請以純粹黑白列印

DLL ADVANCED TECHNIQUES

井民全製作

Explicit DLL Module Loading and Symbol Linking

- 若要呼叫 DLL 的function,你必須先把 DLL map 到該 process 的 address space
  - 方法 1 (implicitly load): load-time dynamic linking
    - 當 thread 呼叫該 function 之前, Loader 就必須先把所有需要的DLL,全部載入
  - 方法 2 (explicit load): run-time dynamic linking
    - 需要時才載入
    - Thread 可以下指令將需要的 DLL 載到記憶體,並且得到呼叫函式的 virtual address, 執行呼叫

DLL export 的部分

使用 DLL 的部分

export 函式原型,結構 Symbol 的 header file

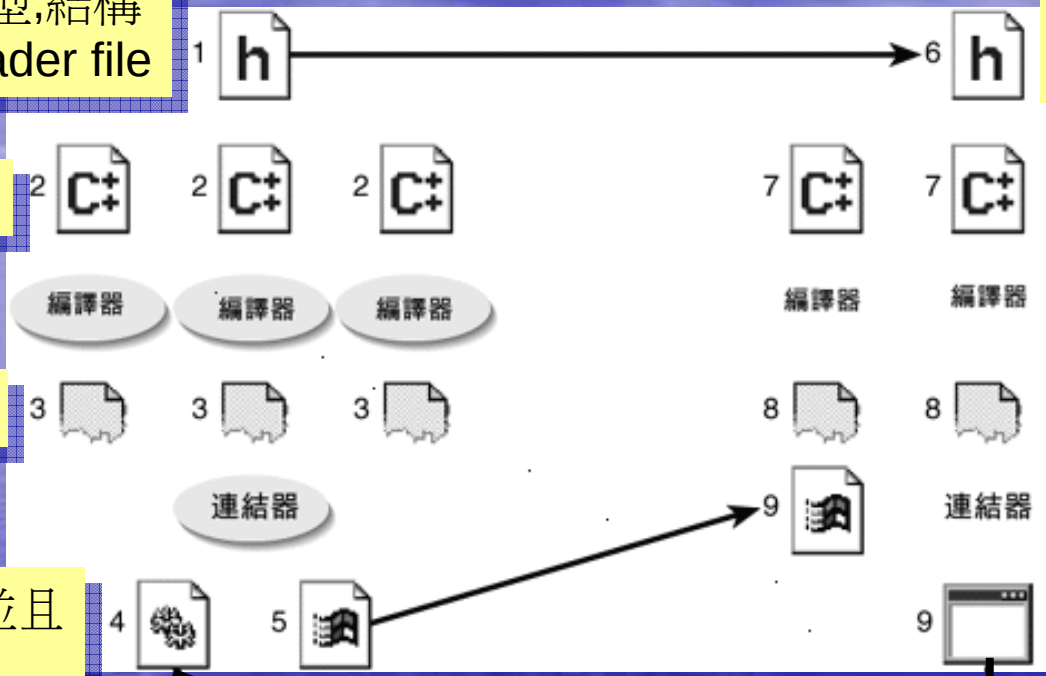
Import 你要使用的函式原型

函式的實作

不直接參考 DLL 中的 function

產生 obj 檔

產生 obj



組合成 DLL 檔並且產生 lib 檔

將所有的 obj 組合成 exe 執行檔

10. Loader 爲 exe 模組建立 address space,並且執行

11. Thread 呼叫 LoadLibrary() 將 DLL 載入並且呼叫 GetProcAddress()取得函式的位址

因爲 executable 的部分並沒有直接使用 DLL function, 故不需要 **.lib** 檔

Explicitly Loading the DLL Module

- Process 執行期間,可以呼叫

```
HINSTANCE LoadLibrary (PCTSTR pszDLLPathName);
```

DLL 檔案名稱

```
HINSTANCE LoadLibraryEx(
                        PCTSTR pszDLLPathName,
                        HANDLE hFile,
                        DWORD dwFlags);
```

保留, 必須爲 NULL

DONT_RESOLVE_DLL_REFERENCE: 系統不會呼叫 DllMain 並且也不會自動載入該 DLL 所 reference 的其他 DLL

LOAD_LIBRARY_AS_DATAFILE: 單純的視 DLL 爲 data

Explicitly Unloading the DLL Module

- 使用 FreeLibrary(HINSTANCE hinstDll) 釋放佔用的空間

由 LoadLibrary 傳回來的 handle

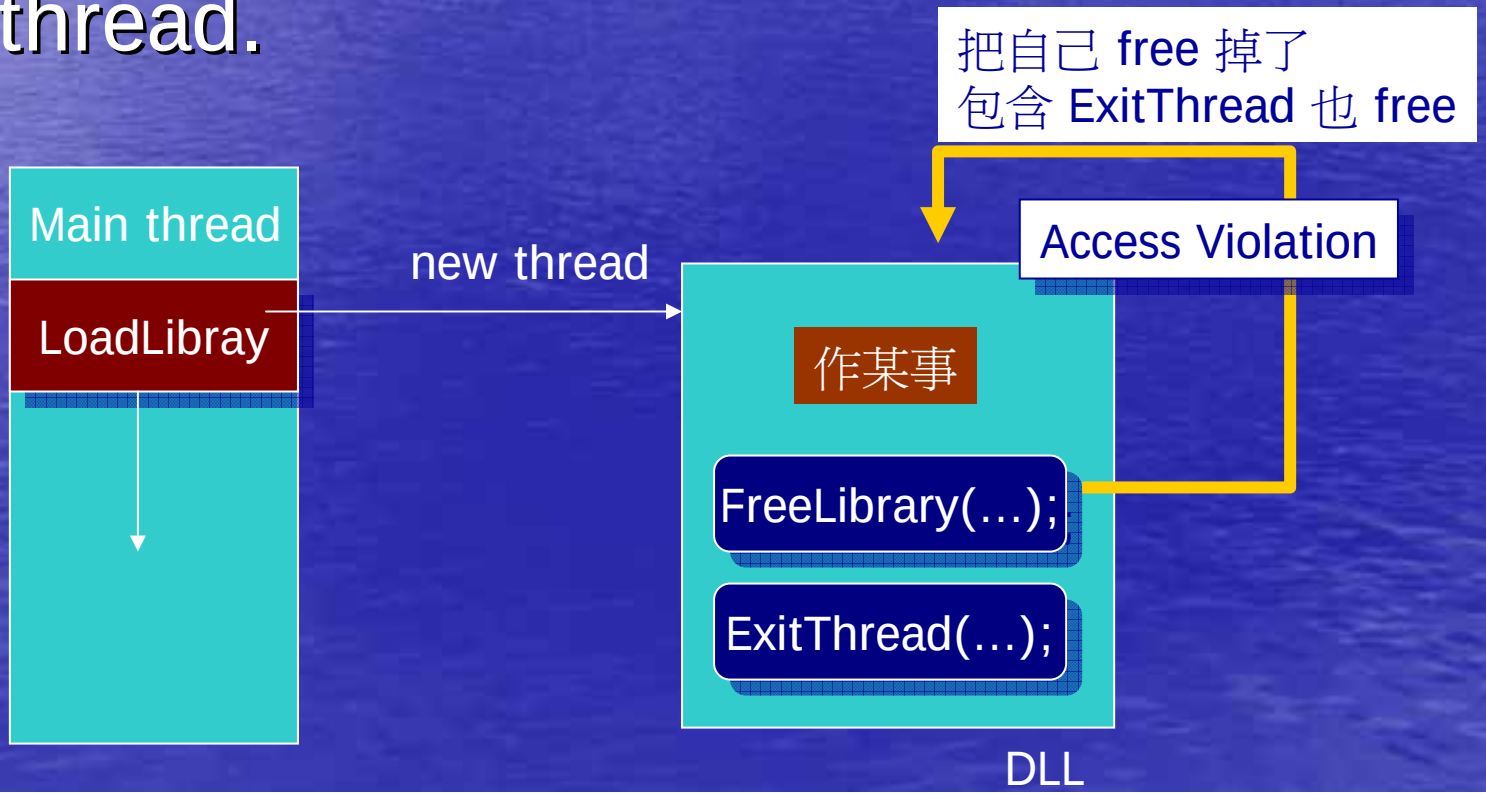
- VOID **FreeLibraryAndExitThread**(HINSTANCE hinstDll,
  DWORD dwExitCode);

爲何不寫成

FreeLibrary(hinstDll);
ExitThread (dwExitCode);

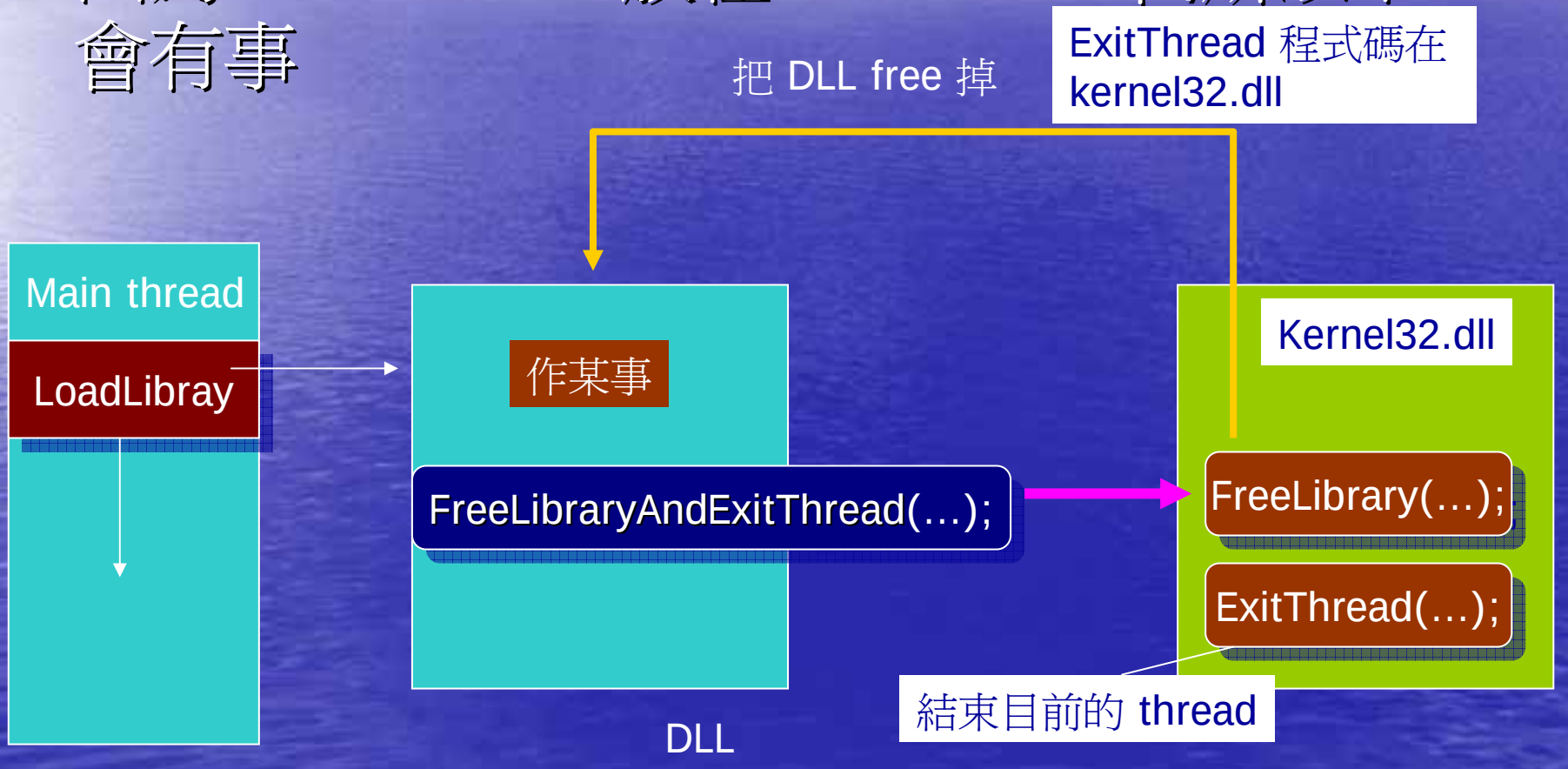
情況是這樣的

- 若你希望你的 DLL 建立一個新的 thread 來做某些事情, 當作完後,自動 free DLL 並且停止 thread.



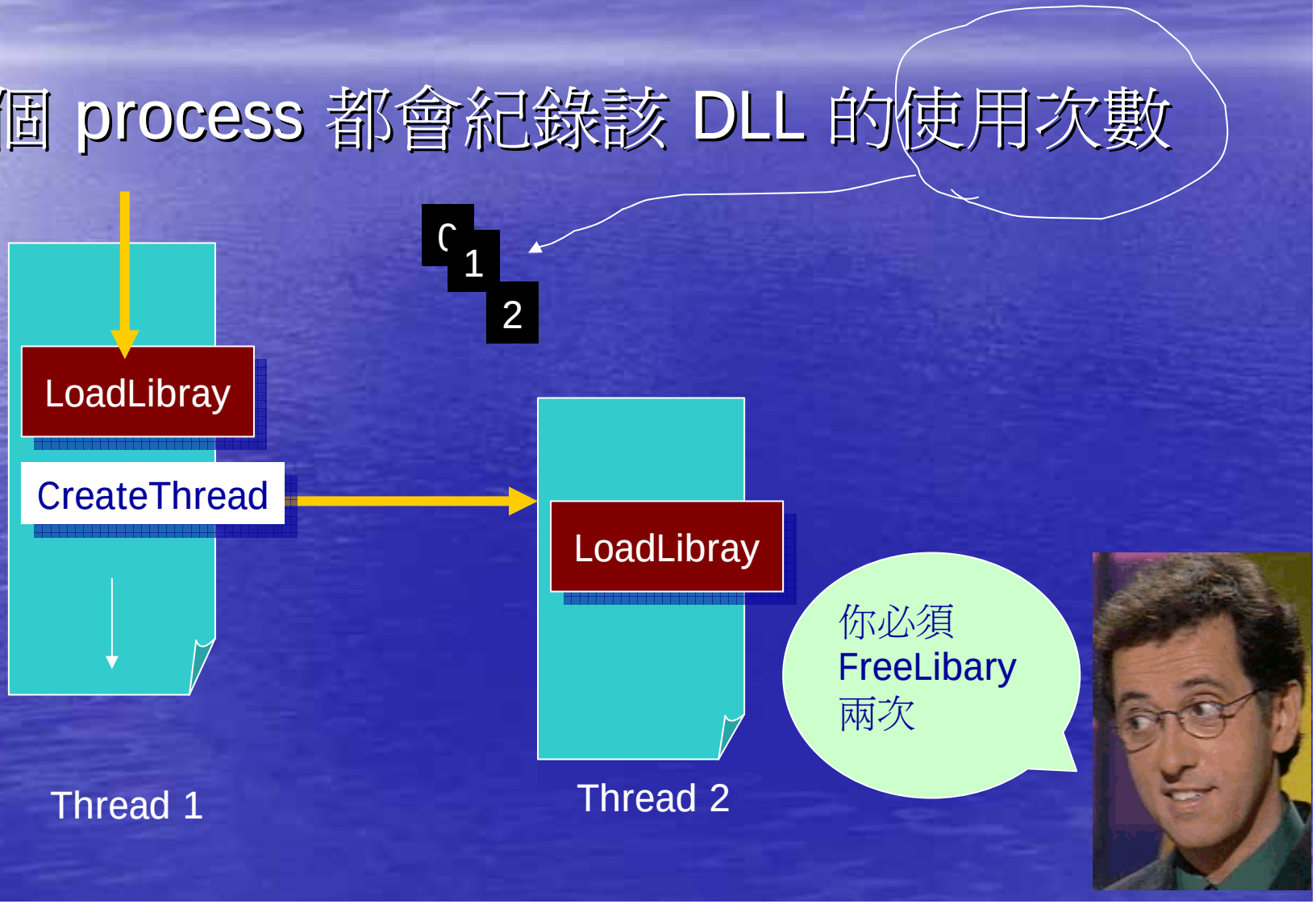
如果呼叫FreeLibraryAndExitThread

- 因爲 ExitThread 放在 Kernel32 中,所以不會有事

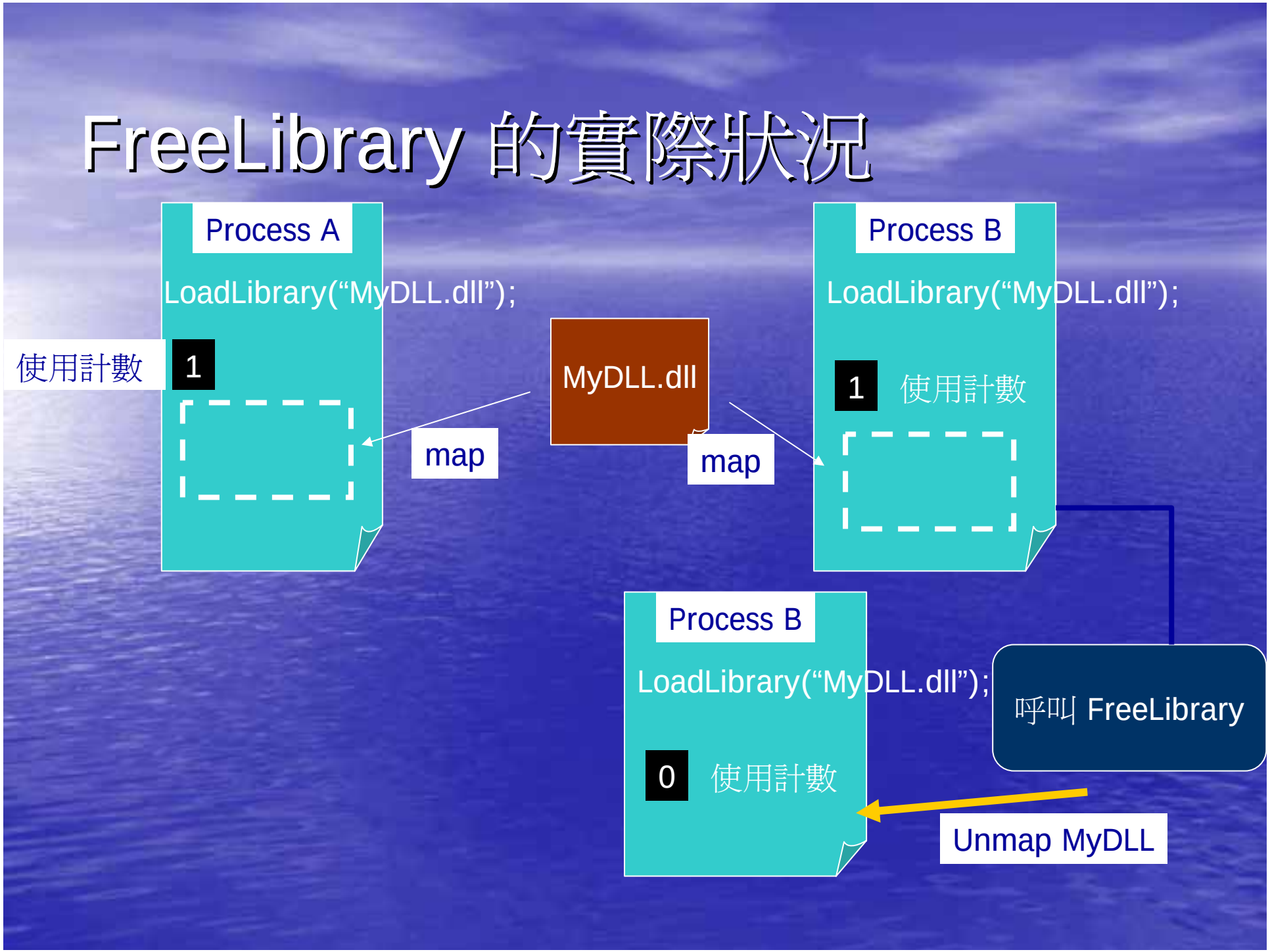


FreeLibrary 的實際狀況

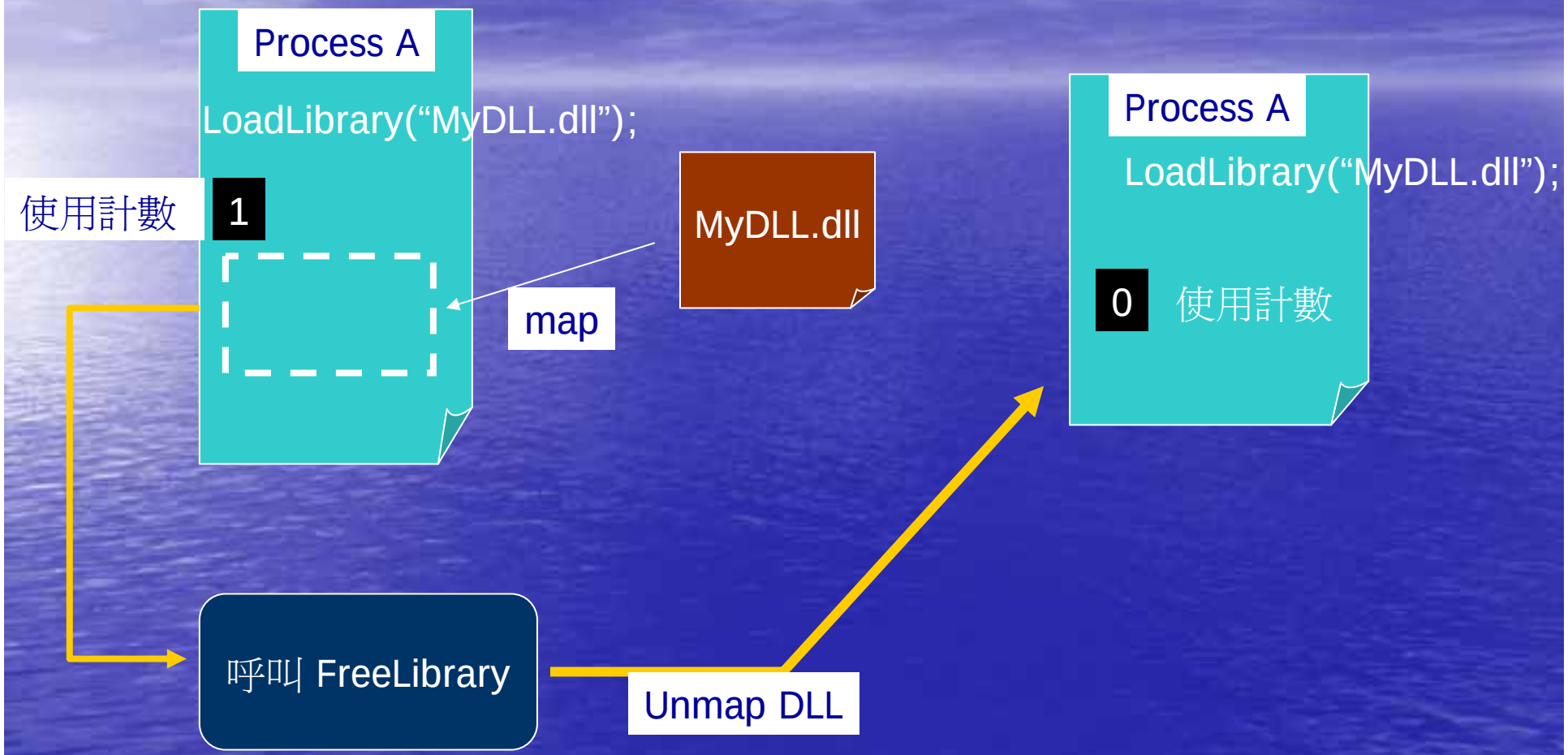
- 每個 process 都會紀錄該 DLL 的使用次數



FreeLibrary 的實際狀況



FreeLibrary 的實際狀況



檢查 MyDLL 是否已經被載入

- 使用 GetModuleHandle

```
HINSTANCE hinstDll = GetModuleHandle("MyLib");
if (hinstDll = = NULL) {
   hinstDll = LoadLibrary("MyLib");}
```

判斷 MyLib.dll 是否已經被其他的 Thread 載入到 process 的位址空間

若沒有的時候,我們才載入 MyLib.dll

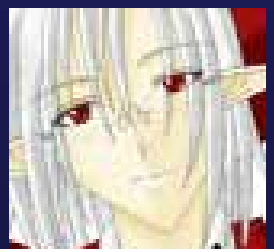
取得某個 DLL 的完整路徑

```
#include "stdafx.h"
#include <windows.h>          // for win32AIP
#include <iostream>           // for cout
using namespace std;
void ShowError();

int _tmain(int argc, _TCHAR* argv[]){
        HINSTANCE hinstDll = GetModuleHandle("MyDLL");
        if (hinstDll == NULL) {
                hinstDll = LoadLibrary("MyDLL");
                if(hinstDll==NULL)
                        ShowError();
        }
        LPTSTR Buffer=new TCHAR[100];
        DWORD size=GetModuleFileName(hinstDll,Buffer,100);
        cout << Buffer << endl;
        getchar();

        return 0;
}
```

別忘了把 MyDLL.dll 放到可搜尋的目錄中

讀取目前 MyDLL.dll 的完整路徑

如果查詢失敗,則size == 0



typedef declarations

- typedef 指令建立 synonym(等義)而非新的 type, 目的是幫助簡化宣告

```
typedef unsigned char BYTE;    // 8-bit unsigned entity.
typedef BYTE *        PBYTE;   // Pointer to BYTE.

BYTE Ch;                       // Declare a variable of type BYTE.
PBYTE pbCh;                    // Declare a pointer to a BYTE
                               //  variable.
```

基本 type 爲 unsigned char

基本 type 爲 unsigned char*

Declare a type name

representing a pointer to a function

```
typedef void (*PVFN)();
```

定義 PVFN 爲一個type name,
表示爲指向 沒有參數與回傳值的函數指標

```
void func1(){};
void func2(){};
typedef void (*PVFN)();
int main()
{
   // Declare an array of pointers to functions.
   PVFN pvfn[] = { func1, func2 };

   // Invoke one of the functions.
   (*pvfn[1])();
}
```

Function's prototype

利用 typedef,
宣告 function
pointer 陣列變
的非常簡單

Explicitly Linking to an Exported Symbol

如果你的 DLL 使用 __stdcall call typedef void **(__stdcall** *PVFN)();

```
#include <windows.h>
typedef int (*MYPROC)(int,int);

int _tmain(int argc, _TCHAR* argv[]){
        HINSTANCE hinstDll = GetModuleHandle("MyDLL");
        if (hinstDll == NULL) {
                hinstDll = LoadLibrary("MyDLL");
                if(hinstDll==NULL)
                        ShowError();
        }
        MYPROC ProcAdd;
        ProcAdd=(MYPROC)GetProcAddress(hinstDll,"Add");
        if(ProcAdd==NULL)
                ShowError();

        int data=(ProcAdd)(2,4);

        // Step 4: Free the DLL module
        FreeLibrary(hinstDll);

        return 0;
}
```

定義 MYPROC 爲一個 type name, 表示我們想呼叫function 的pointer

Step 1: 將 DLL 載入

以 DLL 的 export 節區定義爲準 (若 DLL 使用 stdcall 且沒有 . DEF) 則要用 _Add@8 呼叫

Step 2: 宣告 ProcAdd 變數用來存放 function Add 的指標

注意轉型

把 Add function 取出來

Step 3: 呼叫 Add 範例

別忘了把 MyDLL.dll 放到可搜尋的目錄中

The DLL's Entry-Point Function

- 每一個 DLL 都有一個 Entry-point function
  - DllMain
- 系統會在不同的時機下,呼叫 DllMain

你也可以不實作
DllMain

情況是這樣的

DLL 所在的 virtual address

指示是否爲
explicitly link ➔ 0
Implicitly link➔ 非 0 值

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {
   switch (fdwReason) {
      case DLL_PROCESS_ATTACH:

      break;

      case DLL_THREAD_ATTACH:
                                  // A thread is being created.
      break;

      case DLL_THREAD_DETACH:
                                  // A thread is exiting cleanly.
                                  break;

      case DLL_PROCESS_DETACH:

       break;
   }
   return(TRUE);  // Used only for DLL_PROCESS_ATTACH
}
```

當 DLL 被 mapped 到 Process的 virtual address space

當 DLL 由 process 的 virtual address space unmapped 出來時

注意事項

- DllMain ➔ 用在 initial 本身, 例如設定全區域變數的初值等.
  - 注意 1: 執行時位於同一位址空間的其他 DLL 可能尚未執行自己的 DllMain.

你應該避免在 DllMain 中呼叫其他 DLL export 的 function

  - 注意 2: 應該避免呼叫 LoadLibrary 或 FreeLibrary, 以防止產生自我呼叫的 loop
  - 注意 3: 避免呼叫 User、Shell、ODBC、COM、RPC、socket 等元件的 function, 因爲他們可能尚未被初始化

The DLL_PROCESS_ATTACH Notification

- 當 DLL 第一次被 map 到 process's address space 時就會被呼叫
  - 若有同一個process中的另一個 thread 又呼叫 LoadLibrary, 則 DLL 只會被計數一次

並不會呼叫 DllMain

- 使用方式
  - 若你的 **DLL** 想使用自己 **Heap** 當作儲存空間

在 DLL 中設定 global 變數 存放 Heap 的 Handle
➔ 在 DLL_PROCESS_ATTACH 建立 Heap

初始化 DLL 失敗的情況

- 若 DLL_PROCESS_ATTACH 情況下
  - FALSE

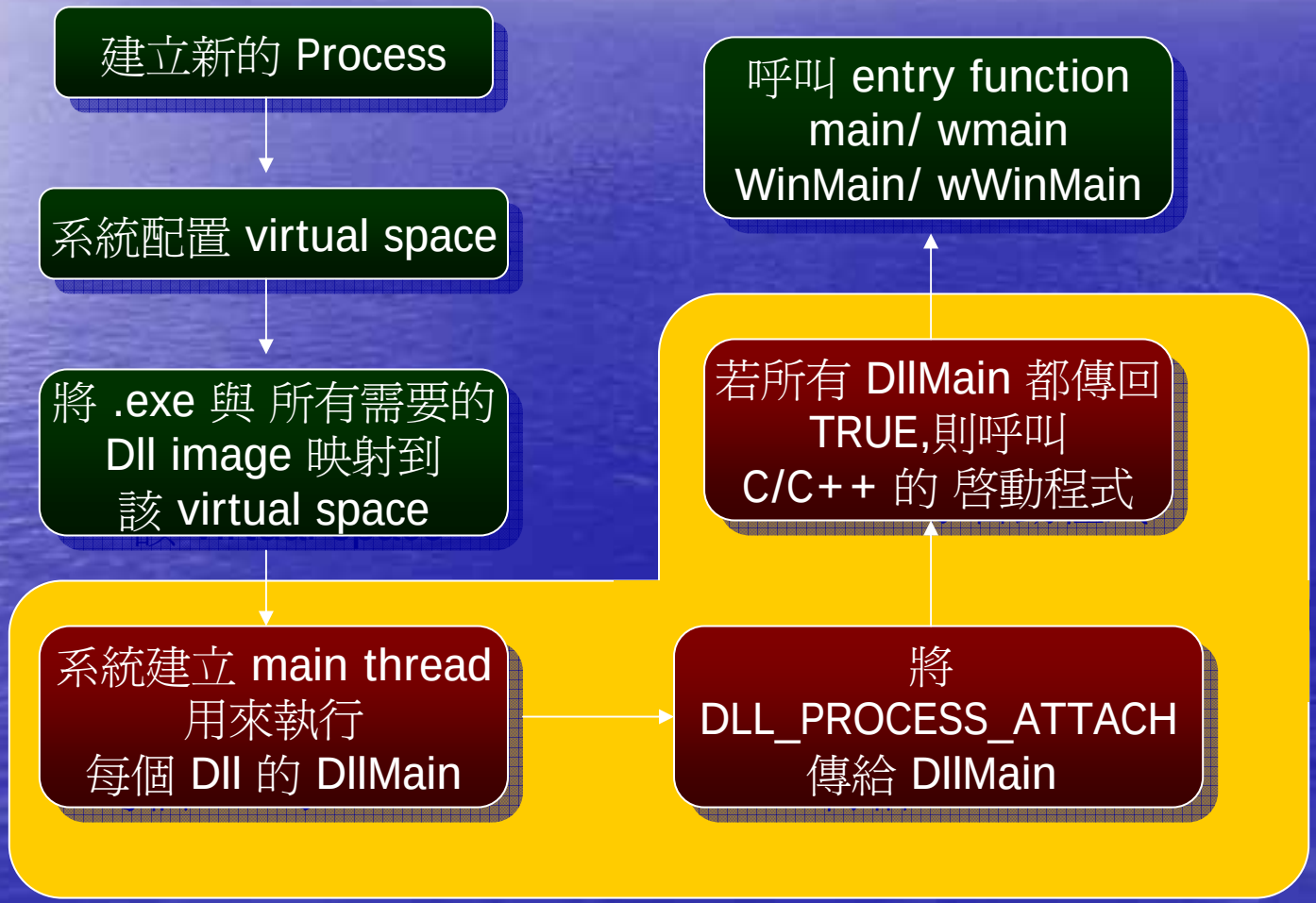
Implicitly 載入的情況


```
BOOL WINAPI DllMain(...){
    switch (fdwReason) {
      case DLL_PROCESS_ATTACH:
            return(FALSE);
        break;
    }
}
```

其他的情況，系統會 *忽略* ***DllMain*** 傳回來的值

系統執行 DllMain 的流程

Implicity linking 的流程



Explicitly linking 的流程

```
void main(){
   HINSTANCE hinstDll;
   hinstDll = LoadLibrary("MyDLL");
}
```

LoadLibrary 處理流程

LoadLibrary 返回

若 DllMain() ➔ FALSE
則 return NULL

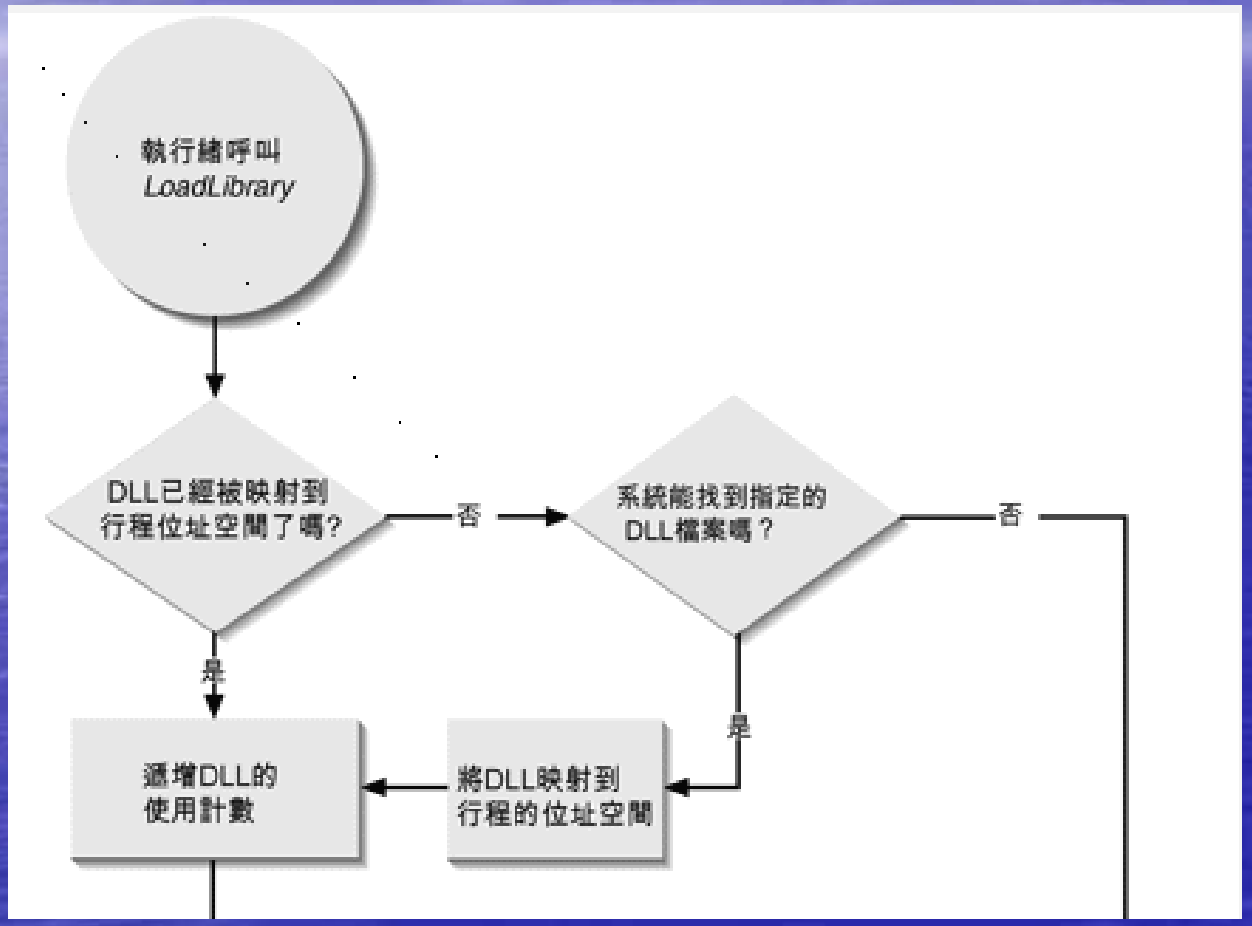
DLL_PROCESS_DETACH

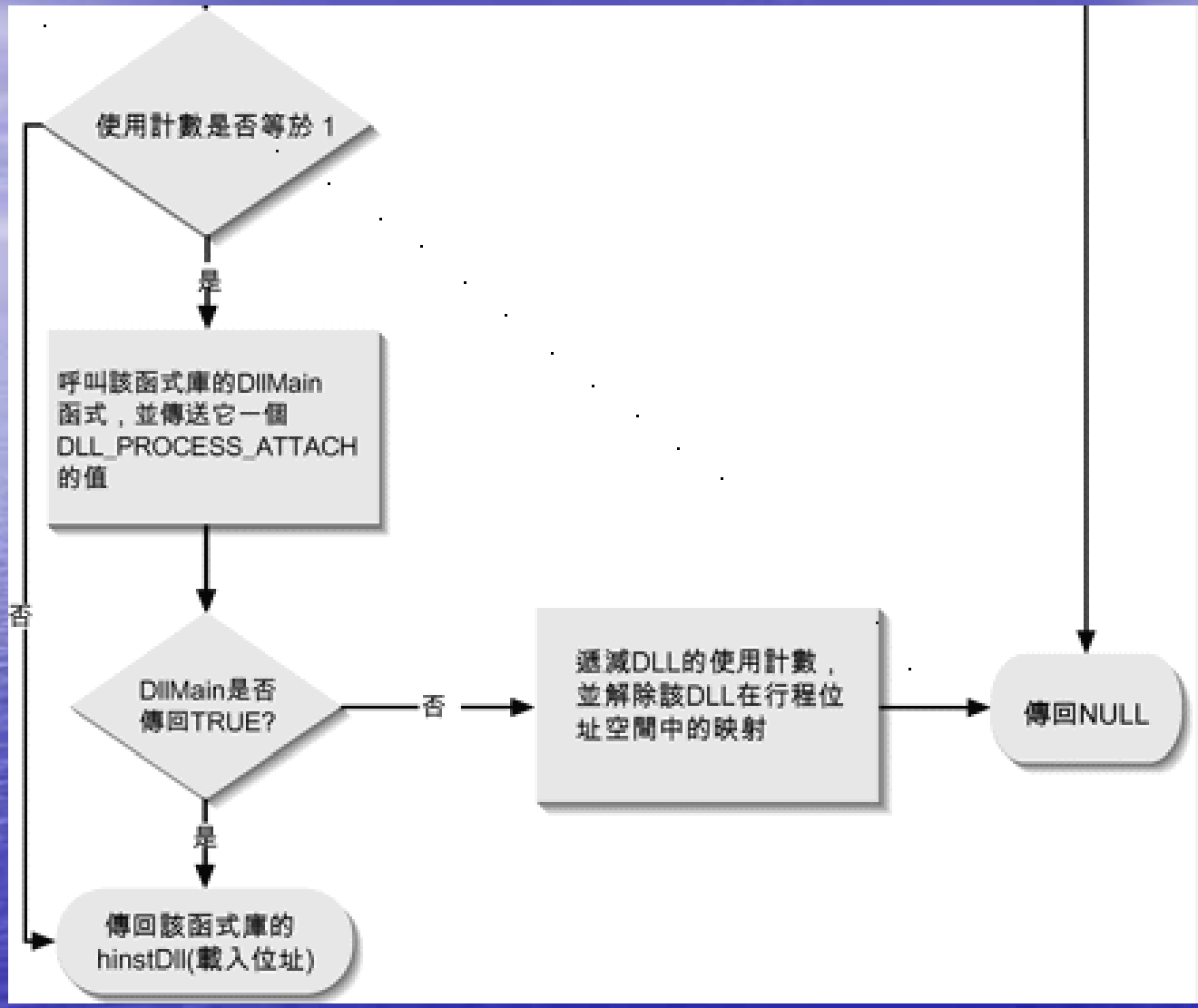
- 當 Dll 被解除映射時,會被通知
- 使用時機
  - 若你在 ATTACH 時建立 heap, 則你在 DETACH HeapDestroy 這個 heap
- 下列情況下, 系統不會通知你
  - 當有人呼叫 TerminateProcess
  - 若某個其他 DLL 在 ATTATCH 狀態下,傳回 FALSE, 則不會呼叫這個 DLL

所以 terminate process 請用 ExitProcess

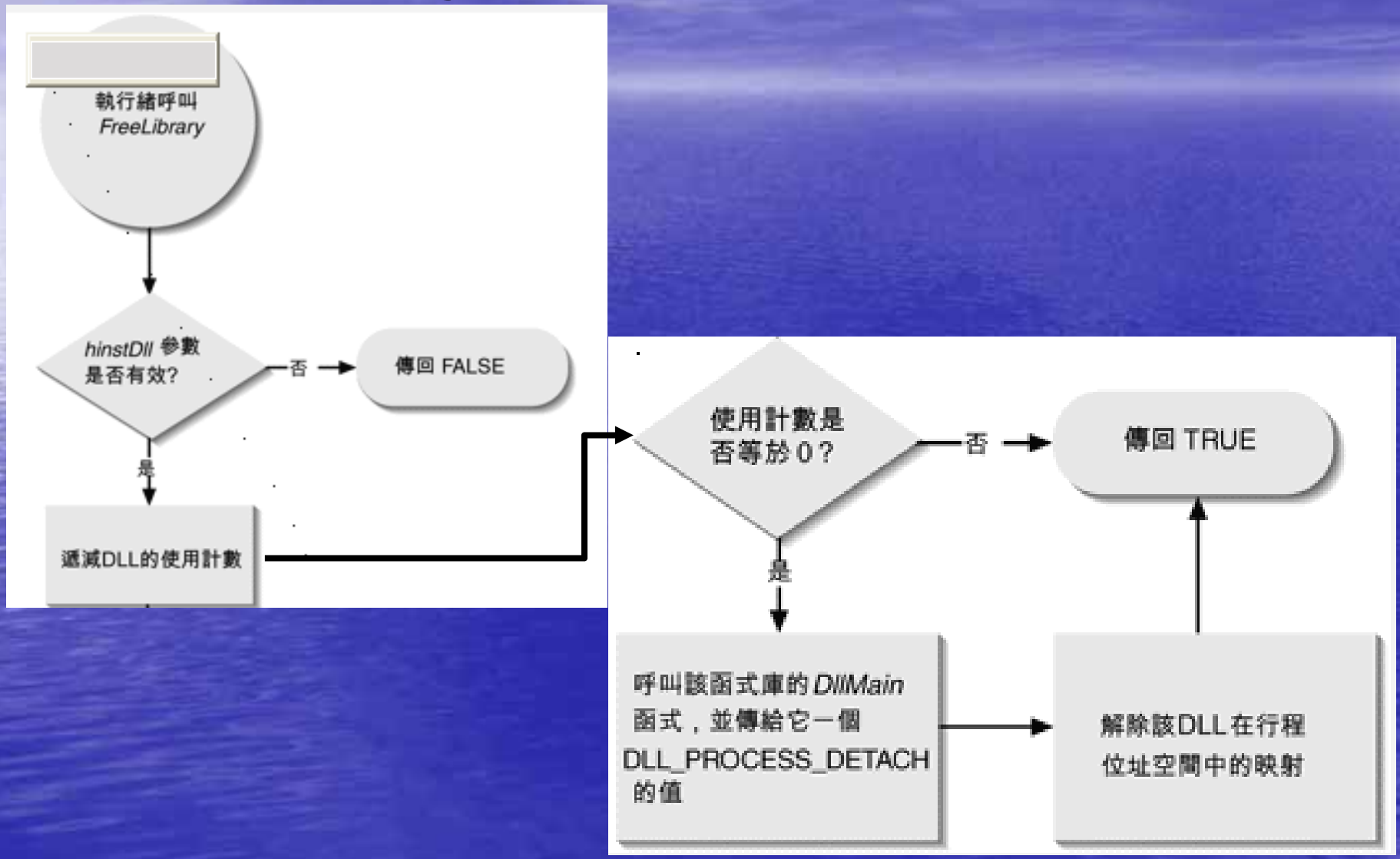
```
void main(){
   HINSTANCE hinstDll;
   hinstDll = LoadLibrary("MyDLL");
  TerminateProcess(GetCurrentProcess(),1);
}
```

呼叫 LoadLibrary 的流程




使用計數是否等於 1
是
呼叫該函式庫的DllMain
函式，並傳送它一個
DLL_PROCESS_ATTACH
的值
否
DllMain是否
傳回TRUE?
否
遞減DLL的使用計數，
並解除該DLL在行程位
址空間中的映射
傳回NULL
是
傳回該函式庫的
hinstDll(載入位址)

FreeLibrary 的流程



DLL_THREAD_ATTACH

- 當你的 Process 建立了新的 thread
  - 所有該 process 使用的 DLL 都會被呼叫

DLL_THREAD_DETACH

- 當 thread function 返回時, 會呼叫 ExitThread ➔ 呼叫所有 DLL 通知有 thread 已經離開

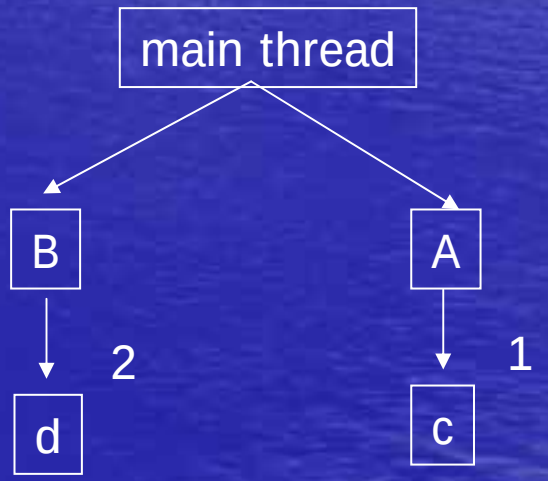
TerminateThread 不會有通知的動作

Serialized calls to DllMain

- 我們都知道每次建立新的 thread 時,都會呼叫 DllMain,並且得到DLL_THREAD_ATTACH 通知

但是, 有沒有可能多個 thread 同時在 DllMain 中執行呢 ?

No




```
DllMain(){
}
```

d 必須等待 c 完成 DllMain 的呼叫

Bug

不要在 DllMain中
呼叫 WaitForSingleObject

```
BOOL WINAPI DllMain( … ) {
   HANDLE hThread;
   DWORD dwThreadId;

   switch (fdwReason) {
   case DLL_PROCESS_ATTACH:

      hThread = CreateThread(NULL, 0, SomeFunction, NULL,
       0, &dwThreadId);

      WaitForSingleObject(hThread, INFINITE);

      // We no longer need access to the new thread.
      CloseHandle(hThread);
      break;
 …
}
```

Step 1: 建立 thread

Step 2: Suspend 目前的 thread,直到
新的thread 完成工作

發生了死結
因爲 新的 thread
無法執行 DllMain

DllMain 與 C/C++ runtime library

現在的情況是 若在 DllMain 外面有一些 global 物件或 static 物件. 那麼這些 物件的 constructor 與 destructor 要如何被呼叫?

```
#include "horses.h"

CHorse  Equus( ARABIAN, MALE );
CHorse  Sugar( THOROUGHBRED, FEMALE );

BOOL    WINAPI   DllMain (HANDLE hInst,
                          ULONG ul_reason_for_call,
                          LPVOID lpReserved)
```

兩個 global 物件,並呼叫建構子

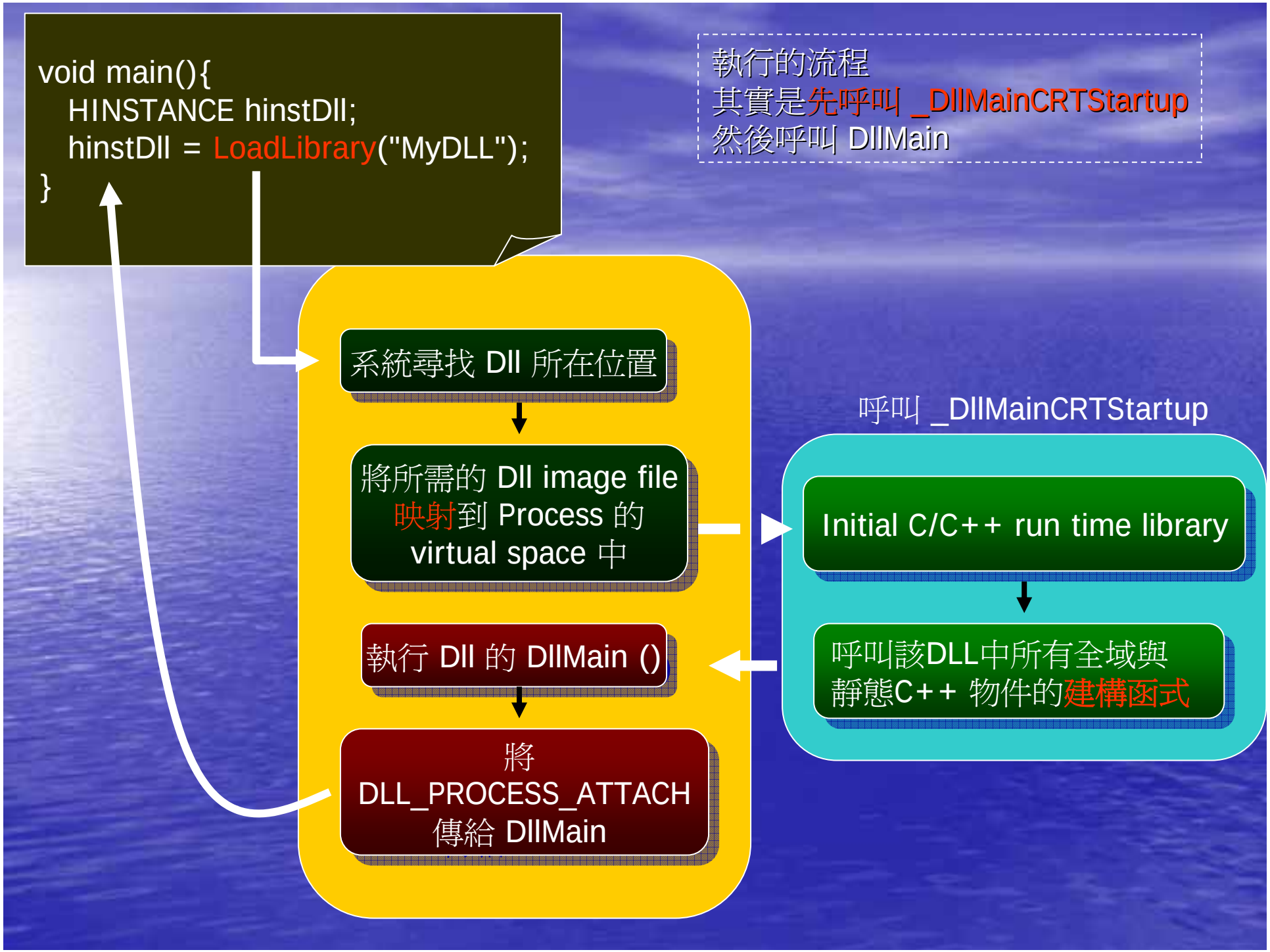
_DllMainCRTStartup function

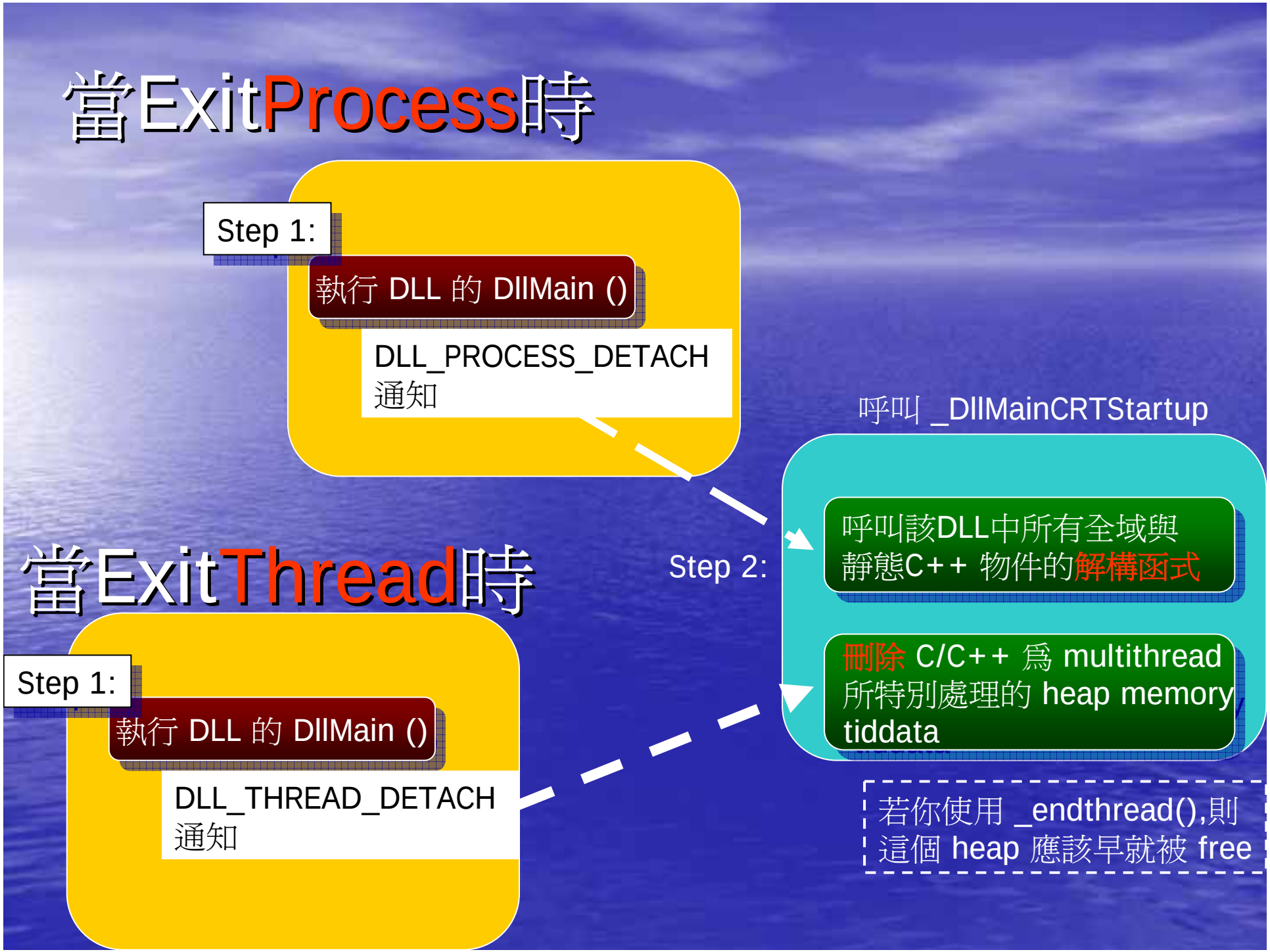
- 其實, 在你 link DLL 時, linker 會自動幫你加入一個初始化C++物件的function

_DllMainCRTStartup()

```
void main(){
   HINSTANCE hinstDll;
   hinstDll = LoadLibrary("MyDLL");
}
```

系統尋找 Dll 所在位置


void main(){
HINSTANCE hinstDll;
hinstDll = LoadLibrary("MyDLL");
}
執行的流程
其實是先呼叫 _DllMainCRTStartup
然後呼叫 DllMain
系統尋找 Dll 所在位置
將所需的 Dll image file 映射到 Process 的 virtual space 中
執行 Dll 的 DllMain ()
將 DLL_PROCESS_ATTACH 傳給 DllMain
呼叫 _DllMainCRTStartup
Initial C/C++ run time library
呼叫該DLL中所有全域與靜態C++ 物件的建構函式


當ExitProcess時
Step 1:
執行 DLL 的 DllMain ()
DLL_PROCESS_DETACH
通知
呼叫 _DllMainCRTStartup
Step 2:
呼叫該DLL中所有全域與
靜態C++ 物件的解構函式
刪除 C/C++ 爲 multithread
所特別處理的 heap memory
tiddata
若你使用 _endthread(),則
這個 heap 應該早就被 free
當ExitThread時
Step 1:
執行 DLL 的 DllMain ()
DLL_THREAD_DETACH
通知

若你沒有自己寫 DllMain

- Linker 會幫你加上預設 DllMain

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad)
{
   if (fdwReason == DLL_PROCESS_ATTACH)
      DisableThreadLibraryCalls(hinstDll);
   return(TRUE);
}
```

當 thread 被建立或刪除時，不需要呼叫 DllMain

也就是 DLL_THREAD_ATTACH 與 DLL_THREAD_DETACH 的 notification 會被 disable ！
這樣,會使程式的大小縮小. 尤其是對於那些 multithread 又使用 DLL 的 server 程式

- End