並行計算

EP03: Process基本概念： 1/2

程式測試只能顯示程式有錯（bugs），
但卻不能證明程式中沒有錯。

Edsger W. Dijkstra

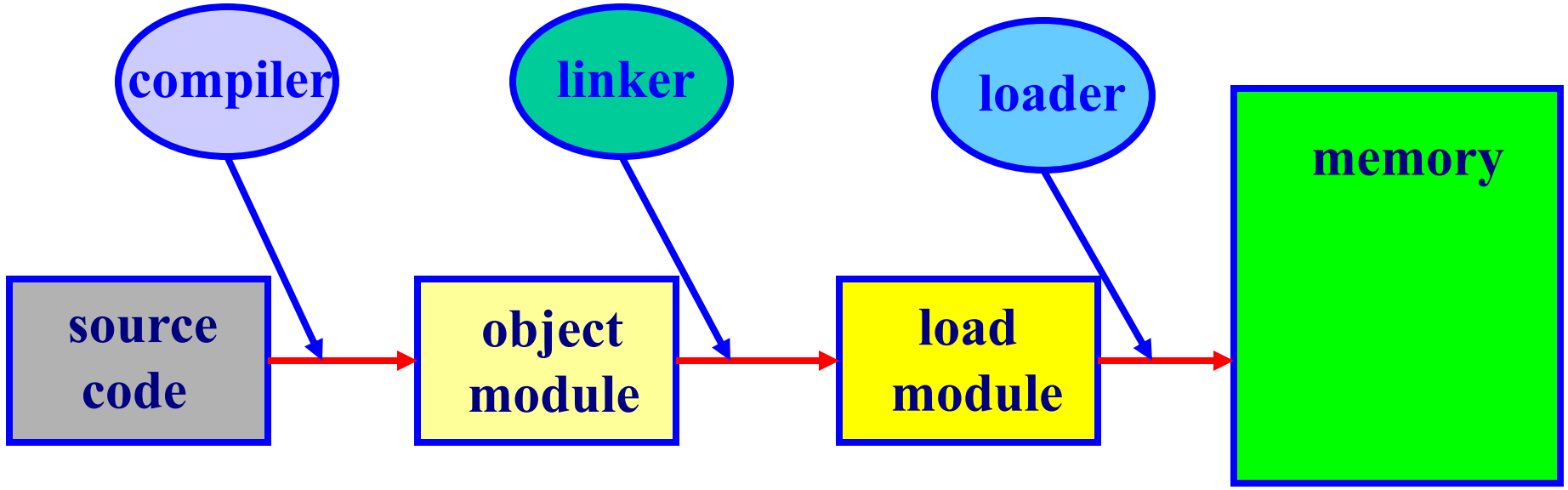
本集內容

- 何謂Process？
- Process的記憶體使用方式、 Process記憶體空間、Process狀態
- OS內表示Process的方式
- 調度Process、切換環境
- Process作業方式
- `fork()`
- 小心`printf()`
- `wait()`
- 幾個簡單問題
- 程式習題

請看影片的說明，那兒有取得投影片和程式的連結

從編譯到執行

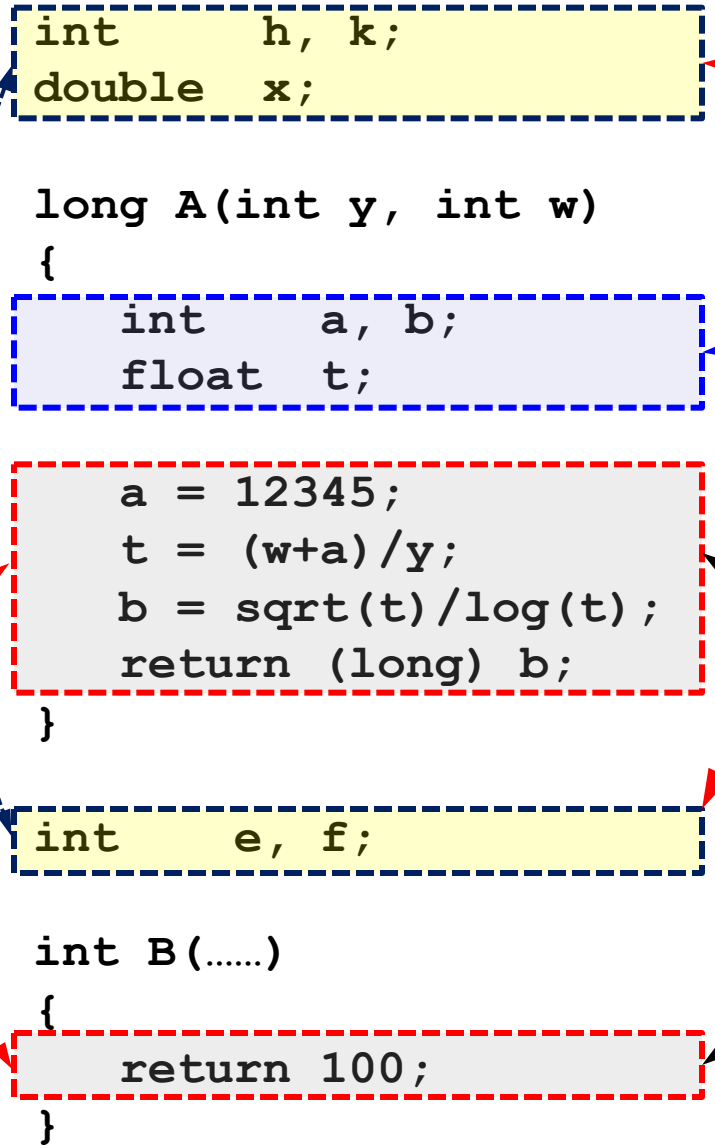
- 編譯程式（compiler）把原始程式編譯到 `.o` 檔案。
- 連結程式（linker）把 `.o` 檔和其它程式庫函數串成一個可執行檔（譬如 `a.out`）。
- 載入程式（loader）把可執行檔抄入記憶體準備執行。



什麼是一個Process?

- 當OS要執行一個程式（可執行檔）時，會把這個檔案抄入記憶體、再把控制權交給這個程式的第一道（機器）指令，於是程式開始執行。
- 一個process就是一個正在執行的程式。
- Process比程式多出許多東西，因為除了程式的可執行檔之外，一個process還包含了OS為了執行這個程式而產生的許多資訊。這包含了程式計數器（program counter）、堆疊（stack）、資料區域（data section）、指令區域（code section）、等等用來執行程式的東西。
- 還有，一個程式可以變成好幾個process執行；譬如，在不同視窗下執行同一個 `a.out`。

程式使用記憶體方式: 1/10



程式執行時在記憶體中位置固定

整體變數

它們是在任何函數之外宣告的變數

局部變數

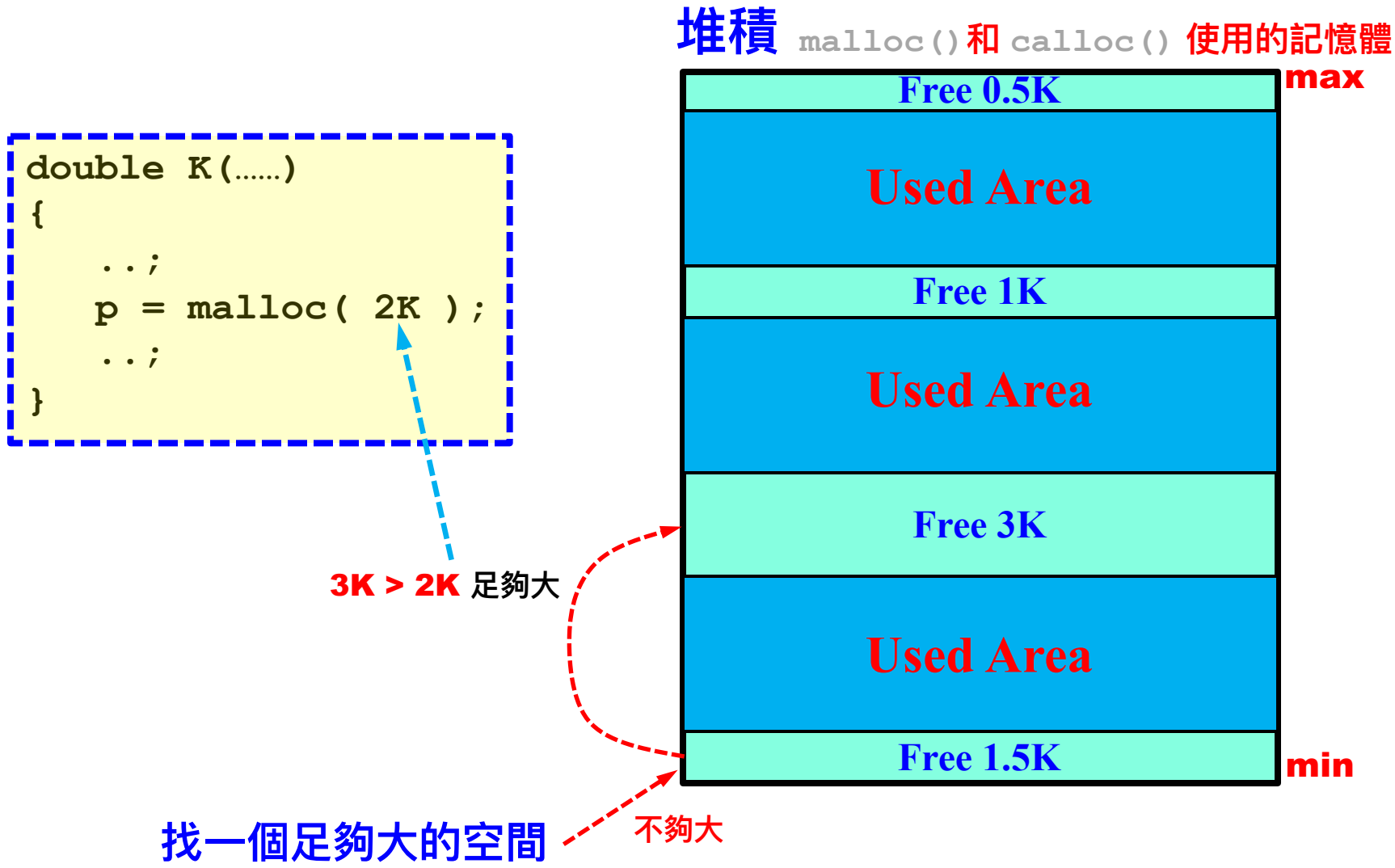
這些是在函數內部宣告的變數，
它們只能在該函數中使用。
在該函數之外是看不到這些變數的、
當然就無法使用。

指令

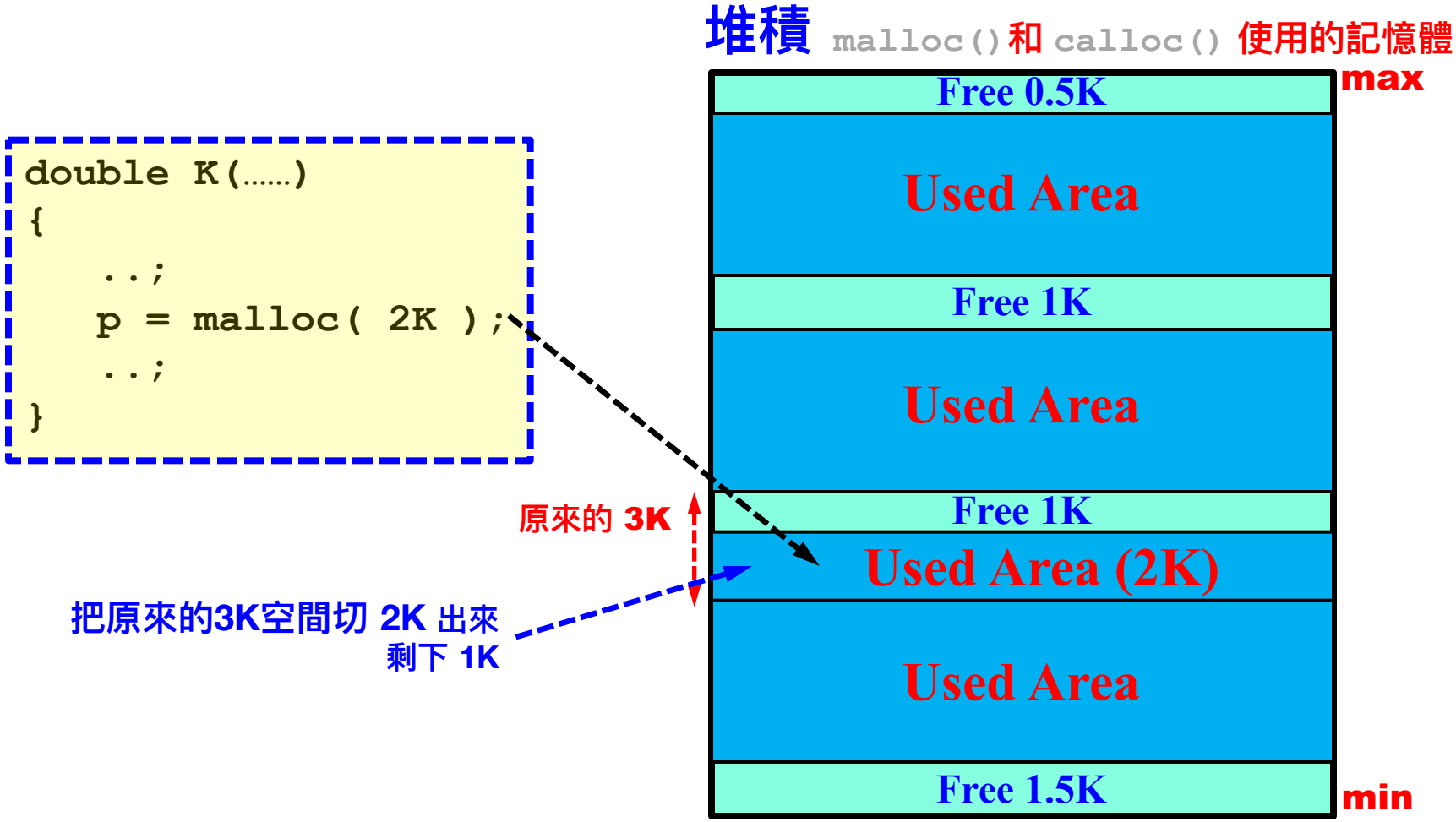
這些是可以執行的（機器）指令

編譯之後，編譯程式就知道整體變數的記憶體需求
連結之後，連結程式就知道程式指令的使用空間

程式使用記憶體方式: 2/10



程式使用記憶體方式: 3/10



給 malloc()和 calloc() 用的空間叫做堆積 **heap**

程式使用記憶體方式: 4/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```

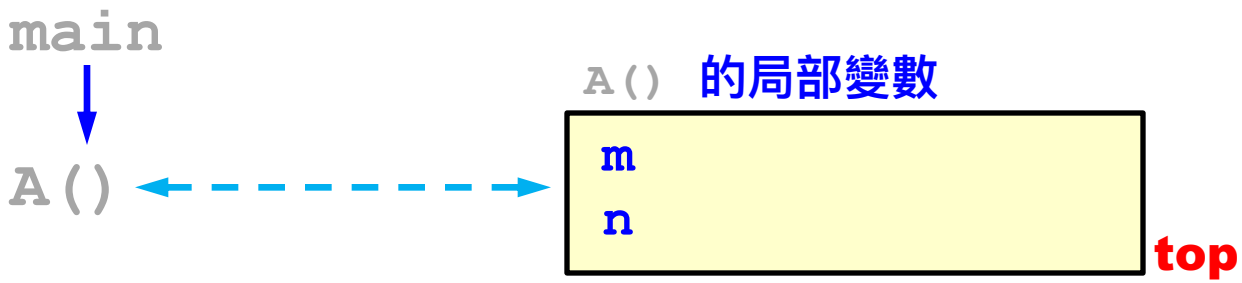
注意：叫用一個函數時，所有在該函數中宣告的變數必須要存在，才能使該函數正常執行。離開這個函數之後，在該函數內宣告的變數就不需要了。

- 當 main() 叫用 A() 時，m 和 n 必須存在可以使用
- 當 A() 叫用 B() 時，b[20] 必須存在可以使用
- 當 B() 叫用 C() 時，w 和 t 必須存在可以使用
- 當 C() 返回時，w 和 t 就不需要了
- 當 B() 返回時，b[20] 就不需要了
- 當 A() 返回時，m 和 n 就不需要了

叫用和返回（return）的順序是先入後出的（First-In-Last-Out）
於是為函數中的變數配置記憶體也是先入後出

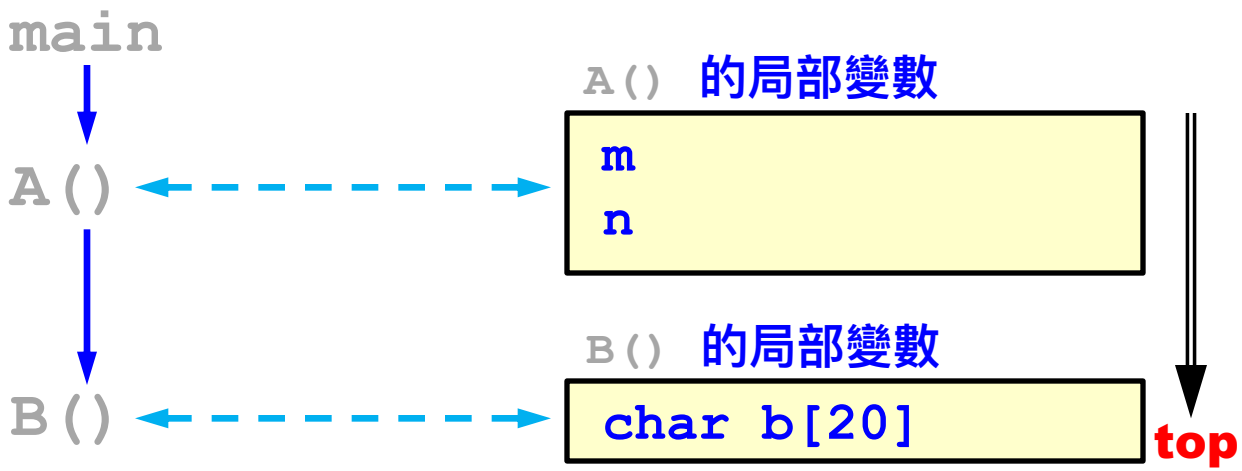
程式使用記憶體方式: 5/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```



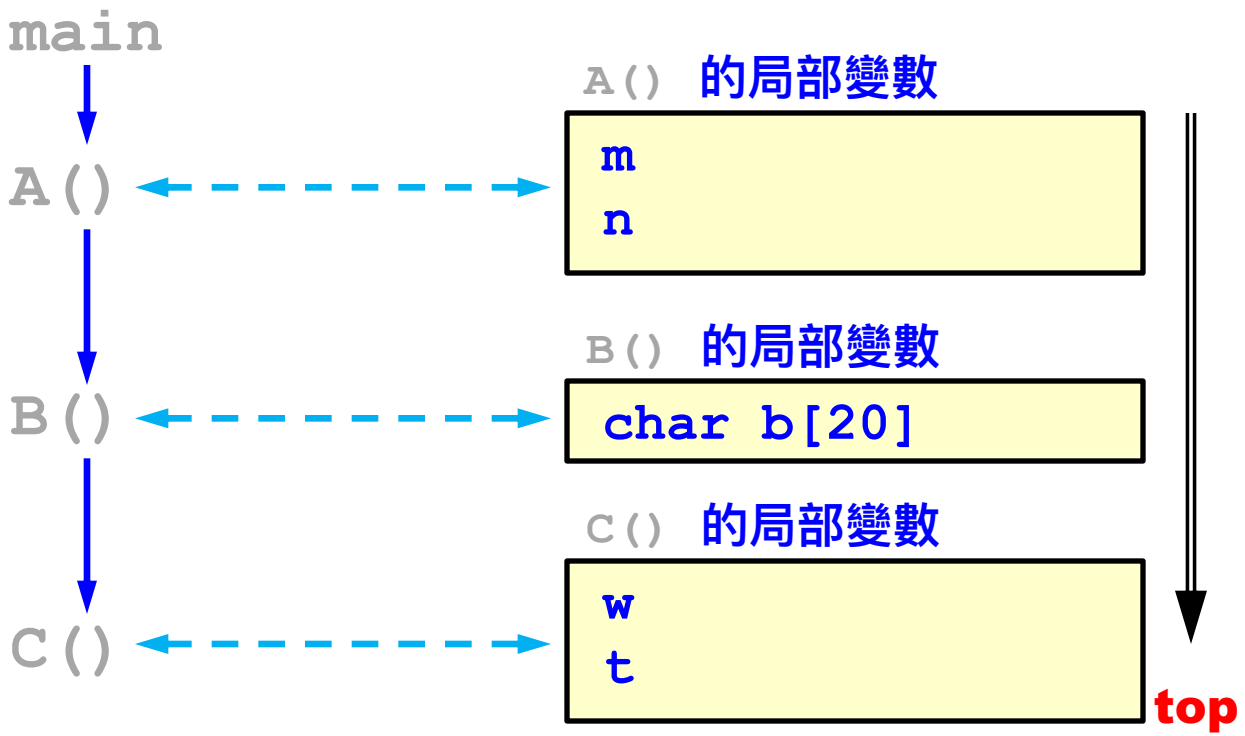
程式使用記憶體方式: 6/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```



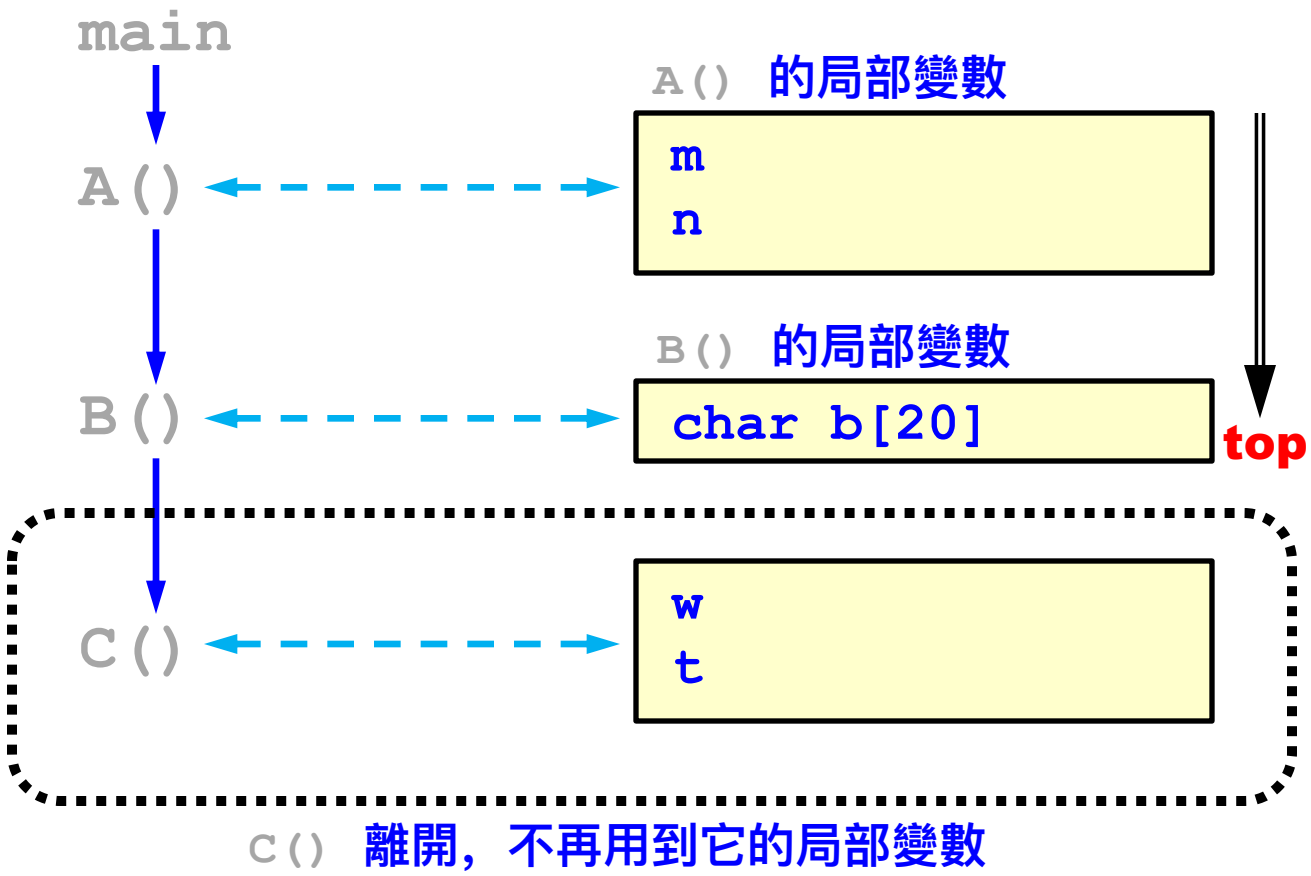
程式使用記憶體方式: 7/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```



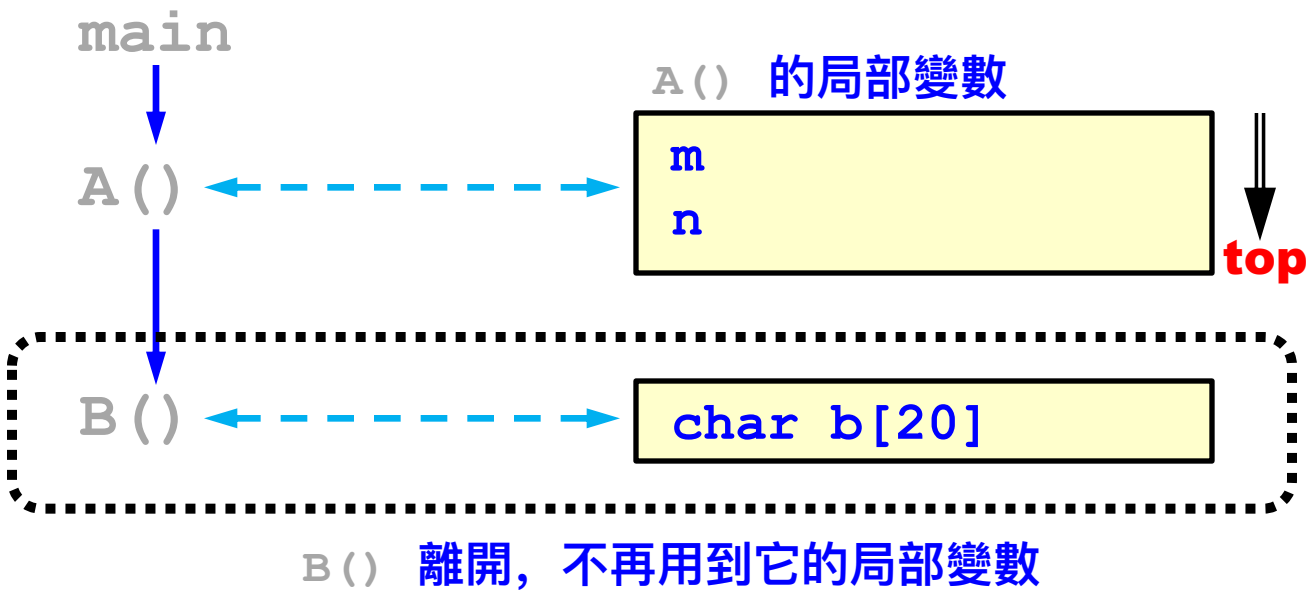
程式使用記憶體方式: 8/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```



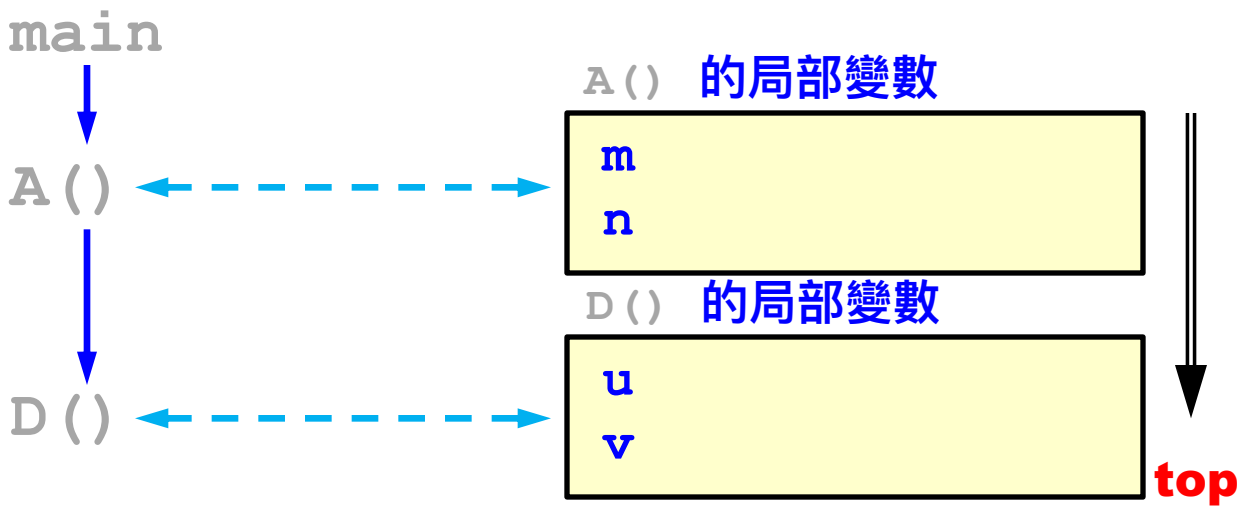
程式使用記憶體方式: 9/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```



程式使用記憶體方式: 10/10

```
int D(.. ..)
{
    int u,v;
    …
}

double C(.. ..)
{
    double w;
    long   t;
    …
}

long  B(.. ..)
{
    char b[20];
    .. ..
    C(.. ..);
}

void A(.. ..)
{
    short m, n;
    B(.. ..);
}
```

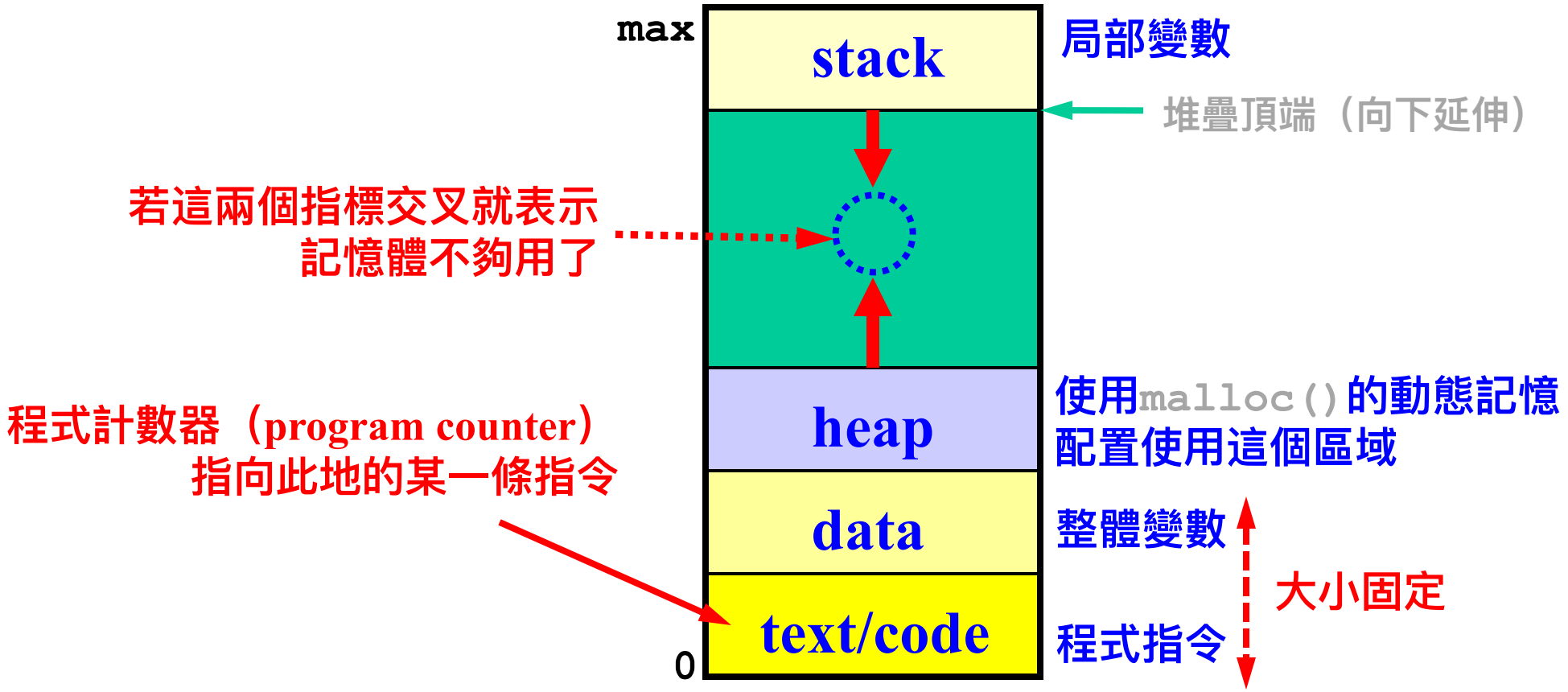


若 A() 叫用 D()，就得為 D() 配置局部變數，
於是很可能會蓋過原來 B() 用到的記憶體。

我們學到什麼?

- 叫用－返回的順序是先進後出或後進先出的。
- 這正好是個堆疊（stack）。
- 為各函數的局部變數配置記憶體得用堆疊（stack）。
- 綜合起來，為了`malloc()` 和 `calloc()` 配置記憶體，我們需要一個堆積（heap）；為了各函數的局部變數配置記憶體（先進後出），我們需要一個堆疊（stack）。

Process的記憶體空間

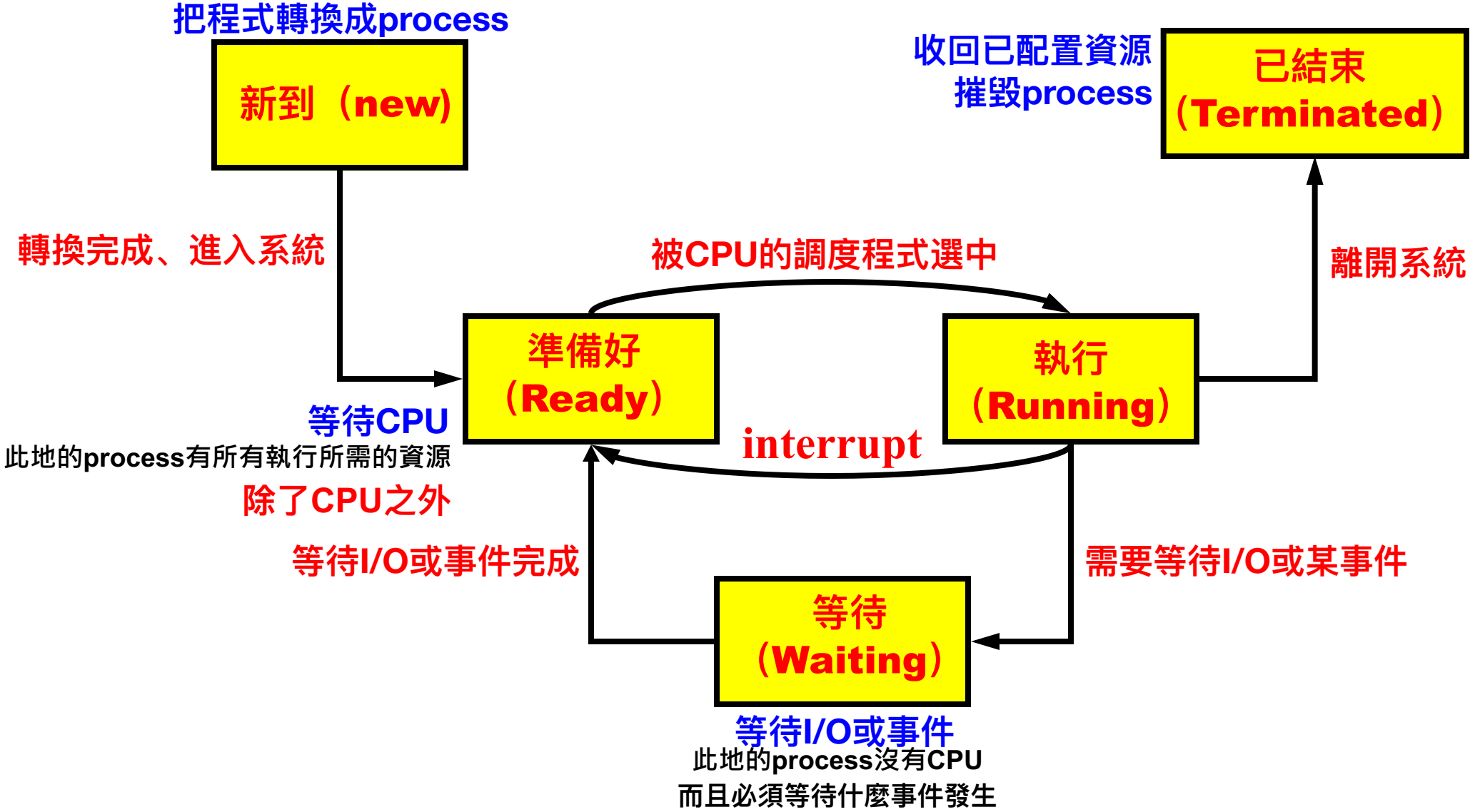


Process狀態

在任何時刻，一個process必定在下列五種狀態之一： 新到（new）、執行（running）、等待（waiting）、準備好（ready）、和已結束（terminated）：

- 新到（New）：OS正在把程式轉換成process
- 執行（Running）：該process正在使用CPU執行指令
- 等待（Waiting）：該process正在等待它期望的事件發生（譬如它起動的輸入/輸出完成）
- 準備好（Ready）：該process擁有所有除了CPU之外的資源；也就是說，只要OS給它CPU，它就可以執行
- 已結束（Terminated）： 該process已經結束執行（不論是正常或不常），正在等OS做一些收尾的工作

Process狀態圖


把程式轉換成process
新到（new)
收回已配置資源
摧毀process
已結束
(Terminated)
轉換完成、進入系統
被CPU的調度程式選中
離開系統
準備好
（Ready）
執行
（Running）
等待CPU
此地的process有所有執行所需的資源
除了CPU之外
interrupt
等待I/O或事件完成
需要等待I/O或某事件
等待
（Waiting）
等待I/O或事件
此地的process沒有CPU
而且必須等待什麼事件發生

Process狀態圖

Lubomir Bic and Alan C. Shaw,
***The Logical Design of Operating Systems*,**
Second Edition,
Prentice Hall, 1988

OS內如何表示一個Process

- 每一個process在產生時都會有一個process ID（就是一個唯一的正整數）。
- 有關該process的資訊會儲存在一個叫做Process Control Block（PCB，process控制表）的表格內；依系統不同，這未必是一個簡單的表、而可能是一個頗複雜的資料結構。
- 這些PCB會用表中的指標欄串到不同所在；譬如說，在「準備好」狀態中的process就會串到一個或多個「準備好調度/等待線（ready queue）」中。

調度Process: 1/2

- 因為系統中process的數目通常遠大於CPU或核心的數目，OS必須要讓CPU有最大的使用率（也就是閒置時間最低）、而且也不能讓準備好的process等太久。
- 為了決定哪個process可以做什麼，各個process會被串到若干調度線（scheduling queue）中，也有人稱為排班、排程線。
- 譬如，除了「準備好」的調度/等待線之外，每一個事件（等待某磁碟作業完成）都會有該事件的調度線，有時同一事件的調度線可能很複雜、會被分成好幾層或好幾條。

調度Process: 2/2

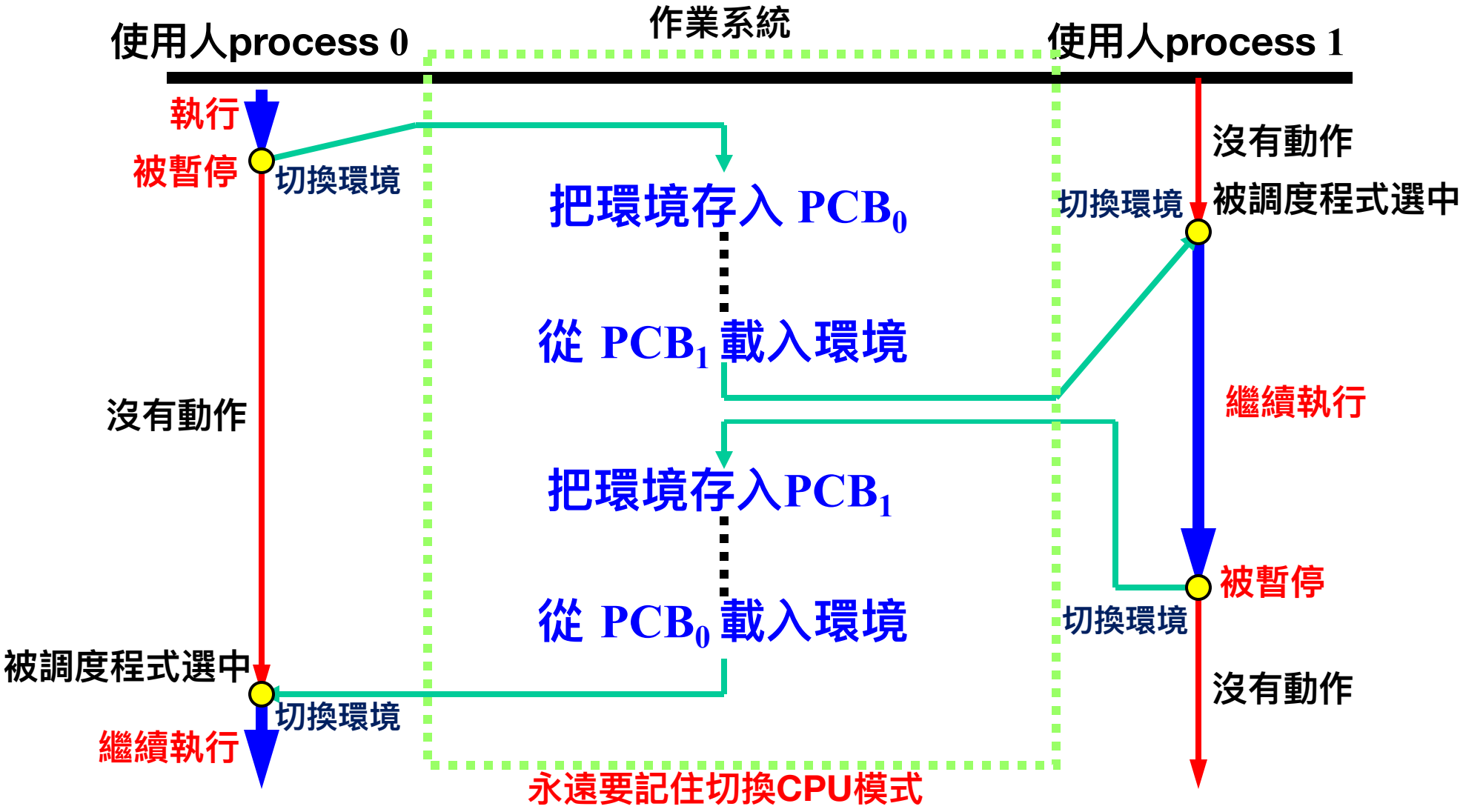
- 「準備好」調度線（或等待線）可能有若干條或若干層子線，這些是已經取得供它執行的資源、但獨缺CPU的process；這表示一旦把CPU給了該process，它立即可以執行。
- OS中有一個CPU調度程式（CPU Scheduler）。
- 當有一個CPU或核心空了下來，CPU調度程式會查「準備好」調度線，找一個合用的process，把CPU交給它、讓它繼續執行。
- 決定挑哪個process繼續執行的策略就叫做調度策略（Scheduling Policy）。
- 調度策略對本課程並不重要，因為一個並行程式是否能正常而且正確地工作不能依賴系統的調度策略。
- 您會在作業系統課中學到調度策略。

切換環境: 1/2

- 什麼是process的執行環境（簡稱環境，context）？這包含了執行一個process所需的所有資訊；譬如：process ID、process狀態、CPU暫存器的值、程式計數器（program counter）、以及其它記憶體（堆疊、堆積所在）和檔案管理等等的資訊。
- 什麼是環境切換（context switch）？當CPU調度程式挑了一個process、把CPU給它使用之前，調度程式得做以下的事：
  - 把目前正在執行但被暫停的process的環境保存起來（通常在PCB）
  - 把這個被暫停的process放到「準備好」或對應的「等待」調度線中
  - 把被挑選要執行的process的環境抄到各對應的地方（譬如把暫存器的值抄回暫存器）
  - 執行該process 被保存起來的程式計數器指向的指令

這兩個地方可能需要切換CPU的執行模式

切換環境: 2/2


使用人process 0
作業系統
使用人process 1
執行
被暫停
切換環境
把環境存入 PCB0
從 PCB1 載入環境
沒有動作
把環境存入PCB1
從 PCB0 載入環境
被調度程式選中
切換環境
繼續執行
永遠要記住切換CPU模式
沒有動作
切換環境
被調度程式選中
繼續執行
被暫停
切換環境
沒有動作

程式寫作

我們使用類似Unix的系統（Unix、Linux、macOS 等）
若您只用Windows，您得裝一個VM（Virtual Machine）

Process的作業方式

- 有三個常見的作業方式：
  - ❖產生Process：建立一個process以及它的執行環境。這個被產生的新process是產生它的process的子process（child process），而產生的process叫做母process（parent process）；這個母-子關係是個樹狀結構。在Unix類似系統中使用 `fork()` 函數產生子process。
  - ❖終止Process：終止一個process執行。在Unix類似系統中使用 `exit()` 函數。
  - ❖和子process匯合（Join）：等待一個子process執行完成。在Unix類似系統中使用 `wait()` 函數。
- `fork()`、`exit()`和 `wait()`都是系統叫用函數、都會產生trap。

一些必需的引入檔

- 程式中在使用process之前，必須引入 `sys/types.h` 和 `unistd.h` 。
- `sys/types.h` 中定義所有系統中的資料型別，而 `unistd.h` 宣告了標準的符號常數（也就是給常數一個符號名稱）和型別。

```
#include   <sys/types.h>
#include   <unistd.h>
```

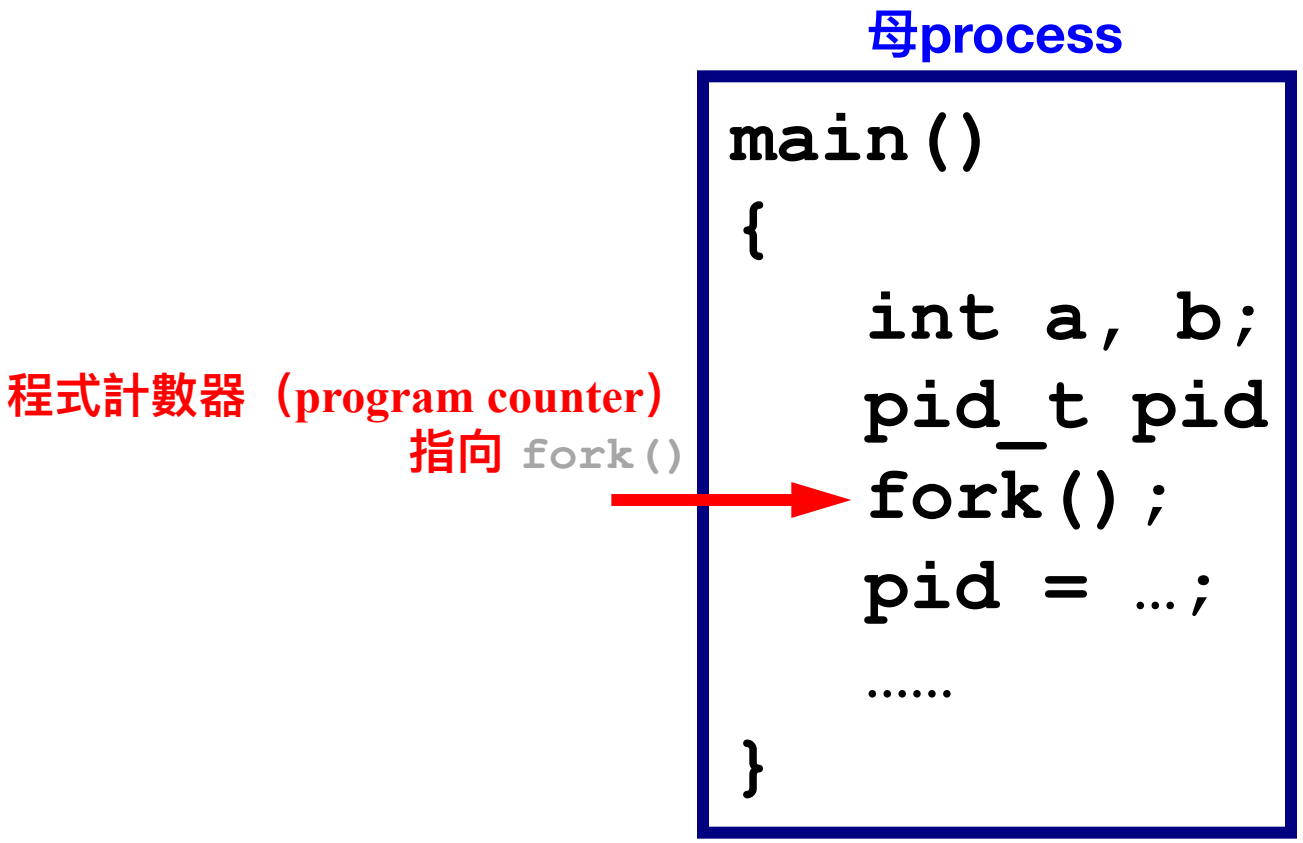
fork( )系統叫用函數

- 函數 `fork()` 的用途是為叫用它的process產生一個子process。產生和被產生的process分別叫做母和子的關係（母process和子process），這個母子關係形成一個樹狀結構。
- 函數 `fork()` 並不需要任何參數！
- 如果叫用 `fork()` 是成功的，Unix產生一個和叫用的process完全一樣、但卻是分離的記憶體空間，然後讓被產生的子process執行。細節容後再詳細解釋。
- 母process和子process都從 `fork()` 函數的下一條指令開始執行。

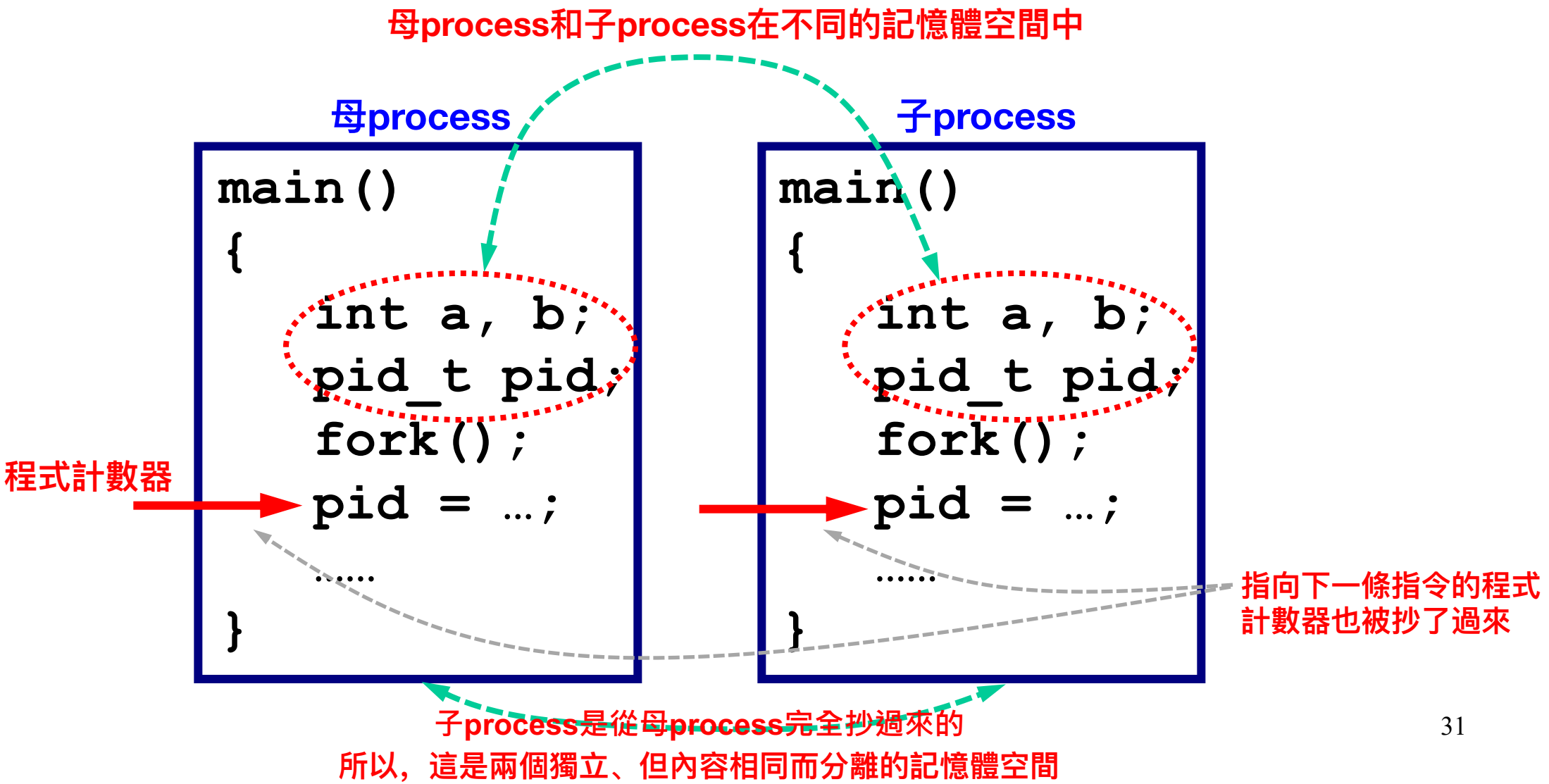
fork( ) 的傳回值

- 若傳回一個負值，這表示叫用 `fork()` 失敗、並沒有產生子process。
- 若傳回的值為0，表示該process是子process。
- 若傳回的值大於0，這個值就是被產生的子process的process ID。 這個ID的型別是`pid_t`，它是在前述的引入檔中宣告的。
- 函數 `getpid()` 傳回叫用它的process的ID。
- 函數 `getppid()` 傳回叫用它的process的母process的ID。如果該process沒有母process，就傳回 `1`；稍後我們會看到此地 `1`的代表意義。

執行 fork( ) 之前


母process
main()
{
int a, b;
pid_t pid
fork();
pid = …;
……
}
程式計數器（program counter）
指向 fork()

執行 fork( ) 之後


母process和子process在不同的記憶體空間中
母process
子process
main()
{
int a, b;
pid_t pid;
fork();
pid = …;
.....
}
main()
{
int a, b;
pid_t pid;
fork();
pid = …;
.....
}
程式計數器
指向下一條指令的程式
計數器也被抄了過來
子process是從母process完全抄過來的
所以，這是兩個獨立、但內容相同而分離的記憶體空間

小心 printf()：1/3

- 在同一個終端機視窗執行的若干process共用一個`stdout`，也就是該視窗做輸出。
- 由於交錯執行，這個共用 `stdout` 會使`printf()`發生資源共享的問題。
- 如果不同process同時用 `printf()` 做輸出，於是交錯執行很可能會把一個未完成的`printf()`暫停、而讓另一個 `printf()`執行，於是視窗中同一列上可能會夾雜著來自不同的process的輸出。

小心 printf()：2/3

- 下面是兩個process的一部分。
- Process 1 印出A之後被暫停，而由process 2 繼續（反過來也如此）。
- 於是兩個process就都往同一個stdout交錯地輸出！
- 視窗中同一列就可能出現來自process 1和process 2的輸出內容。

Process 1	Process 2
`A = 10; B = 20;`	`X = 100; Y = 200;`
`printf("A=%d B=%d\n",A,B);`	`printf("X=%d Y=%d\n",X,Y);`

小心 printf()：3/3

- 下面是上一頁的程式
- 請記住**交錯執行**這個概念

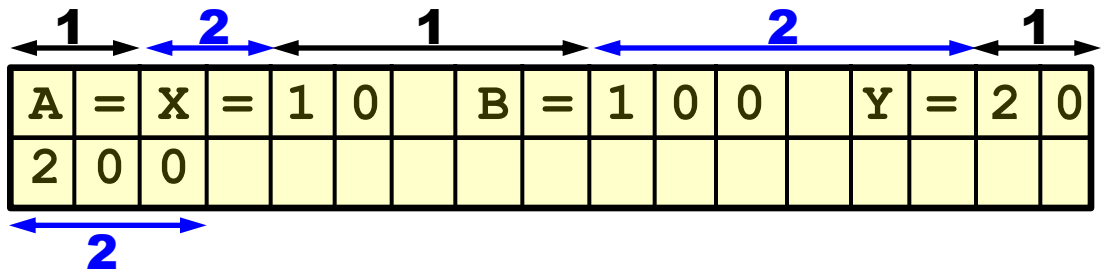
Process 1

```
A = 10; B = 20;
printf("A=%d B=%d\n",A,B);
```

Process 2

```
X = 100; Y = 200;
printf("X=%d Y=%d\n",X,Y);
```

Process 1	Process 2
print A	
	print X
print B	
	print Y



我們假設一個process印完文字、而必須把 int 數值轉換成 ASCII文字（%d）時會被暫停、切換給另一個process。

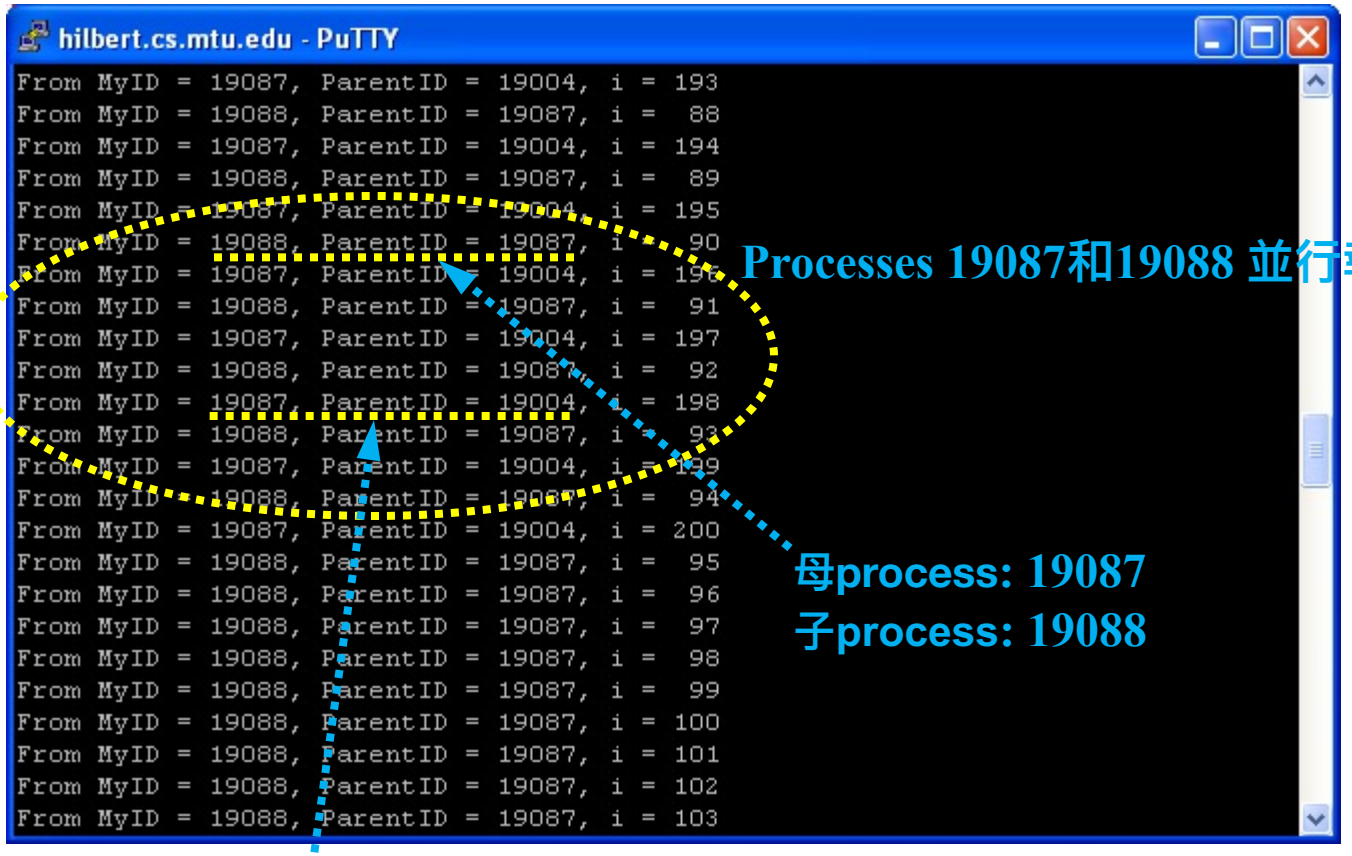
範例 1: 1/2

fork-1.c

```
#include   <stdio.h>
#include   <string.h>
#include   <sys/types.h>
#include   <unistd.h>

void main(void)
{
   pid_t  MyID, ParentID;
   int    i;
   char   buf[100];

   fork();                    // create a child process
   MyID     = getpid();       // get my process ID
   ParentID = getppid();      // get my parent's process ID
   for (i = 1; i <= 200; i++) {
      sprintf(buf, "From MyID=%ld, ParentID=%ld, i=%3d\n",
                   MyID, ParentID, i);
      write(1, buf, strlen(buf)); // 請回想 printf() 的問題
   }                              // 所以我們此地不用printf()
}
```

write()中的 1 就是stdout
把buf處長度為strlen(buf)的字串
輸出到stdout

範例 1: 2/2



Processes 19087和19088 並行執行

母process: 19087
子process: 19088

19004
↓
19087
↓
19088

process 19087的母process為19004
這個終端機視窗是由它的shell程式管理
這個shell 執行`fork-1`

一個 fork( )的典型使用方式

```
main(void)
{
  pid_t pid;

  pid = fork();
  if (pid < 0)
    printf("Oops!");
  else if (pid == 0)
    child();
  else // pid > 0
    parent();
}
```

```
void child(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf(" Child:%d\n", i);
  printf("Child done\n");
}

void parent(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf("Parent:%d\n", i);
  printf("Parent done\n");
}
```

往後為了省投影片空間可能會用printf()
但請記住用 printf() 的問題

執行 fork( ) 之前

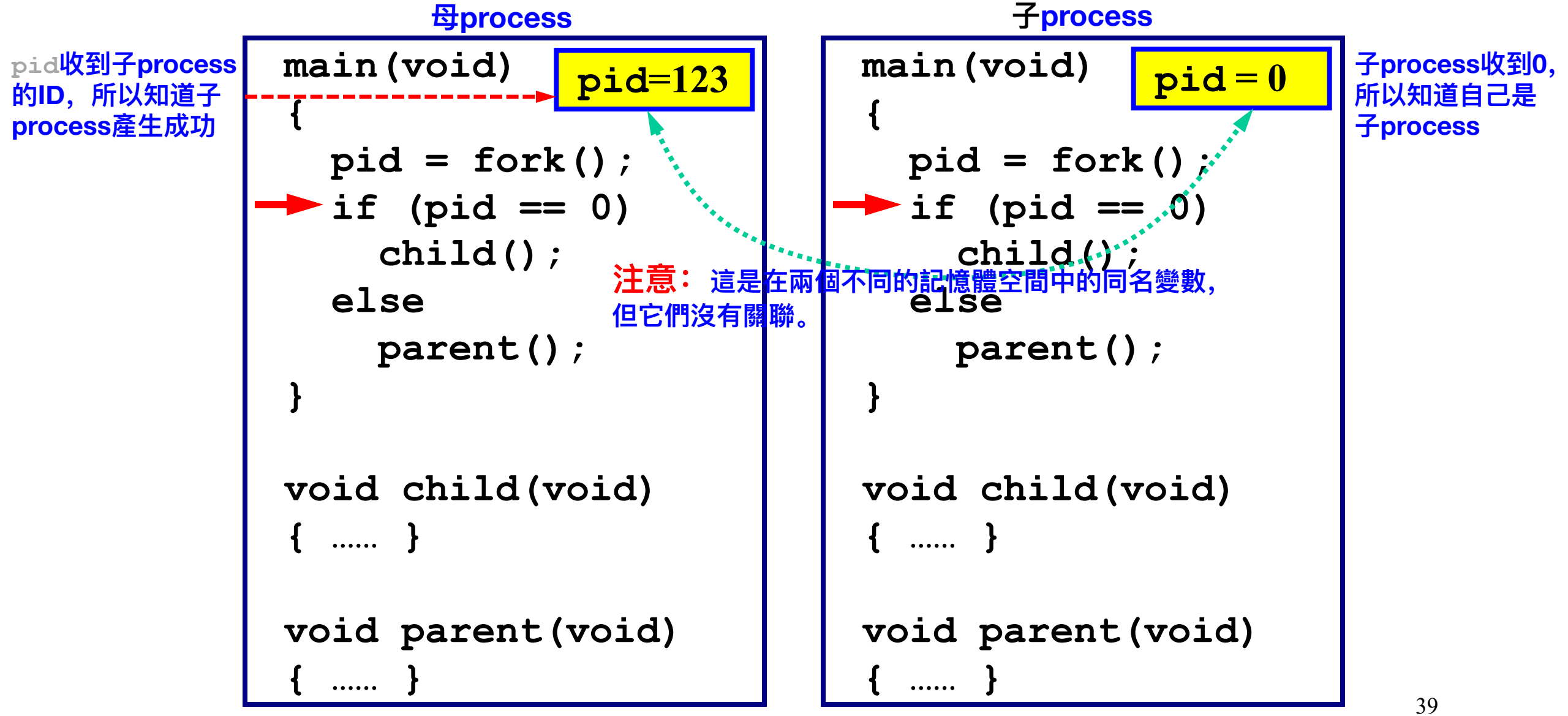
母process

```
main(void)
{
  pid = fork();
    if (pid == 0)
      child();
    else
      parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

pid = ?

目前pid的值未知

執行 fork( ) 之後: 1/2


母process
子process
pid收到子process的ID，所以知道子process產生成功
子process收到0，所以知道自己是子process
pid=123
pid = 0
main(void)
{
pid = fork();
if (pid == 0)
child();
else
parent();
}
void child(void)
{ …… }
void parent(void)
{ …… }
注意：這是在兩個不同的記憶體空間中的同名變數，但它們沒有關聯。

執行 fork( ) 之後: 2/2

parent

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

pid=123

child

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

pid = 0

範例 2: 1/2

fork-2.c

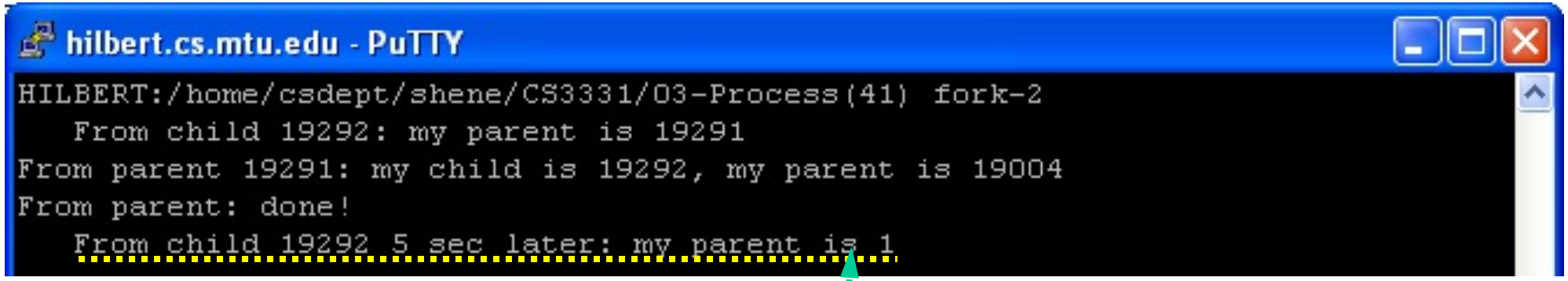
```
#include .....
void main(void)
{
   pid_t  pid;

   pid = fork();
   if (pid == 0) {                  // child here
      printf("   From child %ld: my parent is %ld\n",
                                    getpid(), getppid());
      sleep(5);
      printf("   From child %ld 5 sec later: parent %ld\n",
                                    getpid(), getppid());
   }
   else {                           // parent here
      sleep(2);
      printf("From parent %ld: child %ld, parent %ld\n",
                  getpid(), pid, getppid());
      printf("From parent: done!\n");
   }
}
```

子process停止5秒，但母process只停2秒
母process比子process先結束

所以，子process在母process終結後才用getppid()印出母process的ID

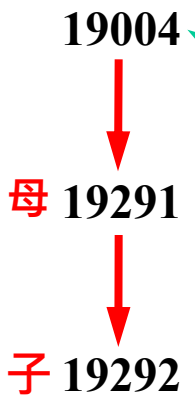
sleep()是一個叫用系統的服務它強迫叫用它的process停止執行若干秒

範例 2: 2/2



母process已經終結的process叫做孤兒（orphan）process
它的母process就被重新定成init，這是Unix中負責產生所有process的process

當母process正在執行時



母process終結

1是一個叫做init的process

19004是產生19291的shell process

範例 3: 1/2

fork-3.c

```
#include   ……     // 不同而且分離的記憶體空間

void  main(void)
{
   pid_t  pid;
   char   out[100];
   int    i = 10, j = 20;

   if ((pid = fork()) == 0) {    // child here
       i = 1000; j = 2000;       // child changes values
       sprintf(out, "   From child: i=%d, j=%d\n", i, j);
       write(1, out, strlen(out));
   }
   else {                        // parent here
      sleep(3);
       sprintf(out, "From parent: i=%d, j=%d\n", i, j);
       write(1, out, strlen(out));
   }
}
```

這是和下列相同的

```
pid = fork();
if (pid == 0)
…
```

強迫母process在子process結束後才印出結果

範例 3: 2/2

```
HILBERT:/home/csdept/shene/CS3331/03-Process(46) fork-3
   From child: i = 1000, j = 2000
From parent: i = 10, j = 20
HILBERT:/home/csdept/shene/CS3331/03-Process(47)
```

子process把 `i` 和 `j` 分別從 10 和 20 改成 1000 和 2000

母process的 `i` 和 `j` 不受子process影響，因此母process和子process在獨立而且分離的記憶體空間中執行

wait() 叫用系統函數

- `wait()` 會暫停叫用它的process的執行，一直到它的某個子process終止或是收到一個Unix的訊號（signal）為止。
- `wait()` 需要一個指向整數變數的指標做參數，並且傳回那個終止執行的子process的ID。如果叫用`wait()`時並沒有子process在執行，就傳回`-1`。
- 參數的整數指標並不是當成一個整數使用，而是把這個位置中的各個bit賦於某種意義。系統會使用這些bit傳回子process終止執行時各種狀態。

如何使用 wait()

- 等待一個不指明的子process完成

```
wait(&status);
```

- 等待若干個（譬如 n 個）不指明的子process完成

```
for (i = 0; i < n; i++)
    wait(&status);
```

- 等待已經知道ID的那個子process完成

```
while (pid != wait(&status))
    ;
```

若執行while時該子process已經結束，這可能會是個無窮迴圈！

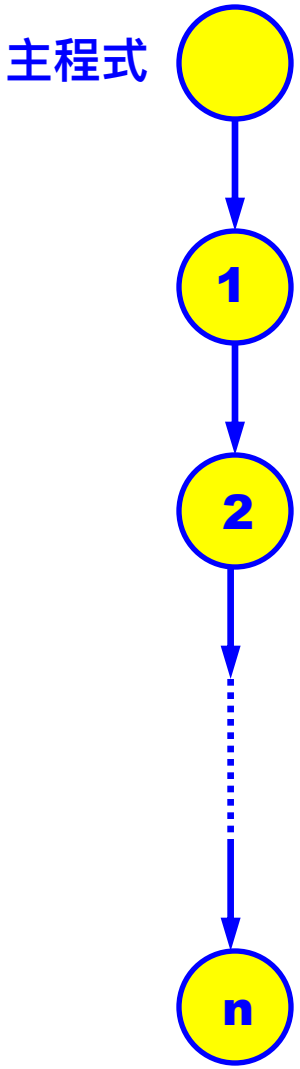
wait() 叫用系統函數例子

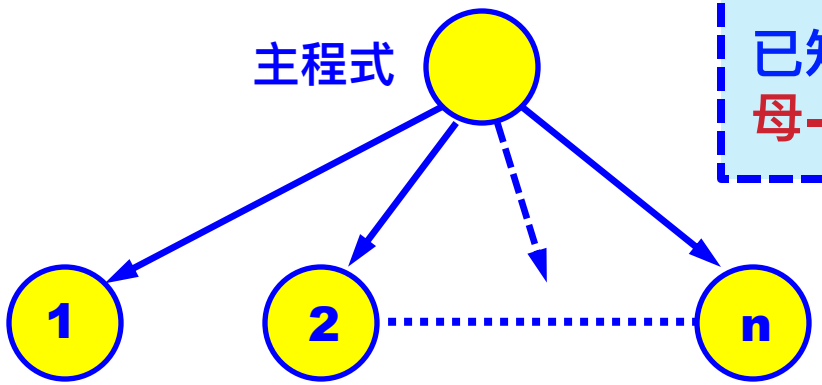
```
void main(void)
{
   pid_t pid, pid_child;
   int   status;

   if ((pid = fork()) == 0)   // child here
      child();
   else {                     // parent here
      parent();
      pid_child = wait(&status);
   }
}
```

想一想

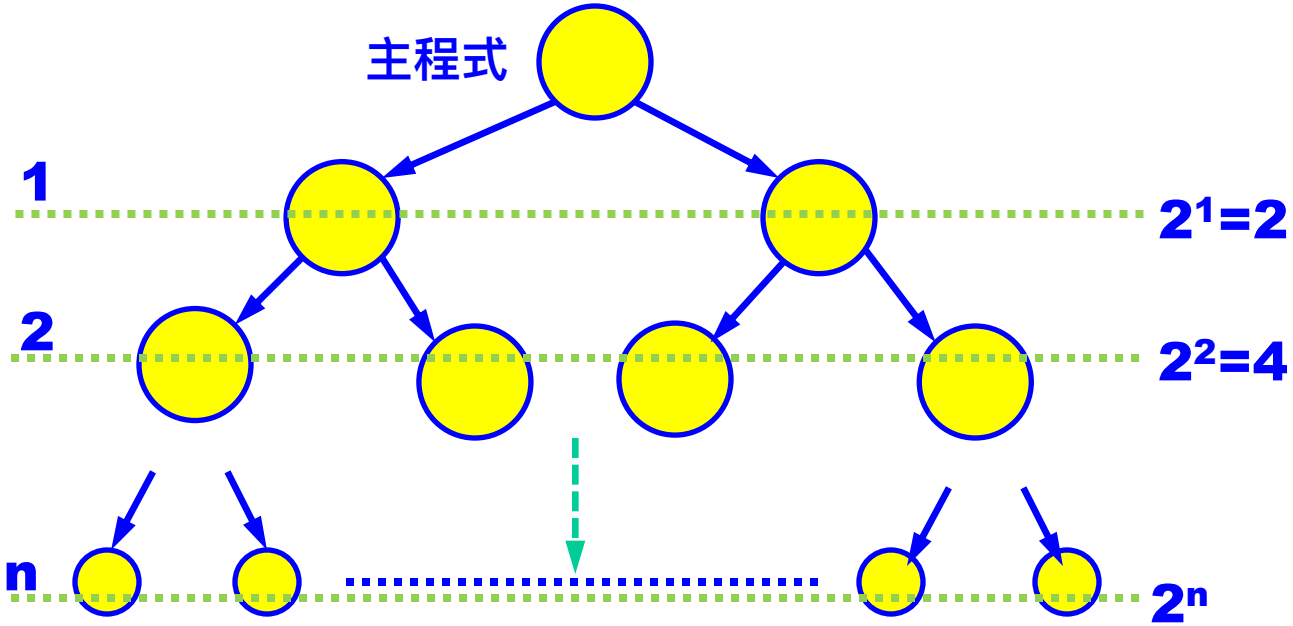
幾個簡單的問題

幾個簡單的問題: 1/2





已知一個正整數 n，寫出可以產生如下母-子關係的process；箭號表示母子關係。



幾個簡單的問題: 2/2

請畫出這幾個程式所產生的process的母-子關係；假設每一個 `fork()` 都成功。

```
void main(int argc, char **argv)
{
   int i, n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (fork())
         break;
   printf("Process %ld with parent %ld\n",
           getpid(), getppid());
   sleep(1);
}
```

```
void main(int argc, char **argv)
{
   int i, n = atoi(argv[1]);
   for (i = 0; i < n; i++)
      if (fork() == -1)
         break;
   printf("Process %ld with parent %ld\n",
           getpid(), getppid());
   sleep(1);
}
```

```
void main(int argc, char **argv)
{
   int i, n = atoi(argv[1]);
   for (i = 0; i < n; i++)
      if (fork() <= 0)
         break;
   printf("Process %ld with parent %ld\n",
           getpid(), getppid());
   sleep(1);
}
```

程式習題

題目

- 請寫一個C的（主）程式，它產生四個子process，P_1、P_2、P_3和P_4。
- 把這個程式叫做 `prog1.c`，編譯連結後得到可執行檔 `prog1`。
- 在執行時使用：`./prog1 m n r s` 此地`m`、`n`、`r`和`s`都是正整數
- 譬如，下面命令列中 `m` 是 `7300000000000000`、`n`為 `10`、`r`為 `100000` 而 `s`為 `200000`：

```
./prog1 7300000000000000 10 100000 200000
        m                n  r      s
```

- 後面再解釋這四個值的意義。

Process P_1 的工作（Fibonacci數）

- 用下面的（遞歸，recursive）式子計算第n個Fibonacci數。使用這個recursive的算法是要讓 P_1 用很長的CPU時間。（為什麼會久？請仔細想一想。）此地的 n 是執行行`prog1`時的第二個參數。

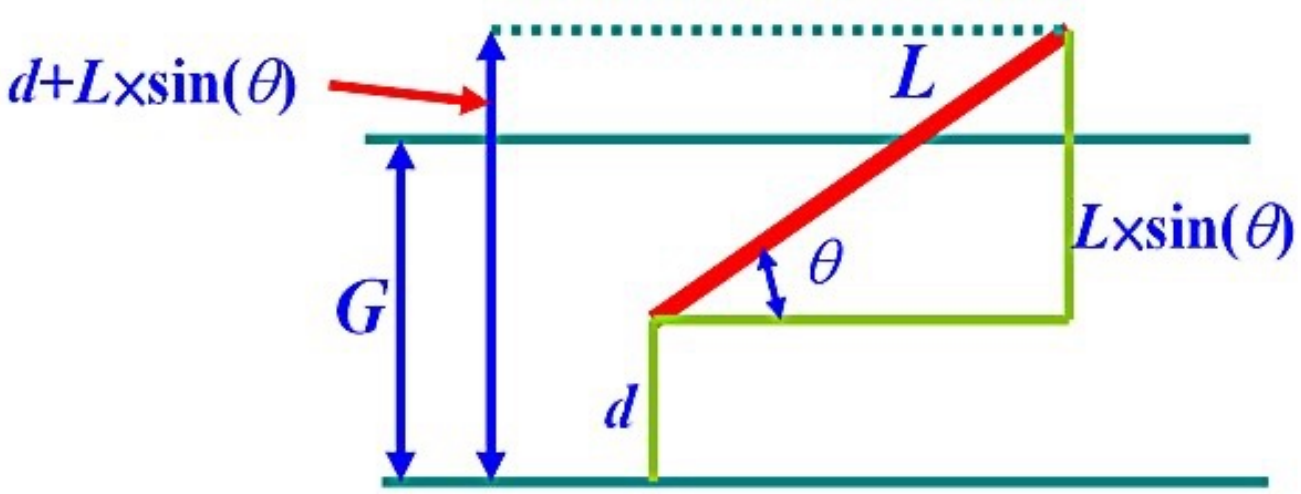
$$f_1 = f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2} \ (n > 2)$$

- 若n = 10， P_1 印出下面內容，每一列前方得有6個空格：

```
Fibonacci Process Started
Input Number 10
Fibonacci Number f(10) is 55
Fibonacci Process Exits
```

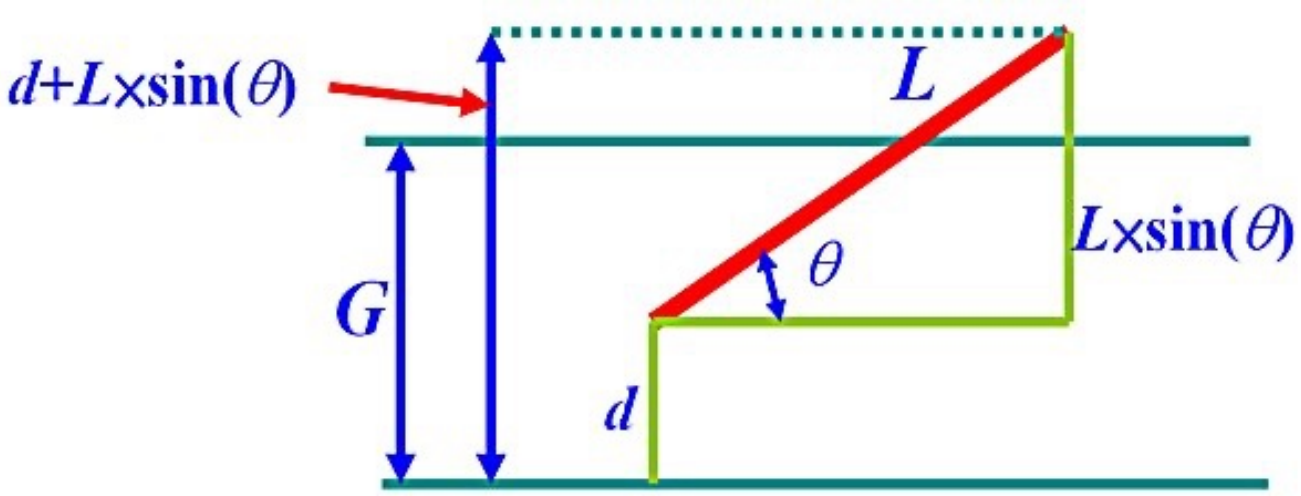
Process P_2 的工作（ Buffon丟針問題）：1/5

- 這個問題是1733年由法國自然學家和數學家George-Louis Leclerc, Comte de Buffon提出的，他自己在1777年解決了這個問題。
- 假設地面上畫了相距為 G 的平行線（見下圖），然後又有一根長度為 L 的針。若隨意把針拋出去，它壓線的機率是多少?



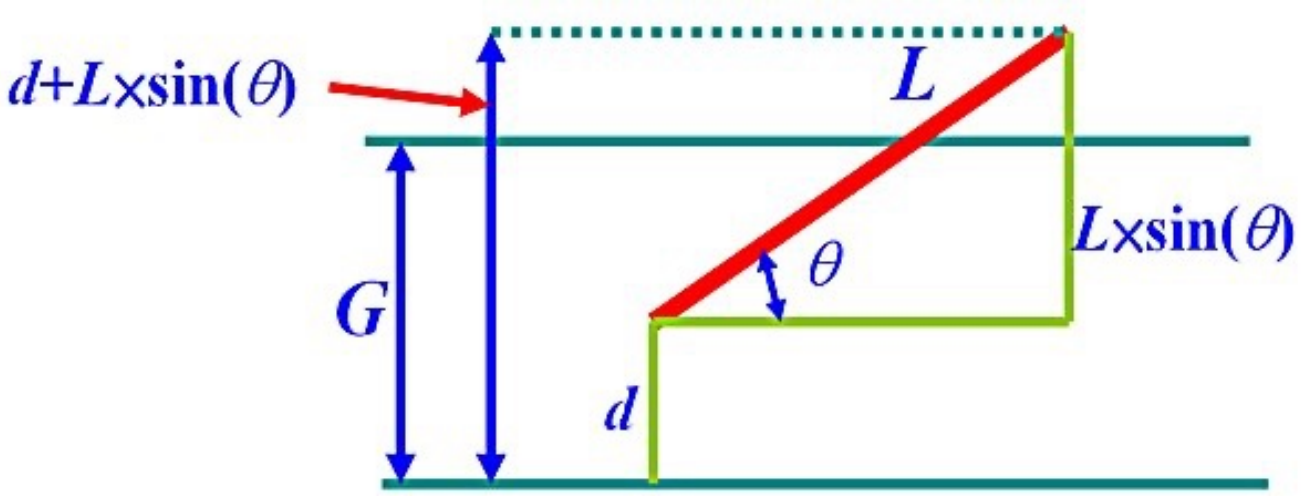
Process P_2 的工作（ Buffon丟針問題）：2/5

- 我們固定針的一頭，當落地後令這個針頭距下方的線的距離為 d，而針和下方線的夾角為θ。於是當$d+L\times\sin(\theta)$大於G，針會壓到上方的線；若小於0，針則會壓到下方的線。答案是$(2/\pi)\times(L/G)$ 。



Process P_2 的工作（ Buffon丟針問題）：3/5

- 為了方便起見，請用$L = G = 1$；於是機率為 $2/\pi = 0.6366$，因此有超過6成的機率針會壓線。
- 不過，我們用實驗計算這個結果。



Process P_2 的工作（ Buffon丟針問題）：4/5

- 我們得產生兩個亂數。
  - 第一個亂數在[0, 1)，它代表 d 的值。
  - 第二個亂數在[0, 2π)，它代表 θ 的值。用`acos(-1.0)`可以取得 π 的值。
- 然後計算 $d+L\times\sin(\theta)$ ，若它大於 1 或小於 0 表示壓線。
- 這道手續反覆 r 次（ r 從命令列參數取得），若有 k 次壓線，那麼當r 夠大時 k/r 大約就是答案。

Process P_2 的工作（ Buffon丟針問題）：5/5

- 我們得產生兩個亂數。別忘了，使用亂數之前程式得引入 `stdlib.h`；在使用`sin()` 和`acos()`時得引入`math.h`，而且在編譯時得用`-lm`明白指出這個程式要用到數學程式庫。
- 這個Buffon丟針問題的process輸出如下；不但如此，每一列輸出前方得有 9 個空格。
- 注意： 因為`rand()`產生的亂數範圍在`0`和`RAND_MAX` 之間，您得把產生的亂數用`(float rand())/RAND_MAX` 轉換到[0,1)；`RAND_MAX`這個亂數的最大值是在`limits.h`中定義，別忘了引入這個檔案。

```
Buffon's Needle Process Started
Input Number 100000
Estimated Probability is 0.63607
Buffon's Needle Process Exits
```

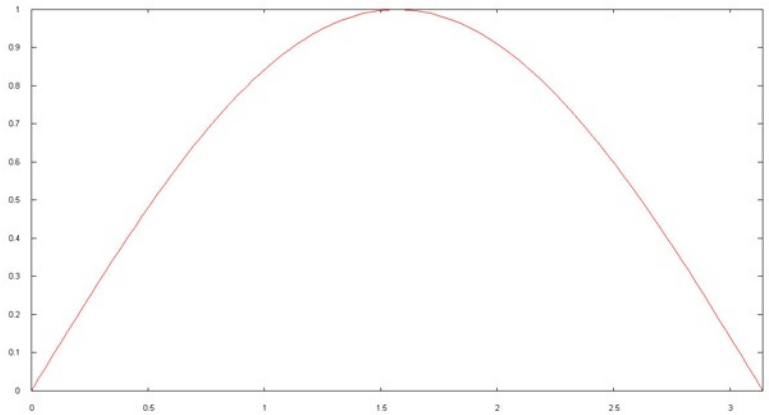
Process P_3 的工作（sin()函數的面積）：1/3

- sin() 函數正的一半的曲線橫座標在0到π之間，縱座標則在0和1之間，這是一個面積為 π 的長方形。

$$\begin{aligned}\int_0^{\pi} \sin(x) &= -\cos(x)|_0^{\pi} \\ &= -(\cos(\pi) - \cos(0)) \\ &= -(-1 - 1) \\ &= 2\end{aligned}$$

計算sin()曲線下的面積

- 如果我們在這個長方形內隨機地選 *s* 個點，若有 *t* 個點落在sin()曲線和*x* 軸之間，那麼sin() 函數曲線下的面積就是長方形面積的*t*/*s*。因為長方形面積是π，因此sin() 和*x*軸所夾著的面積是 $(t/s)\times\pi$ 。



Process P_3 的工作（sin()函數的面積）：2/3

- 我們產生兩個亂數 a 和 b：
  - a 在 $[0, \pi)$，代表橫座標
  - b 在 $[0,1)$，代表縱座標
  - 若$b \leq \sin(a)$，就表示 (a, b)在 `sin()` 函數曲線下方區域內。
- 這道手續反覆做 s 次，若有 t 次產生的點是在 `sin()` 的區域內，於是 `sin()` 涵蓋的面積和外面的長方形面積的比例大致上就是 t/s。
- 因為長方形面積為π， `sin()` 函數曲線下方的面積就是$(t/s) \times \pi$。

Process P_3 的工作（sin()函數的面積）：3/3

- 有關亂數的說明請看Buffon丟針部分。
- s 的值來自執行prog1時命令列的第四個參數。
- 以下是輸出結果，每一列前方得有12個空格。

```
Integration Process Started
Input Number 200000
Total Hit 127498
Estimated Area is 2.0027339
Integration Process Exits
```

Process P_4 的工作（自然對數底e的近似值）：1/4

- 學過微積分的朋友一定都知道$e = 2.71828\cdots$這個值。
- 德國數學家Bernoulli曾經用下式找e的近似值：

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{n} = e = 2.71828\ldots$$

- 若 n 是最大的、上式可能的值。如果 i 從1開始，每次循環時都加倍，而每次循環都用下式計算 e 的近似值：

$$\left(1+\frac{1}{i}\right)^{i}$$

Process P_4 的工作（自然對數底e的近似值）：2/4

- 這個迴圈就從 1 起一直增加，每次都用上式計算。
- 不過程式必須顯示 i 為 1、2、3、…、10這最前面的 10 個值，然後才是加倍、加倍的值；換言之，需要印出上式的 i 值為 1、2、3、…、8、9、10、16、32、64、128、256、…等等。
- 若目前的 i 值大於 n，迴圈就結束。
- P_4 使用命令列上的第一個參數 m 計算 e 的近似值；也就是說，n 的值來自命令列上的第一個參數。

Process P_4 的工作（自然對數底e的近似值）：3/4

- e 的近似值大致如右：
- 這些輸出列前面得要有3個空格。
- 輸出中第一行是 i 的值、第二行是近似值、第三行是近似值和C函數 `exp(1.0)` 的差額。
- 上述的輸出是 m 為 `7300000000000000` 的結果。

```
Approximation of e Process Started
Maximum of the Exponent 7300000000000000
   1 2.00000000000000 0.71828182845904 5
   2 2.25000000000000 0.46828182845904 5
   3 2.37037037037037 0.34791145808867 5
   4 2.44140625000000 0.27687557845904 5
   5 2.48831999999999 0.22996182845904 6
   6 2.52162637174211 0.19665545671693 2
   7 2.54649969704071 0.17178213141833 3
   8 2.56578451395034 0.15249731450869 7
   9 2.58117479171319 0.13710703674584 7
  10 2.59374246010000 0.12453936835904 3
  16 2.63792849736660 0.08035333109244 5
  32 2.67699012937818 0.04129169908086 2
  64 2.69734495256509 0.02093687589394 6
 128 2.70773901968802 0.01054280877102 4
 256 2.71299162425343 0.00529020420561 1
 512 2.71563200016899 0.00264982829005 4
1024 2.71695572946643 0.00132609899260 9
```

接下頁

Process P_4 的工作（自然對數底e的近似值）：4/4

- 因為 n 的值可能非常大，建議在`printf()`中使用`%18lu`，這表示n是個`unsigned long`整數；第二和第三行必須要顯示7到15位有效數字，這表示您得用`double`。

續上頁

```
 8796093022208 2.718281828458891 0.000000000000155
17592186044416 2.718281828458968 0.000000000000077
35184372088832 2.718281828459006 0.000000000000039
70368744177664 2.718281828459026 0.000000000000019
140737488355328 2.718281828459036 0.000000000000009
281474976710656 2.718281828459040 0.000000000000005
562949953421312 2.718281828459043 0.000000000000002
1125899906842624 2.718281828459044 0.000000000000001
2251799813685248 2.718281828459045 0.000000000000000
4503599627370496 2.718281828459045 0.000000000000000
9007199254740992 1.000000000000000 1.718281828459045
18014398509481984 1.000000000000000 1.718281828459045
36028797018963968 1.000000000000000 1.718281828459045
72057594037927936 1.000000000000000 1.718281828459045
```

誤差為0!

為什麼近似值突然變成0？
這是在程式中使用`float`或`double`時必須得知道的。
請仔細想一想！

主程式的工作

- 主程式得印出命令列參數的值、提示已經產生了某個process、等待所有process結束、在所有子process都結束後終結。它的輸出如下；請注意，這些輸出不是一口氣印出來的、而是將要做什麼就印什麼。

這些輸出列前面沒有空白
也就是從第一格開始印

```
Main Process Started
Fibonacci Number 10
Buffon's Needle Iterations = 100000
Integration Iterations = 200000
Approx. e Iterations = 73000000000000000
Fibonacci Process Created
Buffon's Needle Process Created
Integration Process Created
Approximation of e Process Created
Main Process Waits
Main Process Exits
```

一些提示: 1/2

- 雖然每一個process的輸出都有固定順序，但是因為4個process交錯執行，在終端機視窗中主程式和4個process的輸出會交雜出現，這就是為什麼不同process印出結果時前面會加上不同數目空格的理由。
- 請記住`printf()`的問題。
- 記得用`acos(-1.0)`取得π的值

一些提示: 2/2

- 在第一集中我們提到過，使用亂數得引入`stdlib.h`。
- 用`rand()`產生一個整數亂數，它的值在`0`和`RAND_MAX`之間，但本題的亂數都在 [0,1) 之間，得用`(float rand())/RAND_MAX`轉換到[0,1)。`RAND_MAX`是在`limits.h`中定義的。
- 您可以在需要使用亂數的process中用`srand(time(NULL))`起動一系列的亂數，再用`rand()`產生整數亂數。
- 若您用**GNU C**，下面的編譯連結是最基本的做法：

```
gcc prog1.c -std=c89 -lm -o prog1
```

- 再用下列測試您的程式、產生和此地相似的結果:

```
./prog1 7300000000000000 10 100000 200000
```

我們學到了什麼?

- 用直覺方式定義 process
- 討論作業系統表示和處理process的方式
- CPU調度process和環境切換
- 多process程式寫作 (`fork()` 、`exit()` 和`wait()`)
- 下一集會討論進一步的多process程式寫作
- 頭會愈來愈痛，😆

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