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

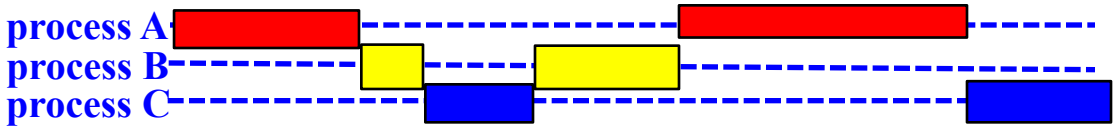
- 這是本人英文版***Concurrent Computing***的中文*精簡*版。
- 重點是在提供對並行計算的全盤理解。
- 一般而言，作業（或操作）系統課程中會有一些相關內容，但本系列的內容更廣泛、更完整、更深入。
- **預備知識**：能寫程式（當然）、一點硬體和系統操作知識、能夠做邏輯論證、敢於跳脫傳統程式寫作窠臼、等等。
- 程式寫作方面使用**Unix**（或**Linux**、**macOS**等）。
- 請看本影片說明欄，該處有下載投影片、程式等內容的連結。

本集內容

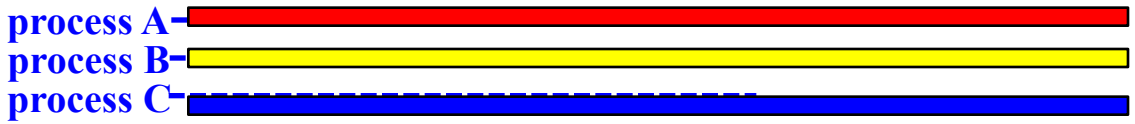
- 交錯執行、平行、並行的差異
- 歷史上的重要事件
- 前庭和後庭
- 把程式送到後庭執行：`&`
- `ps` 和 `top` 命令
- 管道（pipe）：`|`
- Unix的額外命令
- 重導I/O

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

- 不旦同步很難，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

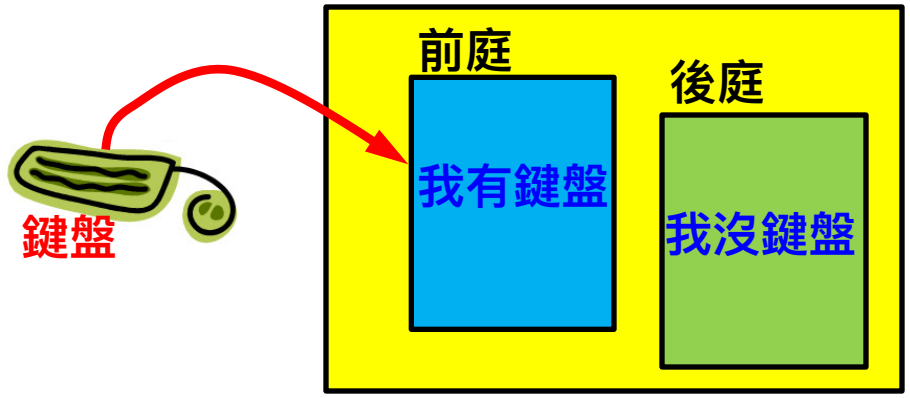
- 事實上，您每天都在這樣做的。
- 您在電腦前坐下來，打開幾個視窗、每一個視窗都執行一個應用程式（譬如Chrome、電郵、Photoshop或Lightroom、PowerPoint等等）。
- 這些程式都在執行，若視窗夠多、多到超過您電腦中的CPU和core的數目，這些程式就不可能同時執行，而變成並行了。
- 讓我們看一個很簡單的例子。

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

procA.c

```
#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

浪費些CPU時間

讓我們試一試並行的滋味: 6/7

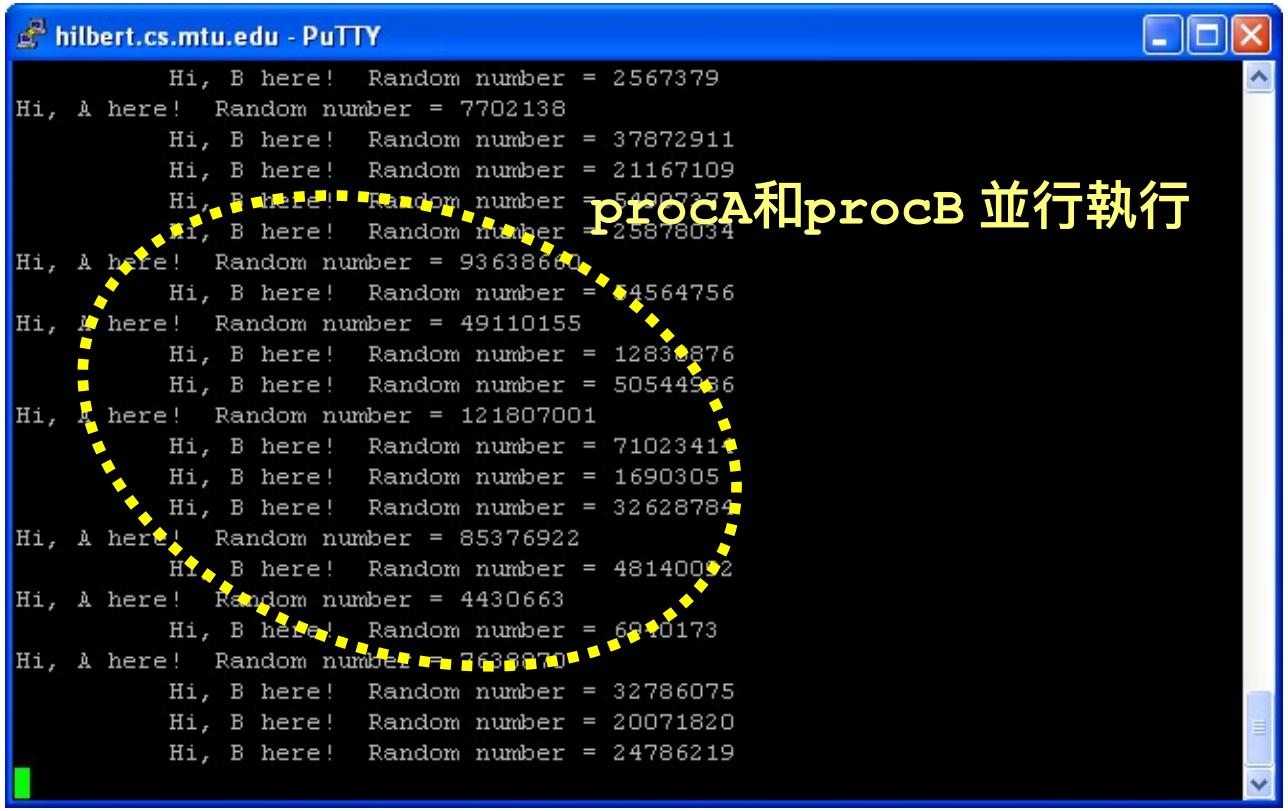
procB.c

```
#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。

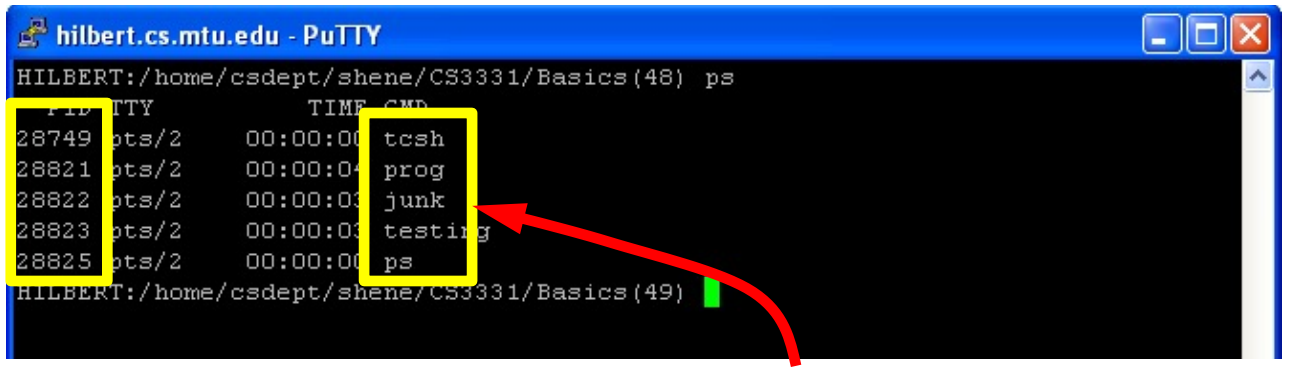
讓我們試一試並行的滋味: 7/7

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



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。

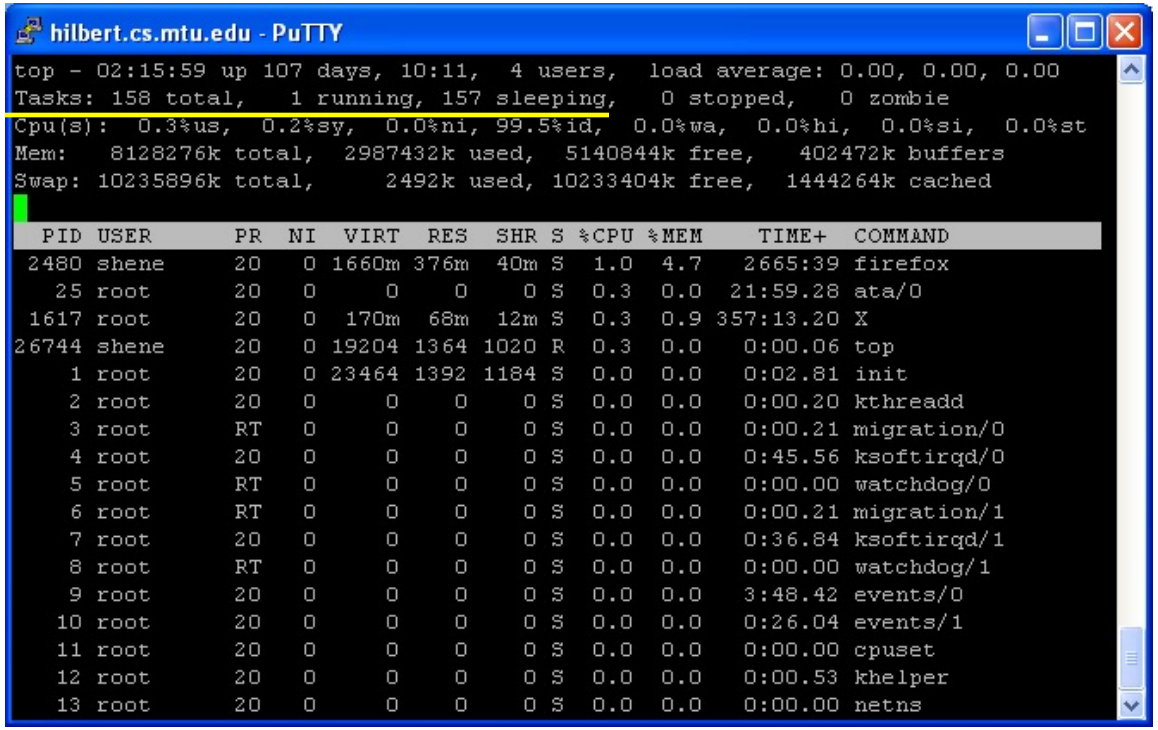


這些是各個**process**的**ID**，也就是**PID**

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。**



158個process

1個在執行

157個在休眠

使用CPU最高的是Firefox，1%

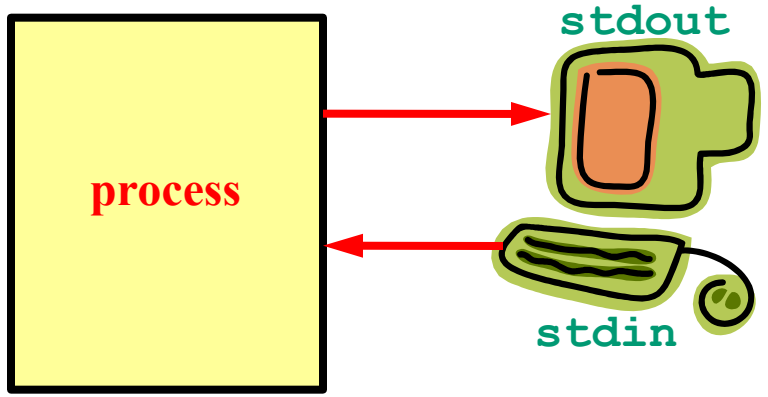
這些process都是並行的

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。

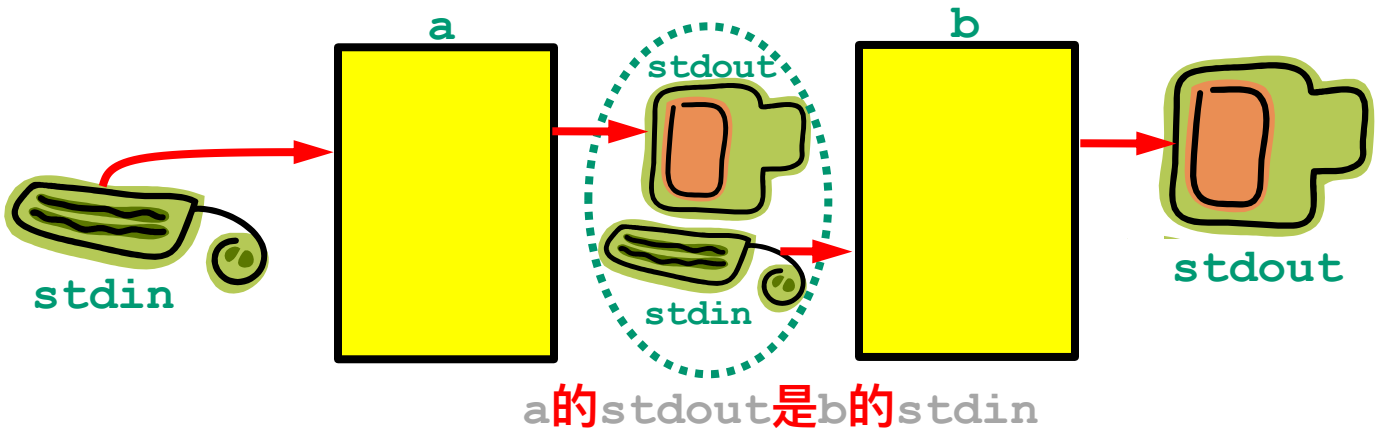
加入一些合作: 2/10

- 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

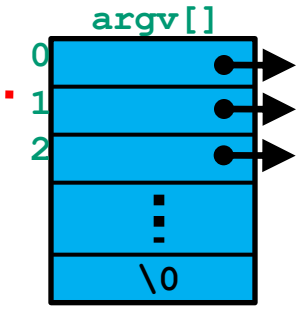
- 若a和b是兩個可執行的程式，a | b表示a的stdout就是b的stdin。
- 這樣，a從它的stdin讀做資料，它的輸出透過stdout直接送到b的stdin去，而b的輸出則透過stdout顯示。我們說，把a的輸出透過管道（pipe）| 送給b的輸入。



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);
      }
}
```



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);
     }
}
```

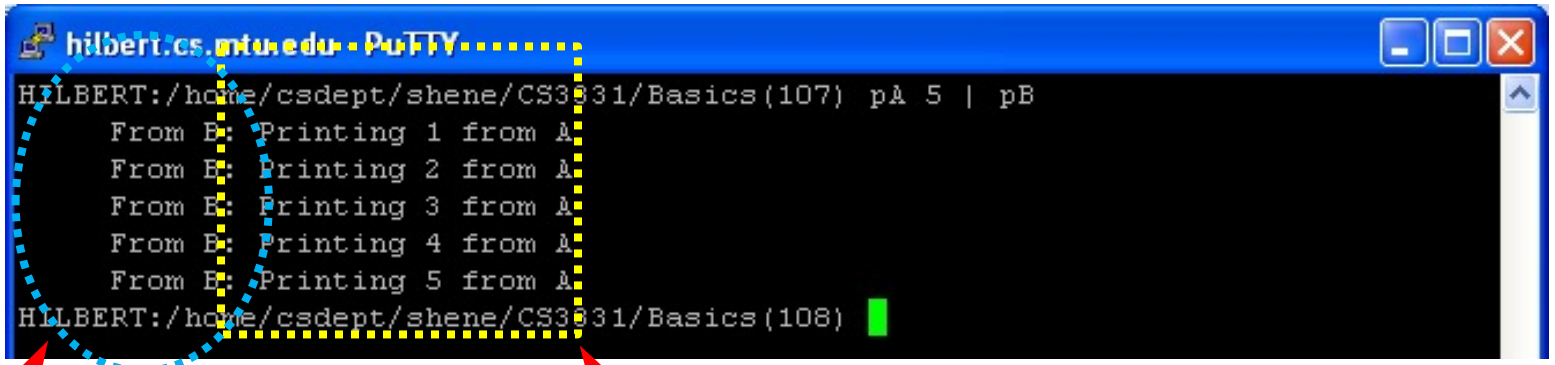
第一個int指出得讀取那麼多列

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



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);
}
```

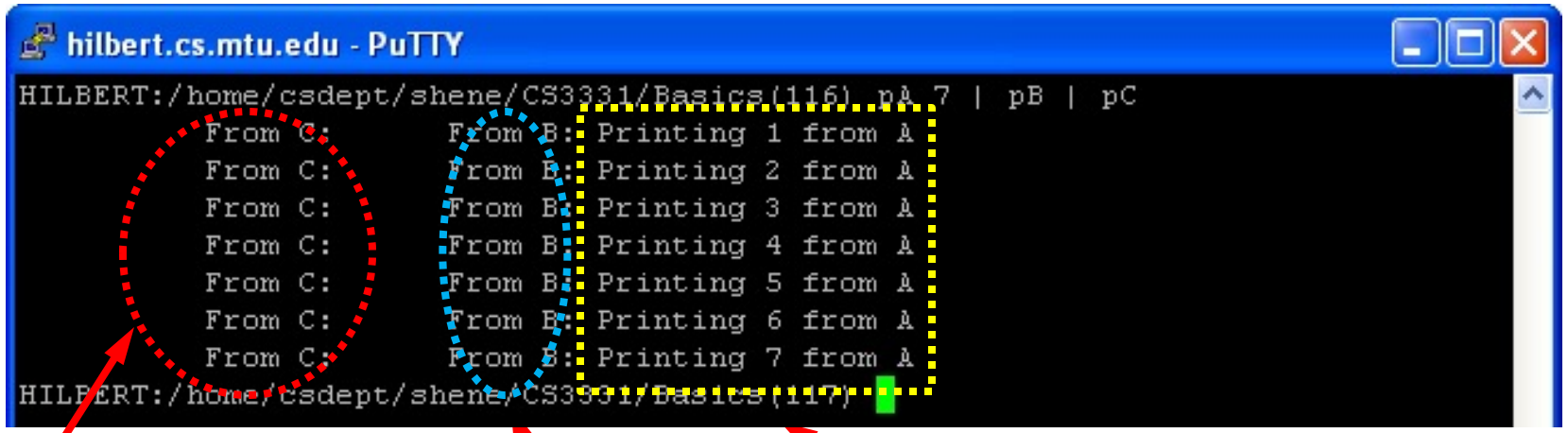
fgets()比gets()安全

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



pC 加上這部分

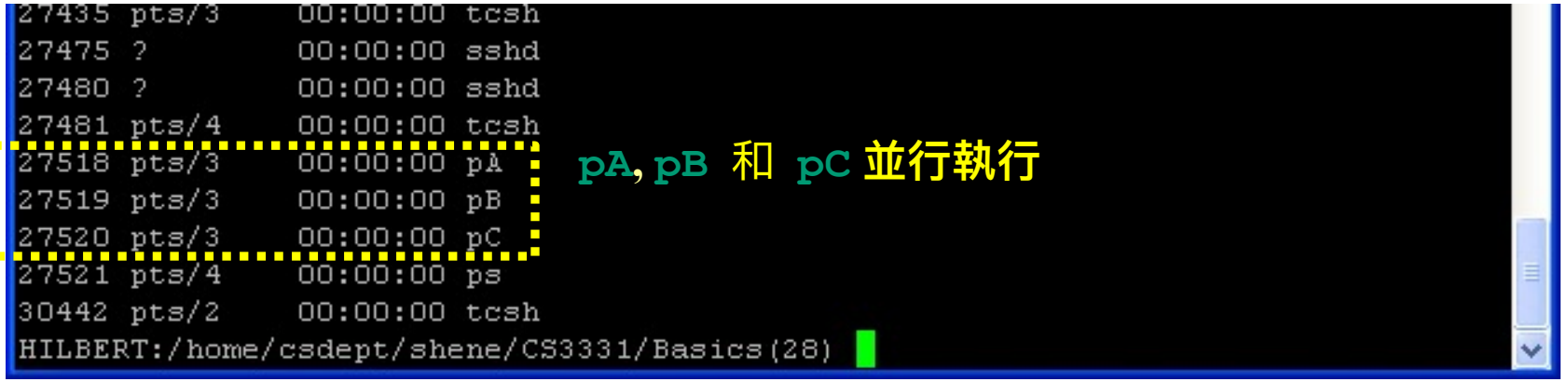
pB 加上這部分

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）



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
```

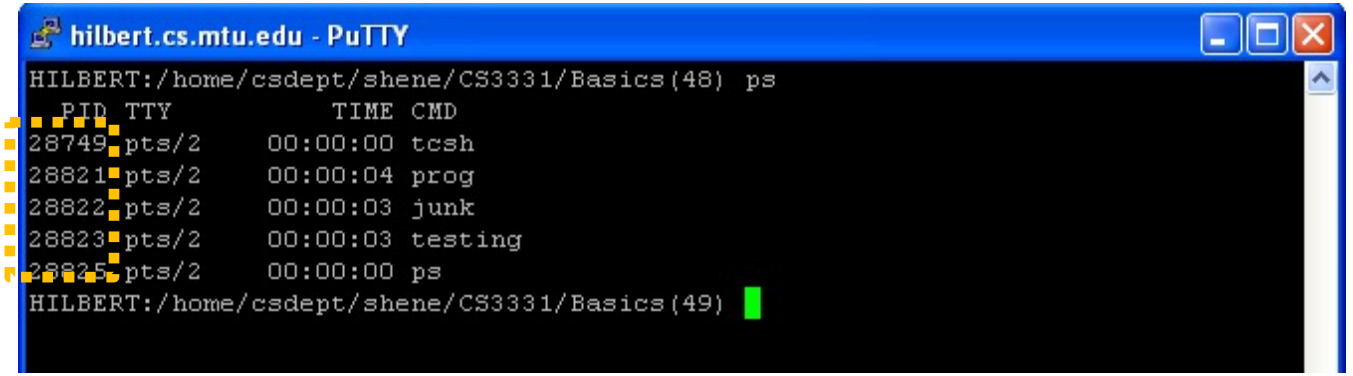
- `fg`: 把最近送到後庭的程式送回前庭執行（它的`stdin`是鍵盤）

```
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



- 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。

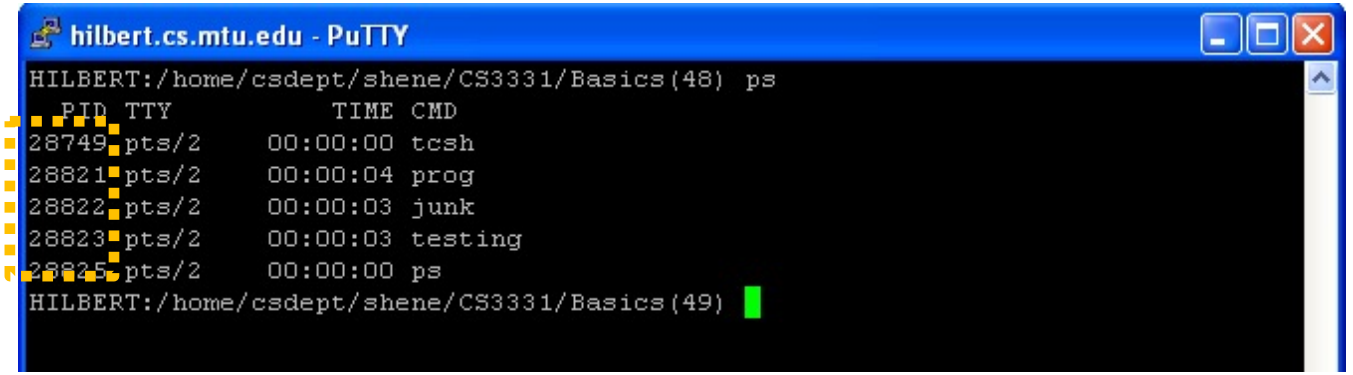
```
kill -KILL pid1 pid2 … pidi
```

把SIGKILL訊號送到 pid1 、 pid2、…、pid*i*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



- 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