  output[100];

    LIMIT = atoi(argv[1]); // read command line argument
    printf("%d\n", LIMIT); // print # of lines
    for (i = 1; i <= LIMIT; i++) { // print the lines
        sprintf(output, "Printing %d from A", i);
        printf("%s\n", output);
    }
}
```



The diagram shows an array named `argv[]` with four elements. The first three elements are indexed 0, 1, and 2, and each has a black arrow pointing to the right. The fourth element contains a vertical ellipsis and a backslash followed by zero (`\0`), representing the null terminator.

print to a character string

從命令列讀一個int並且印出那麼多列輸出

加入一些合作: 5/10

pB.c

```
#include <stdio.h>

int main(void)
{
    int i, LIMIT;
    char input[100];

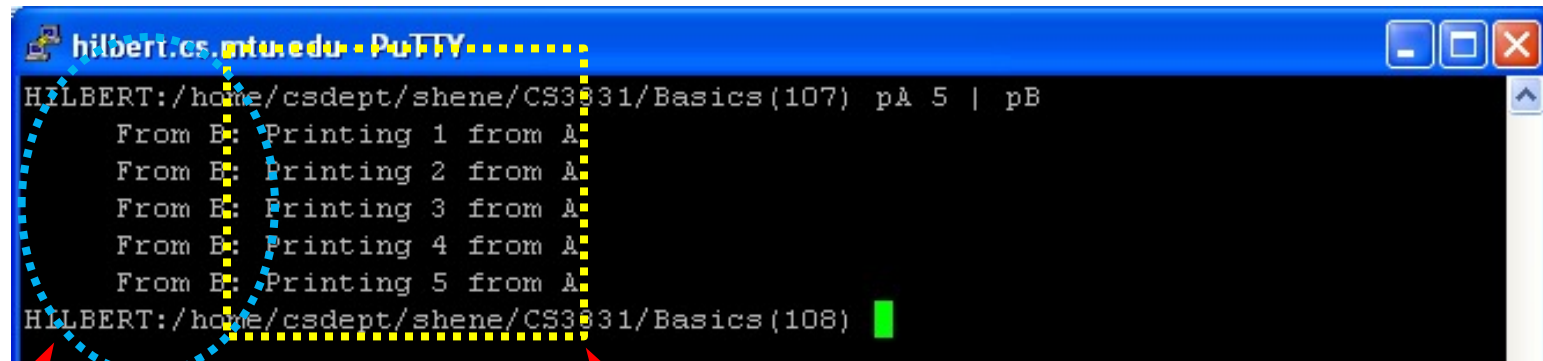
    gets(input); // read a complete input line
    LIMIT = atoi(input); // convert to integer
    for (i = 1; i <= LIMIT; i++) { // repeat
        gets(input); // read a complete input line
        printf("    From B: %s\n", input);
    }
}
```

第一個int指出得讀取那麼多列

使用gets()是很有風險的，因為讀入的char
數目很可能比預留的位置多

加入一些合作: 6/10

下面的輸出是pA 5 | pB 的結果
pA 印出5列



```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3931/Basics(107) pA 5 | pB
From E: Printing 1 from A
From E: Printing 2 from A
From E: Printing 3 from A
From E: Printing 4 from A
From E: Printing 5 from A
HILBERT:/home/csdept/shene/CS3931/Basics(108) █
```

pB 加上這部分

pB 從 pA 收到這個部分

加入一些合作: 7/10

pC.c

```
#include <stdio.h>

int main(void)
{
    int i, LIMIT = 100;
    char input[100];

    // now we use a better way: fgets()
    // keep reading until EOF
    while (fgets(input, LIMIT, stdin) != NULL)
        printf("From C: %s", input);
}
```

fgets() 比 gets() 安全

這是儲存輸入用 char[] 最長的輸入長度, 包含 \0 NULL 表示 EOF

加入一些合作: 8/10

下面是 pA 7 | pB | pC 的結果

pA 印7列

```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3331/Basics(116). pA 7 | pB | pC
From C:      From B: Printing 1 from A
From C:      From B: Printing 2 from A
From C:      From B: Printing 3 from A
From C:      From B: Printing 4 from A
From C:      From B: Printing 5 from A
From C:      From B: Printing 6 from A
From C:      From B: Printing 7 from A
HILBERT:/home/csdept/shene/CS3331/Basics(117)
```

pC 加上這部分

pB 加上這部分

pB 從pA收到這部分

加入一些合作: 9/10

- pA、pB 和 pC 並行執行。
- 執行 `pA 10000 | pB | pC`
- 當 pA、pB 和 pC 執行時，用 `ps -A` 得來的結果

這表示顯示所有 **process** (不同系統可能會有差異)

```
27435 pts/3    00:00:00 tssh
27475 ?          00:00:00 sshd
27480 ?          00:00:00 sshd
27481 pts/4    00:00:00 tssh
27518 pts/3    00:00:00 pA
27519 pts/3    00:00:00 pB
27520 pts/3    00:00:00 pC
27521 pts/4    00:00:00 ps
30442 pts/2    00:00:00 tssh
HILBERT:/home/csdept/shene/CS3331/Basics (28)
```

pA, pB 和 pC 並行執行

加入一些合作: 10/10

- 因為 `pB` 需要 `pA` 的輸出、而 `pC` 則需要 `pB` 的輸出，所以 `pA`、`pB` 和 `pC` 並非獨立而是合作的 **process**；它們的溝通方式是透過“連接”起來的 `stdin` 和 `stdout`，雖然這個溝通方式極為簡單。
- 我們後面還會講到 **process** 和 **thread** 之間更複雜的溝通方式。

幾道額外的Unix命令: 1/5

- **Ctrl-Z**: 這道命令暫停一個前庭程式，並且回到命令列的提示。

- **bg**: 把最近暫停的**process**送到後庭執行

`prog` // 執行叫做 `prog` 的程式 (當然是前庭)

Ctrl-Z // 暫停 `prog`

`bg` // 把 `prog` 送到後庭執行

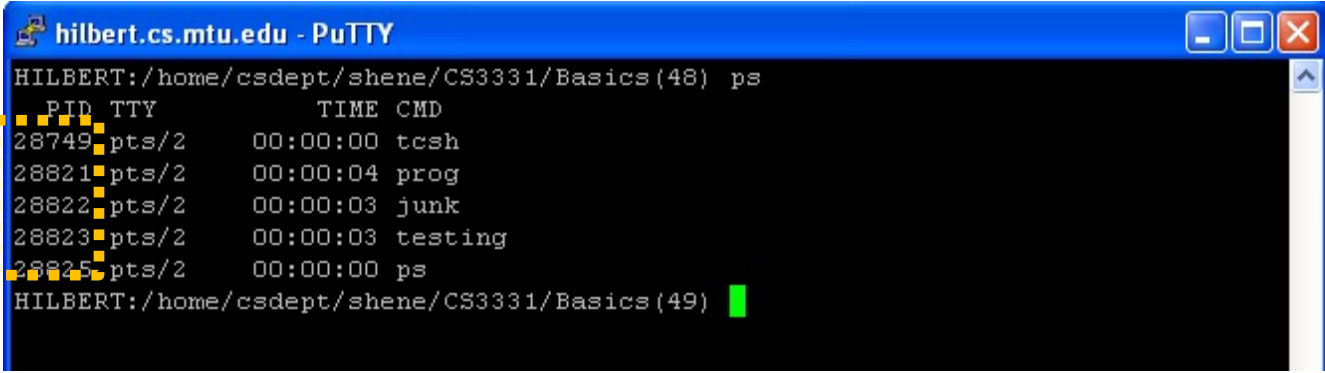
- **fg**: 把最近送到後庭的程式送回前庭執行 (它的`stdin`是鍵盤)

`fg` // `prog` 是在前庭執行

幾道額外的Unix命令: 2/5

- 每一個process都有一個系統賦於的ID，叫做**PID**。

process ID



```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3331/Basics (48) ps
  PID TTY          TIME CMD
 28749 pts/2        00:00:00 tcsh
 28821 pts/2        00:00:04 prog
 28822 pts/2        00:00:03 junk
 28823 pts/2        00:00:03 testing
 28825 pts/2        00:00:00 ps
HILBERT:/home/csdept/shene/CS3331/Basics (49) █
```

- **Unix**系統有不少從系統送給程式的**訊號**（**signal**）。
- 有一些訊號是不能忽略的（令到必行），另一些訊號則可以由程式處理（譬如不理會、或由程式自己處理而不是系統）。
- `kill`命令可以把這些訊號傳給一些程式。

幾道額外的Unix命令: 3/5

- 每一個**Unix**送給程式的訊號從**SIG**開始，後面跟著**3到4**個文字。**SIGKILL**就是終止程式執行的訊號，但**kill**命令中不需要輸入**SIG**這三個字母。

```
kill -KILL pid1 pid2 ... pidi
```

把**SIGKILL**訊號送到 `pid1` 、 `pid2`、`...`、`pidi`這幾個**process**，終止這幾個**process**執行。

幾道額外的Unix命令: 4/5

- `kill -KILL 28821 28823` 終止PID為28821 (程式prog) 和 28823 (程式testing) 的執行。

process ID

```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3331/Basics (48) ps
  PID TTY          TIME CMD
 28749 pts/2    00:00:00 tcsh
 28821 pts/2    00:00:04 prog
 28822 pts/2    00:00:03 junk
 28823 pts/2    00:00:03 testing
 28825 pts/2    00:00:00 ps
HILBERT:/home/csdept/shene/CS3331/Basics (49) |
```

- 這些SIG訊號都有一個整數編號，而SIGKILL的編號是9，所以 `kill -9 28821 28823` 和上面用KILL的命令列功能完全相同。

幾道額外的Unix命令: 5/5

- SIGINT相當於按下**Ctrl-C**鍵，`kill -INT 28821 28823` 相當於對28821和28823兩個**process** 都按下**Ctrl-C**鍵。
- SIGKILL是令到必行，程式不可以拒絕；而SIGINT在程式中可以處理，既可以不理會（這樣**Ctrl-C**根本沒有效用）、也可以讓程式做不同的反應（譬如顯示「**是否按錯鍵？輸入Y或y就停止執行**」等等）。
- SIGSTOP和SIGCONT分別暫停某個**process**的執行、和繼續執行一個被暫停的程式。
- 如何使用訊號不在本課程範圍內，或許日後可以補上這一段。

重導I/O: 1/3

- 一個**Unix**程式一定從`stdin`讀取文字輸入、並且把文字輸出送到`stdout`顯示。但是，我們可以改變`stdin`和`stdout`的來源或去處。
- `prog < data`: 表示程式`prog`的`stdin`改成檔案`data`，於是`prog`的輸入來自檔案`data`。當然，在執行`prog`之前，檔案`data`必須已經存在。
- `prog > report`: 程式`prog`把它的`stdout`輸出送到`report`檔案；在執行`prog`之前，檔案`report`不應該存在。

重導I/O: 2/3

- 請問下面兩道命令

```
pA | pB
```

和重導的I/O有何區別?

```
pA > temp-file
```

```
pB < temp-file
```

- 第一種方式之下，pA 和 pB 並行執行。
- 但是在第二種方式之下，temp-file在pB執行前就必須存在；換言之，pB必須等到pA執行完畢並且建成temp-file之前無法執行，因此pA先執行、做完之後pB才能開始，這不是並行的。

重導I/O: 3/3

- `stdin`和`stdout`都可以同時被重導到它處。
- 下面的命令在執行時，`prog`的輸入來自檔案`data`，而`prog`的輸出則是送到檔案`report`。

```
prog < data > report
```

我們學到了什麼？

- 交錯執行、平行和並行的意義
- 前庭執行和後庭執行：並行的開端
- 兩個**Unix**命令列的運算：& 和 |；它們都能立即產生並行的效果
- `ps` 和 `top` 命令
- 管道 (**pipe**)：|
- **Unix**和並行執行有關的其它（好用）命令
- 重導I/O

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