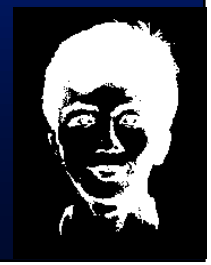
Win32 Programming

- Overview
- Win32 Programming Model
  - WinMain
  - window procedure
  - window class & windows
  - msg & msgLoop
- KERNEL
  - Address Space
  - Memory Management
  - Module, Processes, Threads

侯捷

2009/11/26-27

騰訊



Win32 Project in Visual Studio 2003

獲得

所有檔案內容見後

以 VC6 和 VS 2003 的 IDE 自動化生成的 Win32 程式骨幹，
除了 VS 2003 的 comments 中文化之外，兩者一模一樣。

Win32 Project

Readme.txt

```
========================================================================
    WIN32 應用程式 : w32App 專案概觀
========================================================================
AppWizard 已經爲您建立了這個 w32App 應用程式。
這個檔案含有構成 w32App 應用程式之每一個檔案的內容摘要。
w32App.vcproj
    這是使用應用程式精靈所產生之 VC++ 專案的主要專案檔。
    它含有產生該檔案之 Visual C++  的版本資訊，以及有關使用應用程式精靈所選取之平台、組態和專案功能的資訊。
w32App.cpp
    這是主應用程式原始程式檔。
/////////////////////////////////////////////////////////////////////////////
AppWizard 已經建立了下列資源:
w32App.rc
    這是程式所用的所有 Microsoft Windows 資源的列表。它含有儲存在  RES 子目錄中的圖示、點陣圖和資料指標。
    這個檔案可以直接在 Microsoft Visual C++ 中編輯。
Resource.h
     這是定義新資源 ID 的標準標頭檔。
    Microsoft Visual C++ 會讀取和更新這個檔案。
w32App.ico
    這是用來做爲應用程式圖示 (32x32) 的圖示檔。
    這個圖示是包含在主要資源檔 w32App.rc 中。
small.ico
    這是含有較小型版本 (16x16) 應用程式圖示的檔示檔。
    這個圖示是包含在主要資源檔 w32App.rc 中。
/////////////////////////////////////////////////////////////////////////////
其他標準檔案:
StdAfx.h, StdAfx.cpp
    這些檔案是用來建置名爲 w32App.pch 的先行編譯標頭 (PCH) 檔
    和名爲 StdAfx.obj 的先行編譯型別檔。
/////////////////////////////////////////////////////////////////////////////
其他注意事項:
AppWizard 使用 "TODO:" 註解來指示您應該加入或自訂的原始程式碼部分。
/////////////////////////////////////////////////////////////////////////////
```

W32App.h

```
#pragma once

#include "resource.h“
```

Win32 Project

resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by w32App.rc
//
#define IDS_APP_TITLE                   103
#define IDR_MAINFRAME                   128
#define IDD_W32APP_DIALOG               102
#define IDD_ABOUTBOX                    103
#define IDM_ABOUT                       104
#define IDM_EXIT                        105
#define IDI_W32APP                      107
#define IDI_SMALL                       108
#define IDC_W32APP                      109
#define IDC_MYICON                      2
#define IDC_STATIC                      -1
// 下一個新增物件的預設值
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                     130
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           110
#endif
#endif
```

W32App.rc

1

```
// Microsoft Visual C++ 產生的資源指令碼。
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
// 已從 TEXTINCLUDE 2 資源產生。
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHT)
LANGUAGE 4, 1
#pragma code_page(950)
//////////////////////////////////////////////////////////////////////////////
// 圖示
// 先放置有最低 ID 的圖示，確保應用程式圖示在所有系統上保持一致。
IDI_W32APP      ICON        "w32App.ico"
IDI_SMALL       ICON        "small.ico"
//////////////////////////////////////////////////////////////////////////////
// 功能表
IDC_W32APP MENU
BEGIN
    POPUP "檔案(&F)"
    BEGIN
        MENUITEM "結束(&X)",              IDM_EXIT
    END
    POPUP "說明(&H)"
    BEGIN
        MENUITEM "關於(&A) ...",          IDM_ABOUT
    END
END
```

Stdafx.cpp

```
// stdafx.cpp : 僅包含標準 Include 檔的原始程式檔
// w32App.pch 會成爲先行編譯標頭檔
// stdafx.obj 會包含先行編譯型別的資訊
#include "stdafx.h"
// TODO: 請在 STDAFX.H 中參考您需要的任何其他標頭，而不要在這個檔案中參考
```

W32App.rc 2

```
//////////////////////////////////////////////////////////////////////////////
// 對應鍵
IDC_W32APP ACCELERATORS
BEGIN
    "?",            IDM_ABOUT,              ASCII,  ALT
    "/",            IDM_ABOUT,              ASCII,  ALT
END
//////////////////////////////////////////////////////////////////////////////
// 對話方塊
IDD_ABOUTBOX DIALOG   22, 17, 230, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "關於"
FONT 9, "System"
BEGIN
    ICON            IDI_W32APP,IDC_MYICON,14,9,16,16
    LTEXT           "w32App Version 1.0",IDC_STATIC,49,10,119,8,SS_NOPREFIX
    LTEXT           "Copyright (C) 2009",IDC_STATIC,49,20,119,8
    DEFPUSHBUTTON   "確定",IDOK,195,6,30,11,WS_GROUP
END
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE
BEGIN
    "resource.h\0"
END
2 TEXTINCLUDE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include ""windows.h""\r\n"
    "#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "\0"
END
3 TEXTINCLUDE
BEGIN
    "\r\n"
    "\0"
END
#endif    // APSTUDIO_INVOKED
```

W32App.rc 3

```
//////////////////////////////////////////////////////////////////////////////
// 字串資料表
STRINGTABLE
BEGIN
  IDC_W32APP   "W32APP"
  IDS_APP_TITLE  "w32App"
END
#endif
//////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// 已從 TEXTINCLUDE 3 資源產生。
//
//////////////////////////////////////////////////////////////////////////////
#endif    // 非 APSTUDIO_INVOKED
```

Stdafx.h

```
// stdafx.h : 可在此標頭檔中包含標準的系統 Include 檔，
// 或是經常使用卻很少變更的專案專用 Include 檔案
//
#pragma once
#define WIN32_LEAN_AND_MEAN  // 從 Windows 標頭排除不常使用的成員
// Windows 標頭檔:
#include <windows.h>
// C RunTime 標頭檔
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
// TODO: 在此參考您的程式所需要的其他標頭
```

Win32 Project

W32App.cpp

```
#0001 // w32App.cpp : 定義應用程式的進入點。
#0002 //
#0003
#0004 #include "stdafx.h"
#0005 #include "w32App.h"
#0006 #define MAX_LOADSTRING 100
#0007
#0008 // 全域變數:
#0009 HINSTANCE hInst;                                  // 目前執行個體
#0010 TCHAR szTitle[MAX_LOADSTRING];                    // 標題列文字
#0011 TCHAR szWindowClass[MAX_LOADSTRING];              // 主視窗類別名稱
#0012
#0013 // 這個程式碼模組中所包含之函式的向前宣告:
#0014 ATOM     MyRegisterClass(HINSTANCE hInstance);
#0015 BOOL     InitInstance(HINSTANCE, int);
#0016 LRESULT CALLBACK     WndProc(HWND, UINT, WPARAM, LPARAM);
#0017 LRESULT CALLBACK     About(HWND, UINT, WPARAM, LPARAM);
#0018
#0019 int APIENTRY _tWinMain(HINSTANCE hInstance,
#0020                        HINSTANCE hPrevInstance,
#0021                        LPTSTR    lpCmdLine,
#0022                        int       nCmdShow)
#0023 {
#0024     // TODO: 在此置入程式碼。
#0025     MSG msg;
#0026     HACCEL hAccelTable;
#0027
#0028     // 初始化全域字串
#0029     LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
#0030     LoadString(hInstance, IDC_W32APP, szWindowClass, MAX_LOADSTRING);
#0031     MyRegisterClass(hInstance);
#0032
#0033     // 執行應用程式初始設定:
#0034     if (!InitInstance (hInstance, nCmdShow))
#0035     {
#0036         return FALSE;
#0037     }
#0038
#0039     hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_W32APP);
#0040
```

```
#ifdef _UNICODE
/* ++++++ UNICODE ++++++ */
…
#define __T(x)      L ## x
#define _tmain      wmain
#define _tWinMain   wWinMain

#else  /* _UNICODE */
/* +++++ SBCS and MBCS +++++ */
…
#define __T(x)      x
#define _tmain      main
#define _tWinMain   WinMain
```

hPrevInstance 未用到。**Win95 之前**應該這樣：
if(!hPrevInstance) MyRegisterClass(hInstance);
惟第一個instance才需註冊視窗類別。
若程式只允許執行唯一instance，
可這樣：if(hPrevInstance) return;
例如，MS Word 允許執行多個 instances。
MS PowerPoint 只允許執行一個 instances。

- Windows 程式的進入點是WinMain()
- 每個程式應註冊自己的 window class, 其內指定 msg 處理中心 window-procedure
- 一般而言每個程式有一個msg loop.

Win32 Project

W32App.cpp

```
#0041            // 主訊息迴圈:
#0042            while (GetMessage(&msg, NULL, 0, 0))
#0043            {
#0044                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
#0045                {
#0046                   TranslateMessage(&msg);
#0047                   DispatchMessage(&msg);
#0048                }
#0049            }
#0050
#0051            return (int) msg.wParam;
#0052 }
#0053
#0054
#0055
#0056 //
#0057 //  函式: MyRegisterClass()
#0058 //
#0059 //  用途: 登錄視窗類別。
#0060 //
#0061 //  註解:
#0062 //
#0063 //  只有當您希望此程式碼能相容比 Windows 95 的 'RegisterClassEx' 函式更早的 Win32 系統時，才會需要
#0064 // 加入及使用這個函式。您必須呼叫這個函式，讓應用程式取得與它相關的 '正確格式 (Well Formed)' 圖示。
#0065 //
#0066 ATOM MyRegisterClass(HINSTANCE hInstance)
#0067 {
#0068            WNDCLASSEX wcex;
#0069
#0070            wcex.cbSize = sizeof(WNDCLASSEX);
#0071
#0072            wcex.style                  = CS_HREDRAW | CS_VREDRAW;
#0073            wcex.lpfnWndProc            = (WNDPROC)WndProc;
#0074            wcex.cbClsExtra             = 0;
#0075            wcex.cbWndExtra             = 0;
#0076            wcex.hInstance              = hInstance;
#0077            wcex.hIcon                  = LoadIcon(hInstance, (LPCTSTR)IDI_W32APP);
#0078            wcex.hCursor                = LoadCursor(NULL, IDC_ARROW);
#0079            wcex.hbrBackground          = (HBRUSH)(COLOR_WINDOW+1);
#0080            wcex.lpszMenuName           = (LPCTSTR)IDC_W32APP;
```

Win32 Project

W32App.cpp

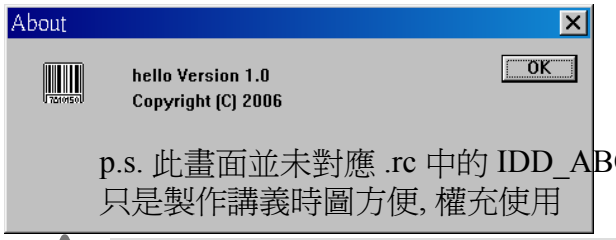
```
#0081              wcex.lpszClassName = szWindowClass;
#0082              wcex.hIconSm   = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
#0083
#0084              return RegisterClassEx(&wcex);
#0085 }
#0086
#0087 //
#0088 //   函式: InitInstance(HANDLE, int)
#0089 //
#0090 //   用途: 儲存執行個體控制代碼並且建立主視窗
#0091 //
#0092 //   註解:
#0093 //
#0094 //        在這個函式中，我們會將執行個體控制代碼儲存在全域變數中，
#0095 //        並且建立和顯示主程式視窗。
#0096 //
#0097 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
#0098 {
#0099    HWND hWnd;
#0100
#0101    hInst = hInstance; // 將執行個體控制代碼儲存在全域變數中
#0102
#0103    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
#0104       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
#0105
#0106    if (!hWnd)
#0107    {
#0108       return FALSE;
#0109    }
#0110
#0111    ShowWindow(hWnd, nCmdShow);
#0112    UpdateWindow(hWnd);
#0113
#0114    return TRUE;
#0115 }
#0116
```

Win32 Project

W32App.cpp

```
#0117 //
#0118 //  函式: WndProc(HWND, unsigned, WORD, LONG)
#0119 //
#0120 //  用途:  處理主視窗的訊息。
#0121 //
#0122 //  WM_COMMAND        - 處理應用程式功能表
#0123 //  WM_PAINT          - 繪製主視窗
#0124 //  WM_DESTROY        - 傳送結束訊息然後返回
#0125 //
#0126 //
#0127 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
#0128 {
#0129             int wmId, wmEvent;
#0130             PAINTSTRUCT ps;
#0131             HDC hdc;
#0132
#0133             switch (message)
#0134             {
#0135             case WM_COMMAND:
#0136                wmId    = LOWORD(wParam);
#0137                wmEvent = HIWORD(wParam);
#0138                // 剖析功能表選取項目:
#0139                switch (wmId)
#0140                {
#0141                case IDM_ABOUT:
#0142                   DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
#0143                   break;
#0144                case IDM_EXIT:
#0145                   DestroyWindow(hWnd);
#0146                   break;
#0147                default:
#0148                   return DefWindowProc(hWnd, message, wParam, lParam);
#0149                }
#0150                break;
```

•WndProc 以switch-case 處理 call back 傳來的各個 msg, 每個 msg 代表一個 event。這種編程形式稱爲 message-based, event-driven.



p.s. 此畫面並未對應 .rc 中的 IDD_ABOUTBOX, 只是製作講義時圖方便, 權充使用

DialogBox(…) 開啓一個對話盒。控制權轉到對話盒函式

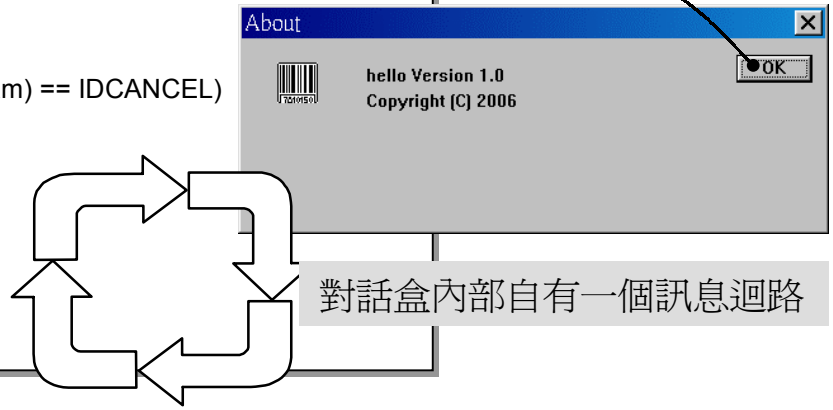
Win32 Project

W32App.cpp

```
#0151            case WM_PAINT:
#0152               hdc = BeginPaint(hWnd, &ps);
#0153               // TODO: 在此加入任何繪圖程式碼...
#0154               EndPaint(hWnd, &ps);
#0155               break;
#0156            case WM_DESTROY:
#0157               PostQuitMessage(0);
#0158               break;
#0159            default:
#0160               return DefWindowProc(hWnd, message, wParam, lParam);
#0161            }
#0162            return 0;
#0163 }
#0164
#0165 // [關於] 方塊的訊息處理常式。
#0166 LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
#0167 {
#0168            switch (message)
#0169            {
#0170            case WM_INITDIALOG:
#0171               return TRUE;
#0172
#0173            case WM_COMMAND:
#0174               if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
#0175               {
#0176                  EndDialog(hDlg, LOWORD(wParam));
#0177                  return TRUE;
#0178               }
#0179               break;
#0180            }
#0181            return FALSE;
#0182 }
```

當 user 結束程式時…
Word 在視窗關閉**前**詢問是否要存檔，
Outlook 在視窗關閉**後**詢問要否寄出送件匣中的信。
它們利用不同的 msg 做這些事情。

[OK] 被按下，產生 WM_COMMAND | IDOK



對話盒內部自有一個訊息迴路

Win32 programming 注意事項

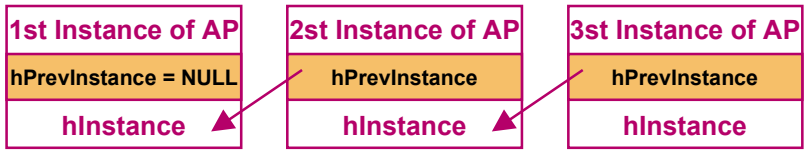
1. int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) /* Windows 3.x (16-bit) definition */
   int APIENTRY WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) /* Windows 32-bit definition */
   (APIENTRY, CALLBACK, WINAPI, PASCAL: ➔ __stdcall // define in "windef.h")

2. Handle & int:
   16-bit --> 32-bit

3. near, far:
   doesn't mean anything

4. Instance & Previous Instance:

Windows 3.X:

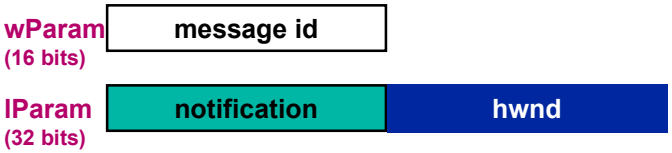


Windows 95/NT:

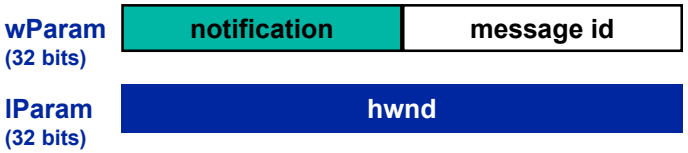
5. LONG FAR PASCAL WndProc(HWND hWnd, WORD message, WORD wParam, LONG lParam) /* A Window Procedure for Windows 3.x */
   LONG APIENTRY WndProc(HWND hWnd, UINT message, UINT wParam, LONG lParam) /* A Window Procedure for Win32 */

6. new Message Handling - wParam & lParam:

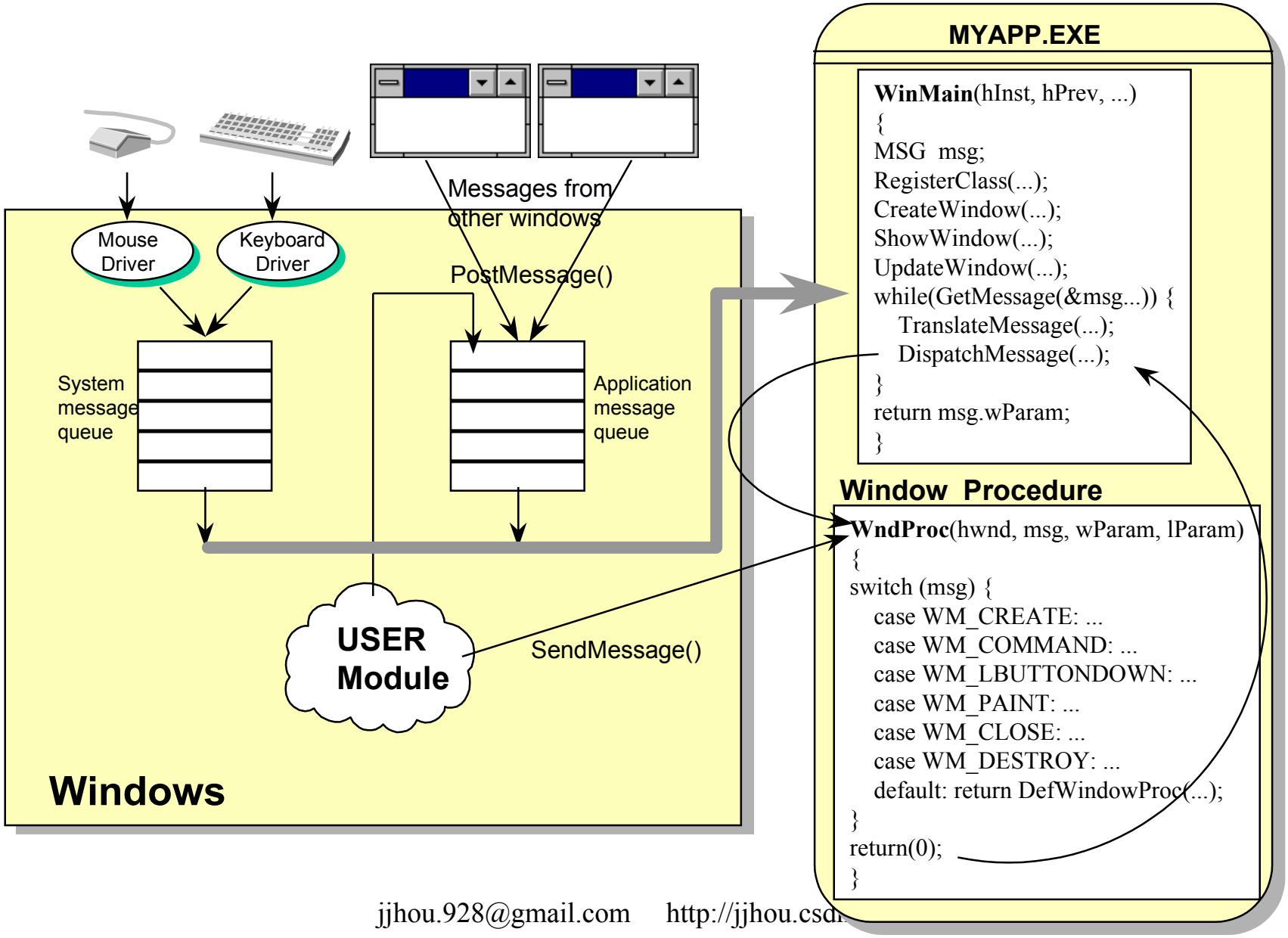
Windows 3.X:



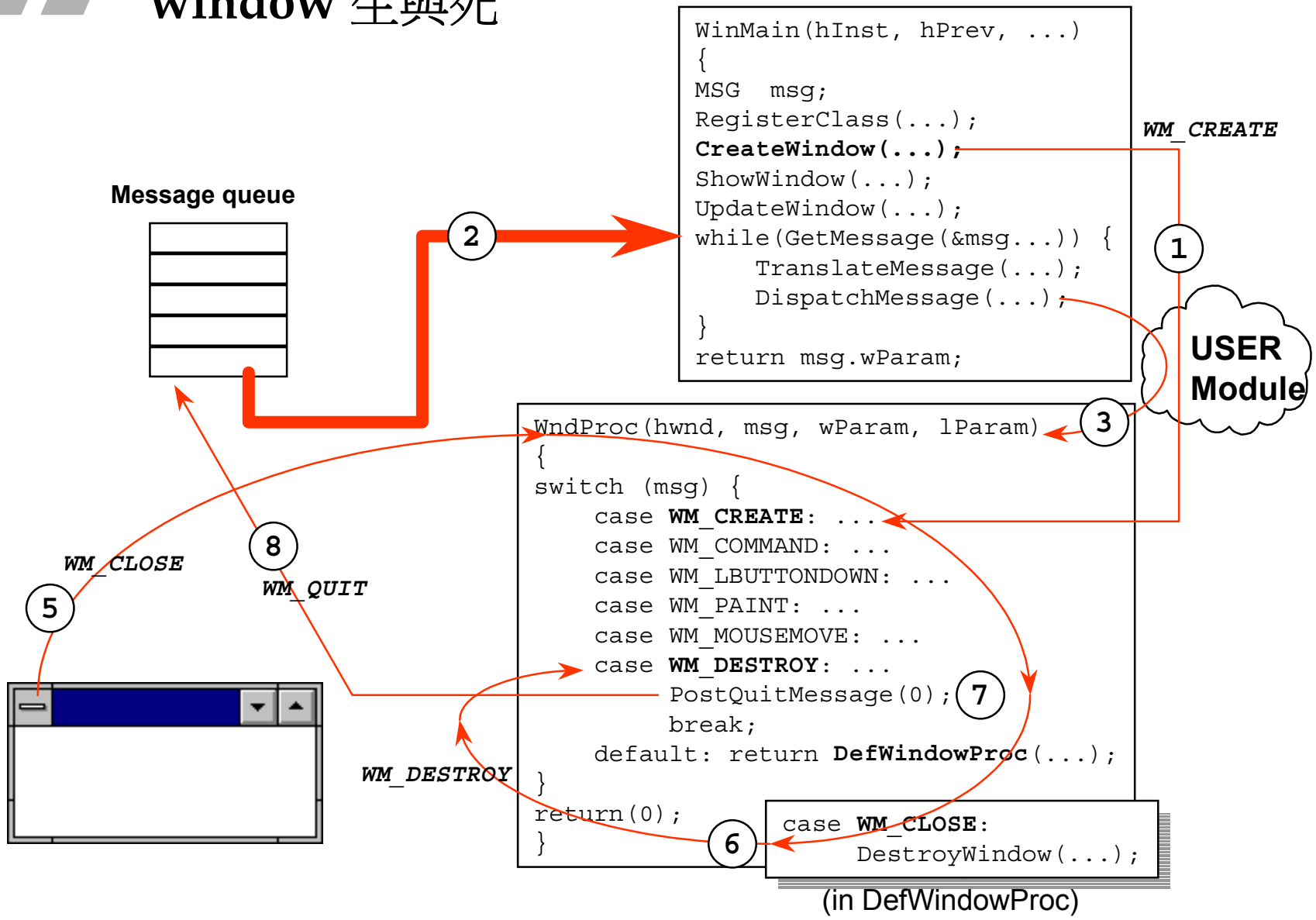
Windows 95/NT:



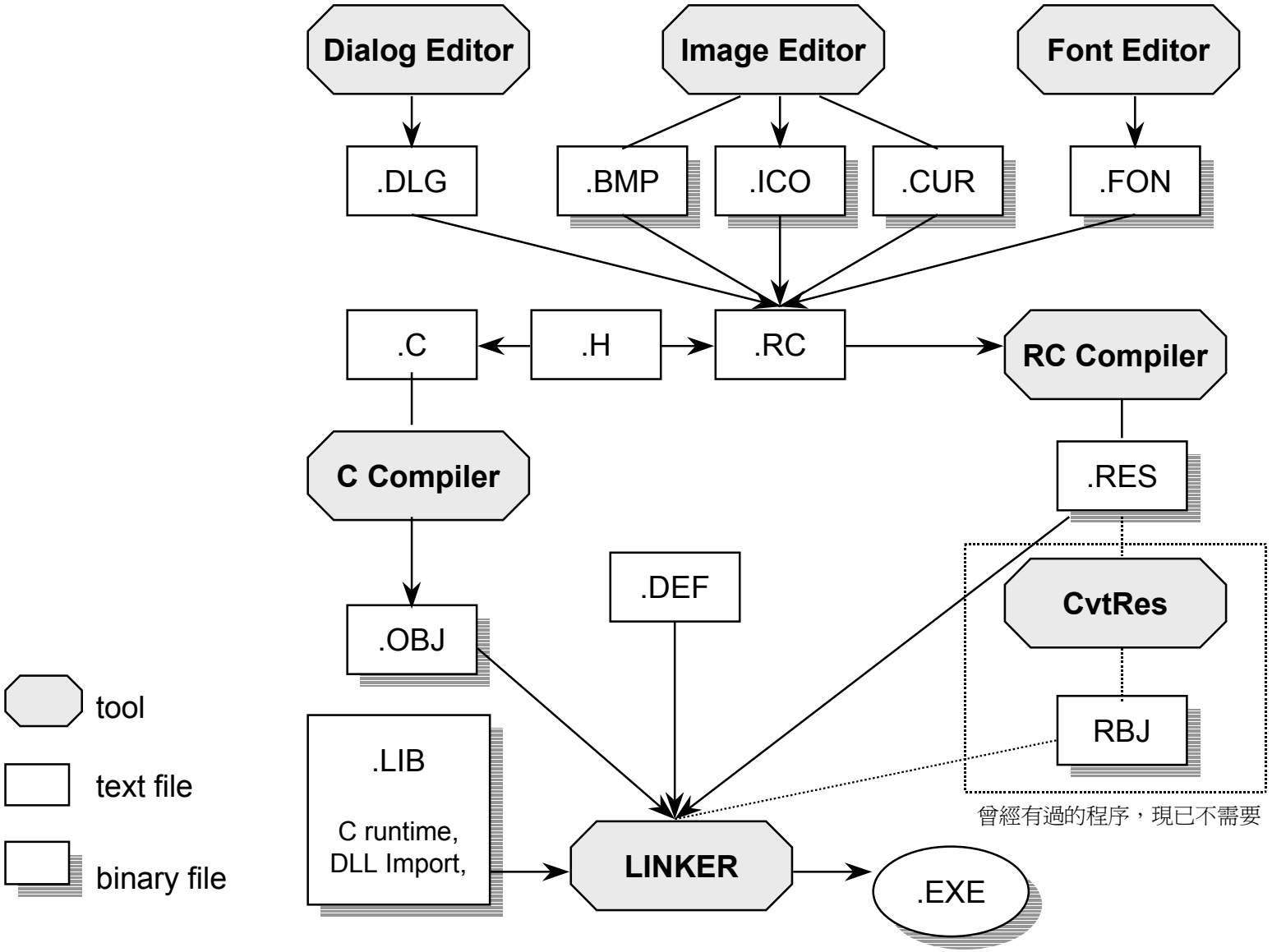
Message Based, Event-Driven



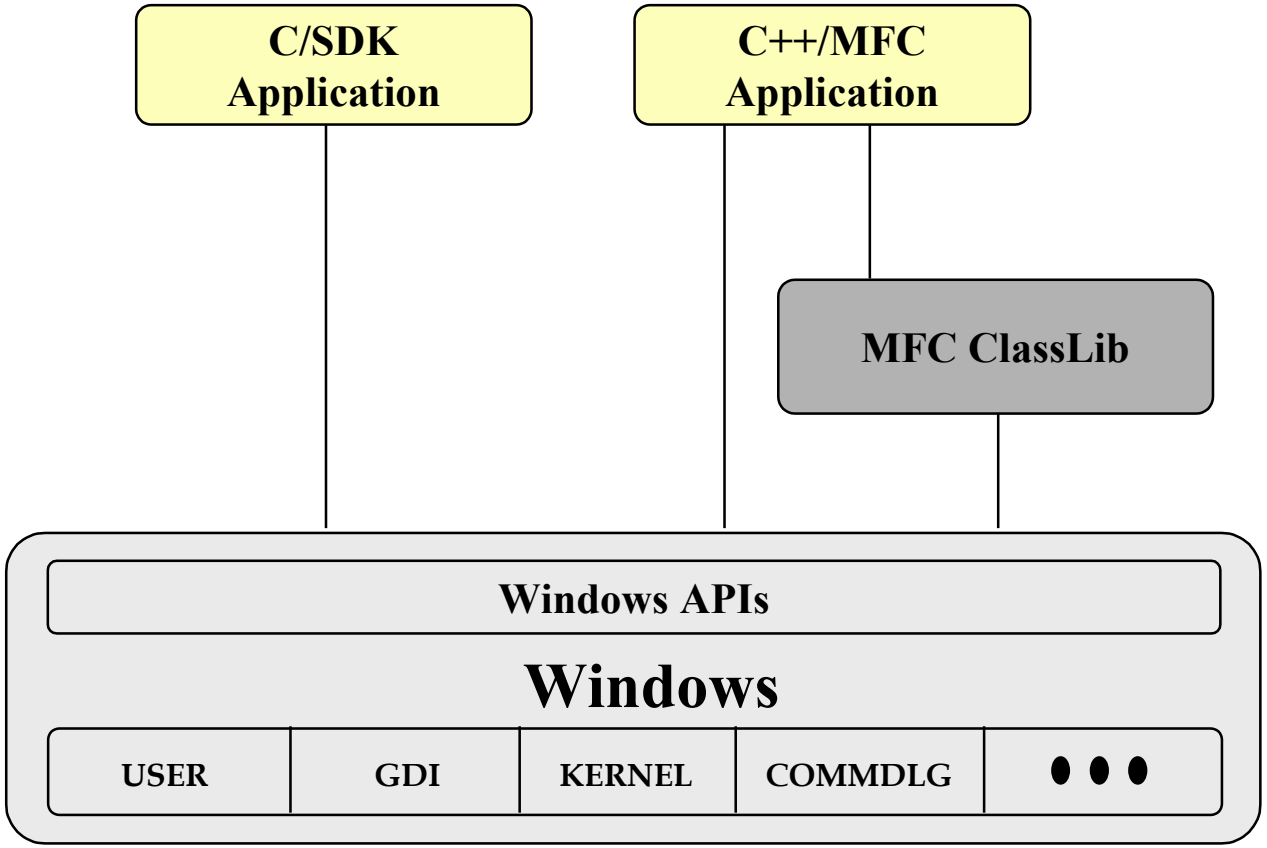
window 生與死


Message queue
2
WinMain(hInst, hPrev, ...)
{
MSG msg;
RegisterClass(...);
CreateWindow(...);
ShowWindow(...);
UpdateWindow(...);
while(GetMessage(&msg...)) {
TranslateMessage(...);
DispatchMessage(...);
}
return msg.wParam;
WM_CREATE
1
USER Module
3
WndProc(hwnd, msg, wParam, lParam)
{
switch (msg) {
case WM_CREATE: ...
case WM_COMMAND: ...
case WM_LBUTTONDOWN: ...
case WM_PAINT: ...
case WM_MOUSEMOVE: ...
case WM_DESTROY: ...
PostQuitMessage(0);
break;
default: return DefWindowProc(...);
}
return(0);
}
7
8
WM_QUIT
WM_CLOSE
5
WM_DESTROY
6
case WM_CLOSE:
DestroyWindow(...);
(in DefWindowProc)

Build a Win32 App.



C/SDK vs. C++/MFC


C/SDK Application
C++/MFC Application
MFC ClassLib
Windows APIs
Windows
USER
GDI
KERNEL
COMMDLG

何謂 Unicode ?

ANSI C run time library 處理本土化時所面臨的問題：

```
len = strlen("this is a testing string.");          // return value is 25
len = strlen("這是一段測試字串(string)。");          // return value is 26 (而不是 17)
```

雙位元組字元集 (double-byte character sets: DBCSs)：

字串中每一字元均由一或二個位元組(byte)所構成,
一般如果第一個位元組的值大於 0x80, 那麼這個字元就是雙位元組字元

```
#include <mbstring.h>                               // msvc++ run time library

len = _mbslen("這是一段測試字串(string)。");         // return value is 17
```

寬字元組字元集 (Unicode)：

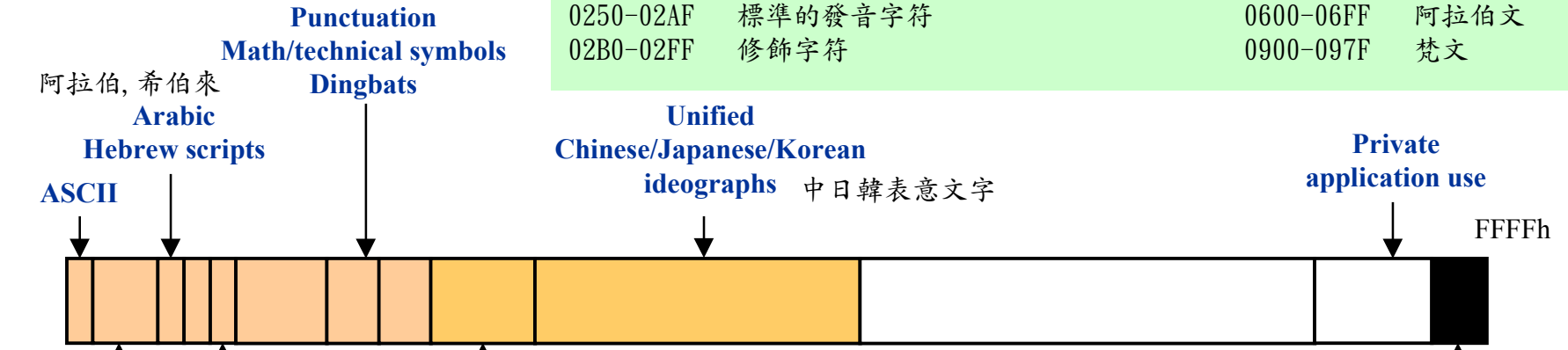
由 Apple & Xerox 1988 年建立的標準(成員包括: IBM, HP, Microsoft, Oracle, Sybase, ...)
字串中每一字元均由二個位元組(byte)所構成, 共有 65,536 個字元可供使用
(已定義字碼約為 35,000 個, 剩餘 29,000 保留未來擴充, 6,000 保留給私人定義使用)

使用 Unicode 的好處：

1. 允許同一份文件可同時展現不同語文
2. 不同語文之資料轉換更容易
3. 可以用同一份程式碼來處理所有語文
4. 增進視窗環境下應用程式之執行效率
5. 極有可能成為未來的資料儲存趨勢

Unicode layout

0000-007F	ASCII	0300-036F	讀音標記
0080-00FF	Latin（包含西歐國家字母）	0400-04FF	斯拉夫文
0100-017F	Latin extended-A	0530-058F	亞美尼亞文
0180-024F	Latin extended-B	0590-05FF	希伯來文
0250-02AF	標準的發音字符	0600-06FF	阿拉伯文
02B0-02FF	修飾字符	0900-097F	梵文



Future use

Provides compatibility

Reference: *The Unicode Standard: Worldwide Character Encoding, V 1.0*

OS 與 Unicode 的關係 - 影響程式效率

Windows NT/2000/XP 系列之作業系統係以 Unicode 來建構，
所有用來建立視窗、顯示文字、字串處理等核心程式碼都要求使用 Unicode，
如果以 ANSI 字串傳入那些函式，系統執行時就需做以下事情(系統自動)：

1. 配置記憶體
2. ANSI 字串轉換為 Unicode
3. 執行該函式要做的事

結果：執行效率變差

merging operator

```
#ifdef _UNICODE
#define __T(x)      L ## x
#else  /* _UNICODE */
#define __T(x)      x
```

```
eg.         DWORD    dwTickCount = GetTickCount();
            for (int i=0 ; i<0xFFFFF ; i++)        SetEnvironmentVariable(_T("=C:"), _T("VALUE1"));
            _tprintf(_T("total time is: %d\n"), GetTickCount()-dwTickCount);
```

```
Pentium III 500 -->  ANSI version          33,678 ms
                     Unicode version       16,393 ms
```

Windows 98/Me 系列之作業系統是以舊有的 16 位元作業系統來建構，
幾乎所有內部的處理，都以 ANSI 字串來處理，
如果以 Unicode 字串呼叫該函式，必須先做以下事情(程式員自己寫)：

1. 配置記憶體
2. Unicode 字串轉換為 ANSI
3. 呼叫該函式

ANSI C 對 Unicode 的支援

in <string.h>

```
typedef     unsigned short     wchar_t
```

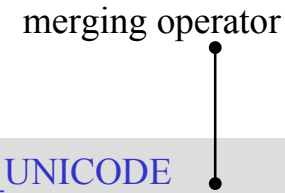
```
char*       strcat(char*, const char*)
wchar_t*    wcscat(wchar_t *, const wchar_t *);

char*       strchr(const char*, int);
wchar_t*    wcschr(const wchar_t*, wchar_t);

int         strcmp(const char*, const char*);
int         wcscmp(const wchar_t*, const wchar_t *);

int         _stricmp(const char*, const char*);
int         _wcsicmp(const wchar_t*, const wchar_t*);

char*       strcpy(char*, const char*);
wchar_t*    wcscpy(wchar_t*, const wchar_t*);

size_t      strlen(const char*);
size_t      wcslen(const wchar_t*);
:           :
char*       pszStr = “error sting”;
wchar_t *   pszStr = L“error sting”;
```

困擾：
若直接呼叫 str 或 wcs 函式，將無法在不改動程式碼的情況下編譯出 ANSI 或 Unicode 版本

利用 TCHAR.H 的輔助定義達成目的

ANSI C header TCHAR.H 的 ANSI/Unicode 一般化巨集定義

TCHAR.H macros	if _UNICODE defined	if _UNICODE undefined
_tprintf	wprintf	printf
_ftprintf	fwprintf	fprintf
_stprintf	swprntf	sprintf
_sntprintf	_snwprintf	_snprintf
_tscanf	wscanf	scanf
_fscanf	fwcsanf	fscanf
_stscanf	swscanf	sscanf
_fgettc	fgetwc	fgetc
_fgettchar	fgetwchar	fgetchar
_fcscat	wcscat	strcat
_tcscmp	wcscmp	strcmp
_tcscpy	wcscpy	strcpy
_tcslen	wcslen	strlen
_tcsstr	wcsstr	strstr
_tcstok	wcstok	strtok
:	:	:
_T("test string")	L"test string"	"test string"
_T('c')	L'c'	'c'




```
#ifdef _UNICODE
#define __T(x)      L ## x
#else  /* _UNICODE */
#define __T(x)      x
#endif  /* _UNICODE */

#define _T(x)       __T(x)
#define _TEXT(x)    __T(x)
```

ANSI/Unicode 之一般化巨集定義

macros	if **UNICODE** defined	if **UNICODE** undefined
PTSTR	PWSTR	PSTR
PCTSTR	PCWSTR	PCSTR

```
#ifdef    UNICODE
#define   CreateDirectory      CreateDirectoryW
#else
#define   CreateDirectory      CreateDirectoryA
#endif
```

Windows NT/2000

```
CreateDirectoryA(...)
{
    allocate memory
    ANSI --> Unicode
    CreateDirectoryW(…)
    free memory
}

CreateDirectoryW(...)
{
    實際運作碼
}
```

Windows 98/Me

```
CreateDirectoryA(...)
{
    實際運作碼
}

CreateDirectoryW(...)
{
    傳回失敗值
}
```

撰寫兼容 ANSI 及 Unicode 的程式碼

- 視字串為 array of characters，不是 array of bytes
- 以 TCHAR 及 PTSTR 表示字元與字串，不使用 char 和 PSTR
- 不使用 char, char*, PSTR 等意義不明確的資料型態
- 使用 macro _T 宣告字元和字串
- 修改字串長度之計算方式

以字元為單位之緩衝區大小應為：
sizeof(szBuffer)/sizeof(TCHAR)
而不是： sizeof(szBuffer)

配置字元緩衝區的記憶體時應為： malloc(nCount*sizeof(TCHAR))
而不是： malloc(nCount)

函式：ANSI 版 vs. Unicode 版

假設你需要寫一個字串反轉函式

```
BOOL    StringReverseW(LPWSTR lpWideCharStr);
BOOL    StringReverseA(LPSTR lpMultiByteStr);

#ifdef _UNICODE
#define   StringReverse  StringReverseW
#else
#define   StringReverse  StringReverseA
#endif
```

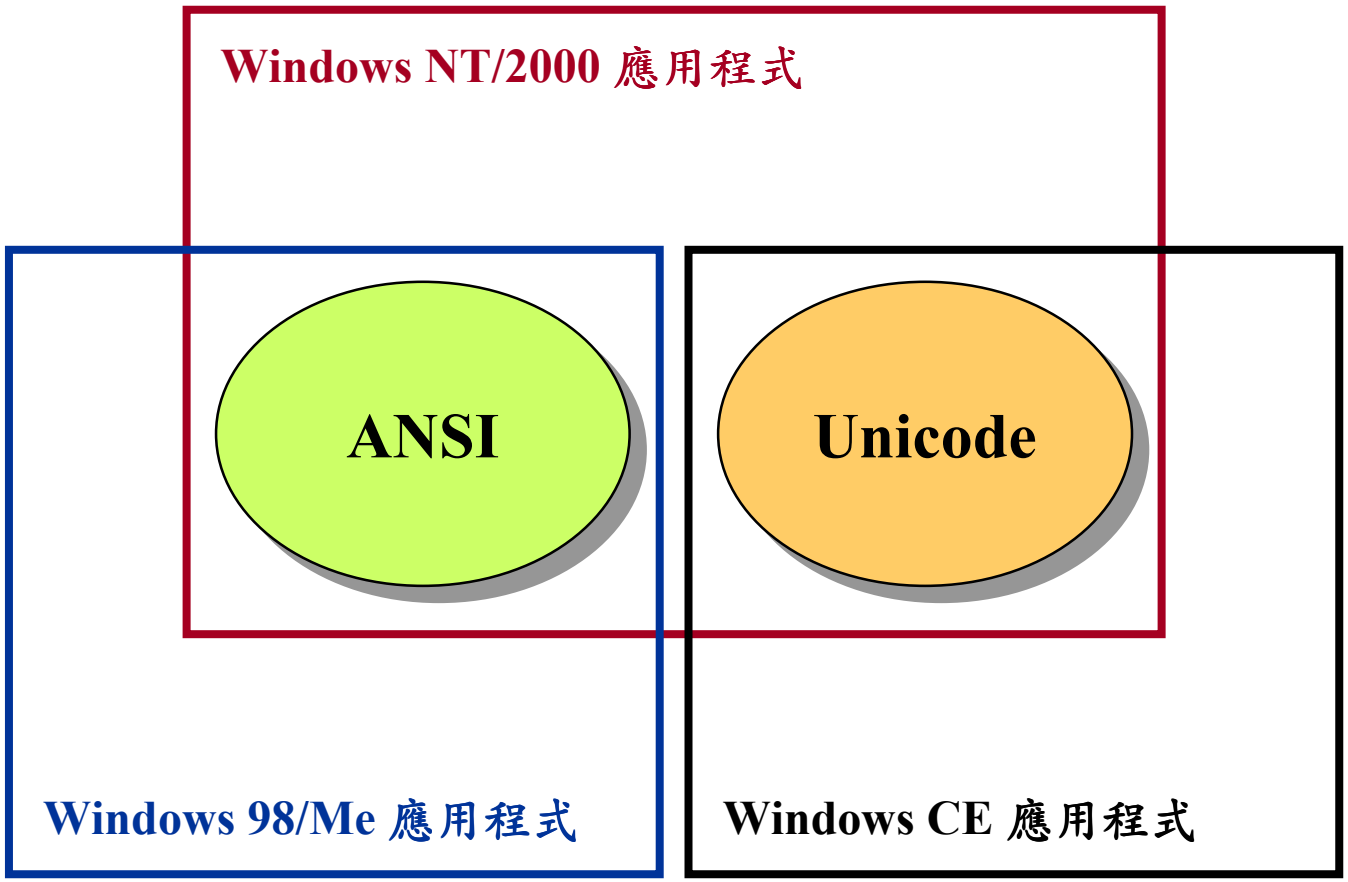
Unicode version

```
BOOL    StringReverseW (LPWSTR   lpWideCharStr)
{
    // get last char pointer
    LPWSTR   lpEndOfStr = lpWideCharStr +
                          wcslen(lpWideCharSet) - 1;
    wchar_t     cCharT;

    while  (lpWideCharStr < lpEndOfStr) {
        // exchange the last char and the first
        cCharT = *lpWideCharStr;
        *lpWideCharStr = *lpEndOfStr;
        *lpEndOfStr = cCharT;

        lpWideCharStr++;
        lpEndOfStr--;
    }
    return TRUE;
}
```

ANSI version

```
BOOL   StringReverseA (LPSTR lpMultiByteStr)
{
    LPWSTR   lpWideCharStr;
    int         nLenOfWideCharStr;
    BOOL      fOK = FALSE;

    // convert ANSI string to Unicode string
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
                                    lpMultiByteStr, -1, NULL, 0);
    lpWideCharStr = HeapAlloc(GetProcessHeap(), 0,
                                    nLenOfWideCharStr);
    MultibyteToWideChar(CP_ACP, 0, lpMultiByteStr, -1,
                                lpWideCharStr, nLenOfWideCharStr);

    // call Unicode version
    fOK = StringReverseW(lpWideCharStr);

    // convert Unicode string to ANSI string
    if (bOK) {
        WideCharToMultiByte(CP_ACP, 0, lpWideCharStr, -1,
                lpMultiByteStr, strlen(lpMultiByteStr), NULL, NULL);
    }
    HeapFree(GetProcessHeap(), 0, lpWideCharStr);

    return fOK;
}
```

Unicode 相關的應用軟體開發



Unicode


个人银行专业版
疑难解答| 留言| 社区| 帮助| 重登录| 退出
一卡通
信用卡
存 折
财务管理
专业版管理
在线客服
功能导航
我有问题
收款方信息查询
填写收款向导?
省/直辖市
市
检索行号
检索不到行号，可直接输入开户行名。 什么是行号?
元 大写
提示：本功能只向其他银行账户汇款，向招行帐户汇款请点击 这里
自动保存收款方信息[R]:
收款方短信通知[R]:
收款方手机号码[O]:
取款密码[P]:
校验码[V]:
2555
请按显示输入
确 定
取 消
说明：
1、请在办理转帐汇款前参阅到帐时间及手续费标准，并据此选择付款方式。
2、向招行信用卡帐户转帐，请选招行同城转账
3、如果您尚未开通异地汇款功能，请点击异地汇款功能申请
4、如果系统报告"通讯故障"，请立即查询交易查询确认交易是否成功。

Unicode


一网通
ALL IN ONE NET
个人银行专业版
疑难解答| 留言| 社区| 帮助| 重登录| 退出
一卡通
信用卡
存 折
财务管理
专业版管理
在线客服
功能导航
当前功能|招行异地汇款
付款帐户[T]:
收款方信息查询
收款方户名[N]:
填写收款向导?
收款方地址[A]:
省/直辖市
市
收款方开户行[B]:
汇入收款方帐号[C]:
重复输入收款方帐号[D]:
汇款金额[E]:
元 大写
壹拾伍万元整
汇款用途[M]:
采用招行系统内快速汇款[Q]:
(收款方账户必须为招商银行"一卡通"才有效)
自动保存收款方信息[R]:
收款方短信通知[R]:
收款方手机号码[Q]:
取款密码[P]:
校验码[V]:
4217
请按显示输入
确 定
取 消
说明：
1、请在办理转帐汇款前参阅到帐时间及手续费标准，并慎重选择付款方式。
2、向招行信用卡帐户转帐，请选招行同城转账
3、如果您尚未开通异地汇款功能，请点击异地汇款功能申请
4、如果系统报告"通讯故障"，请立即查询交易查询确认交易是否成功。

Win32 Heap APIs

此外還有 **GetProcessHeap()**，可見 process 預設就有 heap(s).

The **HeapCreate** function creates a **heap object** that can be used by the calling process. The function reserves space in the virtual address space of the process and **allocates physical storage** for a specified initial portion of this block.

```
HANDLE HeapCreate(
  DWORD flOptions,      // heap allocation flag
  DWORD dwInitialSize,  // initial heap size
  DWORD dwMaximumSize   // maximum heap size
);
```

建立一個 heap object 可被 calling process 使用

The **HeapAlloc** function allocates a block of memory from a heap. The allocated memory is not movable.

```
LPVOID HeapAlloc(
  HANDLE hHeap,  // handle to the private heap block
  DWORD dwFlags, // heap allocation control flags
  DWORD dwBytes  // number of bytes to allocate
);
```

從 heap 配置一塊 memory。此塊 memory 不可搬移，也因此有可能使 heap 變成破碎.
所得區塊 >= 申請尺寸。如果所得大於申請，整個可被 calling process 用。決定所得尺寸可用 HeapSize()

If HeapAlloc succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the allocated block, use the **HeapSize** function.
To free a block of memory allocated by HeapAlloc, use the **HeapFree** function.
Memory allocated by HeapAlloc is not movable. Since the memory is not movable, it is possible for the heap to become fragmented.

Win32 Heap APIs

The **HeapFree** function frees a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

釋放從 heap 配置得來的一塊 memory。

```
BOOL HeapFree(
  HANDLE hHeap,  // handle to the heap
  DWORD dwFlags, // heap freeing flags
  LPVOID lpMem   // pointer to the memory to free
);
```

The **HeapDestroy** function destroys the specified heap object. HeapDestroy decommits and releases all the pages of a private heap object, and it invalidates the handle to the heap.

```
BOOL HeapDestroy(
  HANDLE hHeap   // handle to the heap
);
```

The **HeapSize** function returns the size, in bytes, of a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

```
DWORD HeapSize(
  HANDLE hHeap, // handle to the heap
  DWORD dwFlags, // heap size control flags
  LPCVOID lpMem // pointer to memory to return size for
);
```

Win32 Heap APIs, Examples (from VC malloc source)

```
int __cdecl _heap_init (int mtflag)
{
      //  Initialize the "big-block" heap first.
      if ( (_crtheap = HeapCreate( mtflag ? 0 : HEAP_NO_SERIALIZE,
                                   BYTES_PER_PAGE, 0 )) == NULL )
         return 0;
      //  Initialize the small-block heap
      if (__sbh_heap_init() == 0)   {  HeapDestroy(_crtheap);   return 0;  }
      return 1;
}
```

```
int __cdecl __sbh_heap_init (void)
{
   if (!(__sbh_pHeaderList =
         HeapAlloc(_crtheap, 0, 16 * sizeof(HEADER))))
      return FALSE;
   …
   return TRUE;
}
```

```
void * __cdecl _heap_alloc_base (size_t size)
{
      void * pvReturn;
      if (size <= __sbh_threshold)    //3F8
      {  …    return pvReturn;      }
      if (size == 0)  size = 1;
      size = …;
      return HeapAlloc(_crtheap, 0, size);
}
```

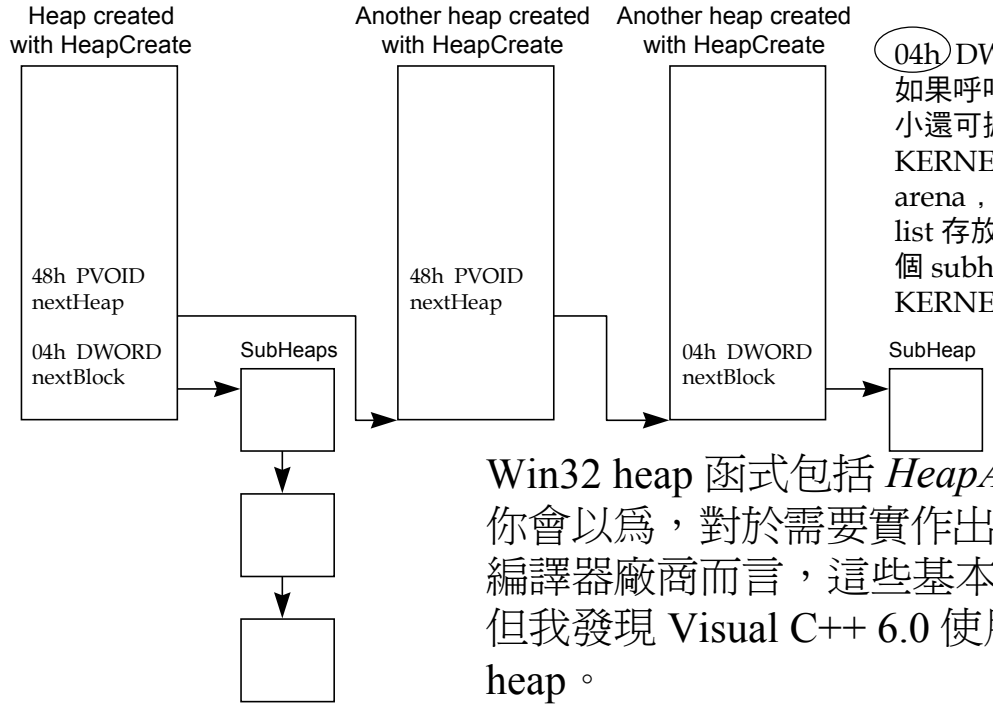
```
#01 PHEADER __cdecl __sbh_alloc_new_region (void)
#02 {
…
#21    //  allocate a new region associated with the new header
#22    if (!(pHeader->pRegion = (PREGION)HeapAlloc(_crtheap, HEAP_ZERO_MEMORY,
#23                                                sizeof(REGION))))
#24       return NULL;
```

Win32 Heap

Win32 作業系統有一些高級 heap 管理函式。在 Windows 95 中每一個區塊只需額外消耗 4 bytes，且理論上可以產生 heap 高達 2GB-4MB。此外 Windows 95 的 Win32 heaps 爲各種大小的區塊維護了四個分離的 free lists，爲的是避免記憶體過度支離破碎。

Windows 95 允許行程擁有一個以上的 heaps。因此總是必須傳一個 heap handle 給任何 Win32 heap 函式，用來表示操作對象。Heap handle 其實就是 heap 的起始點線性位址。

Windows 95 heaps 的另一個好性質是，它們可以成長。這種情況下 KERNEL32 配置額外的記憶體並將之與 heap 產生關聯。此額外記憶體被稱爲 subheaps。



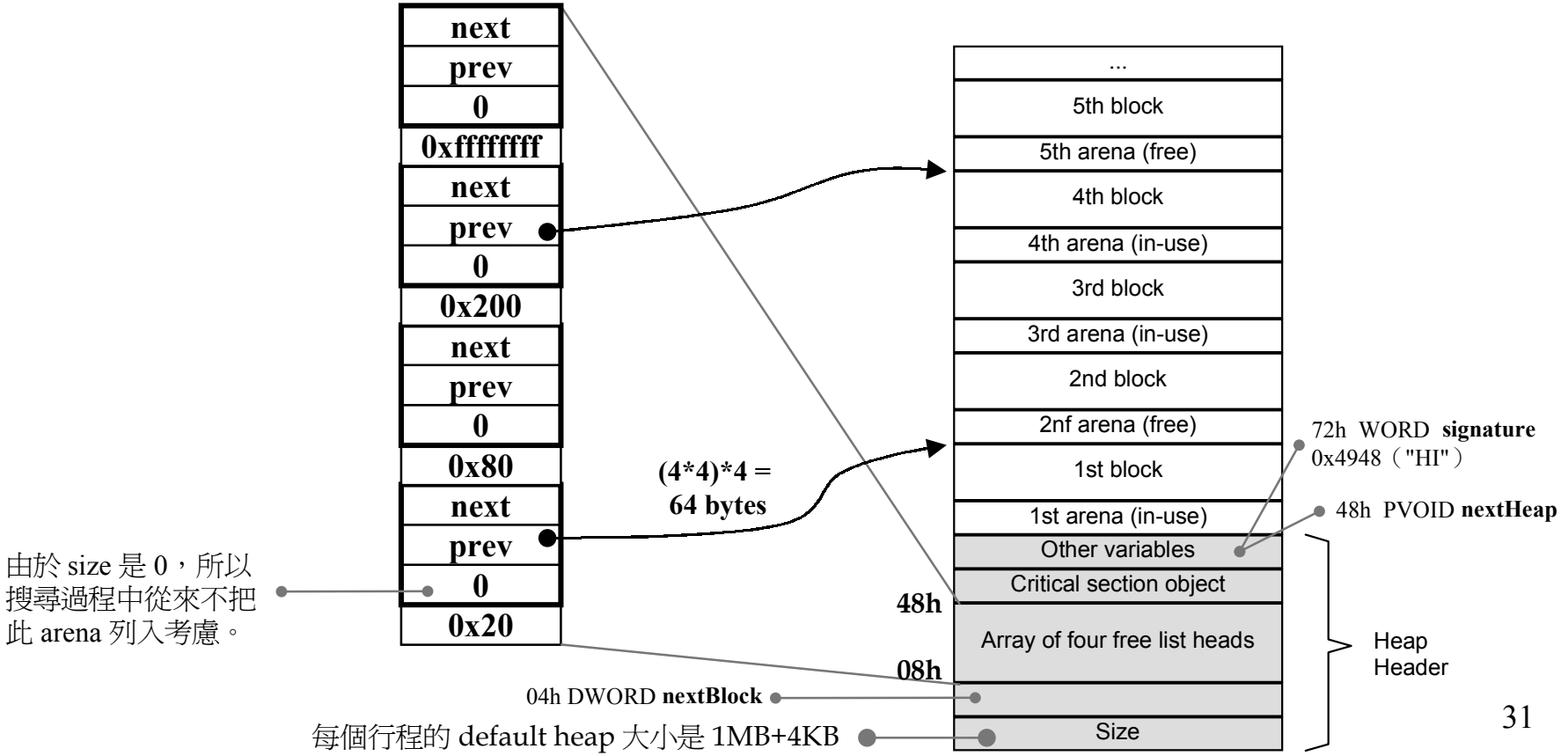
04h DWORD **nextBlock**
如果呼叫 *HeapCreate* 時將 dwMaximum 參數指定? 0，heap ?生之後其大小還可擴充。這種情況下，當呼叫端要求的區塊對此 heap 而言太大，KERNEL32 就會取其他記憶體設? subheaps。Subheaps 也使用 heap arena，但不使用整個表頭結構。?了追?這些 subheaps，KERNEL32 以 list 存放?們，而其 list head 就記錄在主表頭結構的此一欄位。指向下一個 subheap 的指標則放在每個 subheap 的 04 偏移處。一旦 heap 被??，KERNEL32 會走訪 list 中的每個 subheaps，將其 pages 釋放給系統。

Win32 heap 函式包括 *HeapAlloc*、*HeapFree*、*HeapReAlloc* 等等。或許你會以爲，對於需要實作出 *malloc*、*realloc*、*free*、*new*、*delete* 函式的編譯器廠商而言，這些基本的 heap 函式是必然的選擇。不，不一定。但我發現 Visual C++ 6.0 使用 Win32 heap 函式來完成其 C/C++ runtime heap。

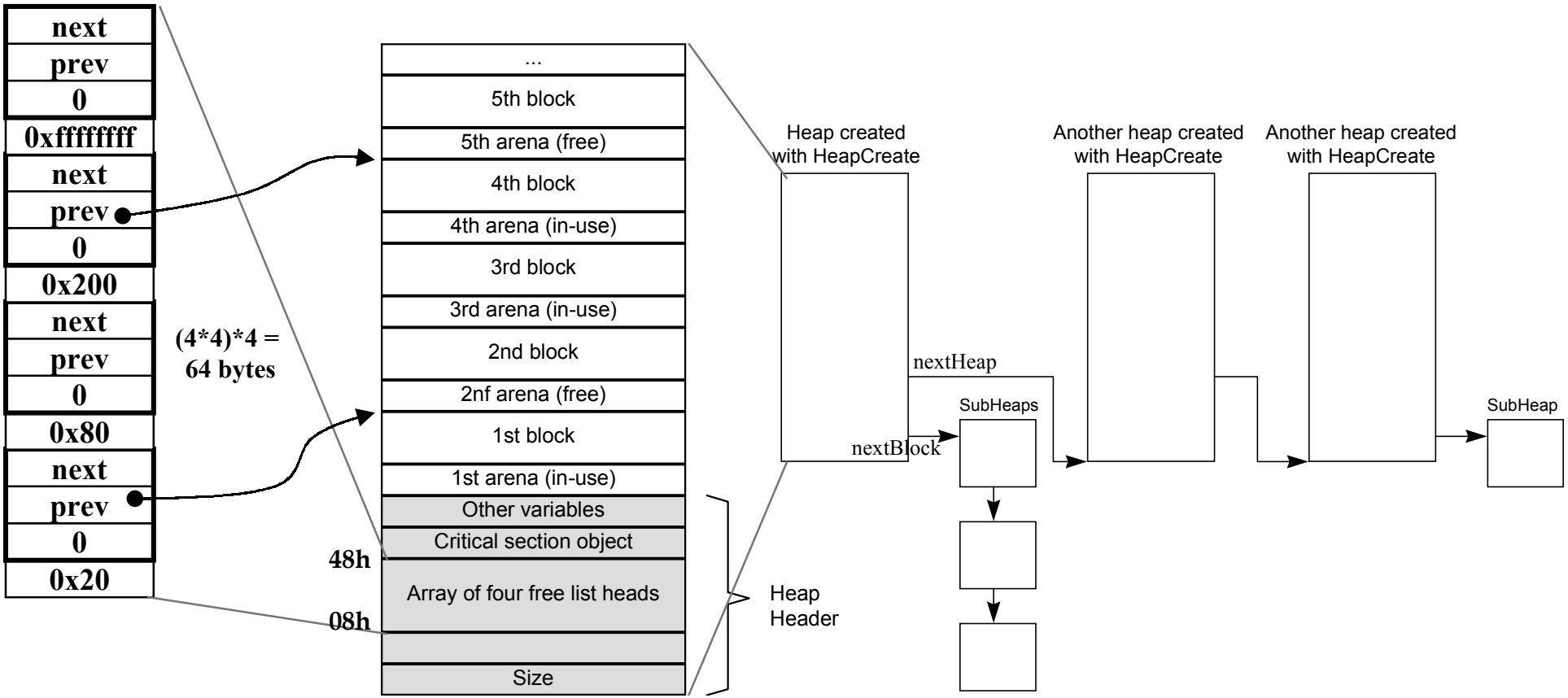
Heap header & Heap arena

Win32 heap 函式其實是 Win32 virtual 函式的上層。
Windows 95 heap 的所有組件都是從 VMM *_PageReserve* 配置的記憶體中產生出來。此塊記憶體被區分爲兩塊。起頭是個 heap 表頭，內含 heap 管理資訊，如 free lists、heap size、heap creation flag 等等。Heap 表頭之下便是 heap 區塊，一開始是個 arena 結構，內含區塊資訊。區塊的起頭緊跟著前一區塊的末尾。所有區塊延伸至 heap 空間尾端，但不是其中每個 page 都一定映射至實際 RAM。

所謂 heap handle，例如以 *GetProcessHeap* 獲得者，就是個 pointer to heap header。*HeapCreate* 的主要工作，除了保留記憶體給 heap 使用，就是將 heap header 初始化。緊跟 header 之後的就是第一個 heap 區塊的 arena。



Heap header & Heap arena



Win32 Virtual Allocate APIs

The **VirtualAlloc** function reserves or commits a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, unless the MEM_RESET flag is set.

```
LPVOID VirtualAlloc(
  LPVOID lpAddress, // address of region to reserve or commit
  DWORD dwSize,     // size of region
  DWORD flAllocationType,    // type of allocation
  DWORD flProtect   // type of access protection
);
```

If the function succeeds, the return value is the base address of the allocated region of pages.
If the function fails, the return value is NULL. To get extended error information

The **VirtualFree** function releases or decommits (or both) a region of pages within the virtual address space of the calling process.

```
BOOL VirtualFree(
  LPVOID lpAddress,  // address of region of committed pages
  DWORD dwSize,      // size of region
  DWORD dwFreeType   // type of free operation
);
```

Win32 Virtual Allocate APIs

•**MEM_COMMIT** Allocates physical storage in memory or in the paging file on disk for the specified region of pages. An attempt to commit an already committed page will not cause the function to fail. This means that a range of committed or decommitted pages can be committed without having to worry about a failure.

•**MEM_RESERVE** Reserves a range of the process's virtual address space without allocating any physical storage. The reserved range cannot be used by any other allocation operations (the malloc function, the LocalAlloc function, and so on) until it is released. Reserved pages can be committed in subsequent calls to the VirtualAlloc function.

•**MEM_RESET** Windows NT: Specifies that memory pages within the range specified by lpAddress and dwSize will not be written to or read from the paging file.

- •When you set the MEM_RESET flag, you are declaring that the contents of that range are no longer important. The range is going to be overwritten, and the application does not want the memory to migrate out to or in from the paging file.
- •Setting this flag does not guarantee that the range operated on with MEM_RESET will contain zeroes. If you want the range to contain zeroes, decommit the memory and then recommit it.
- •When you set the MEM_RESET flag, the VirtualAlloc function ignores the value of fProtect. However, you must still set fProtect to a valid protection value, such as PAGE_NOACCESS.
- •VirtualAlloc returns an error if you set the MEM_RESET flag and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

• **MEM_TOP_DOWN** Allocates memory at the highest possible address.

•**MEM_DECOMMIT** Decommits the specified region of committed pages.
An attempt to decommit an uncommitted page will not cause the function to fail. This means that a range of committed or uncommitted pages can be decommitted without having to worry about a failure.

•**MEM_RELEASE** Releases the specified region of reserved pages. If this flag is specified, the dwSize parameter must be zero, or the function fails.

Win32 Virtual Allocate APIs, Examples

```
pHeader->pHeapData = VirtualAlloc(0, BYTES_PER_REGION, MEM_RESERVE,...)
```

```
#01 int __cdecl __sbh_alloc_new_group (PHEADER pHeader)
#02 {
…
#34   pHeapStartPage = (void *)((char *)pHeader->pHeapData +
#35                              indCommit * BYTES_PER_GROUP);
#36   if ((VirtualAlloc(pHeapStartPage, BYTES_PER_GROUP, MEM_COMMIT,
#37                                                 PAGE_READWRITE)) == NULL)
#38       return -1;
...
```

```
#001 void __cdecl __sbh_free_block (PHEADER pHeader, void * pvAlloc)
#002 {
…
#185       VirtualFree(pHeapDecommit, BYTES_PER_GROUP, MEM_DECOMMIT);
…
#201         //  release the address space for heap data
#202         VirtualFree(__sbh_pHeaderDefer->pHeapData, 0, MEM_RELEASE);
#203
#204         //  free the region memory area
#205         HeapFree(_crtheap, 0, __sbh_pHeaderDefer->pRegion);
```

Win32 Global/Local Allocate APIs

GlobalAlloc

The GlobalAlloc function allocates the specified number of bytes **from the heap**. Win32 memory management does not provide a separate local heap and global heap.

This function is provided only for compatibility with 16-bit versions of Windows.

```
HGLOBAL GlobalAlloc(
  UINT uFlags,        // allocation attributes
  DWORD dwBytes       // number of bytes to allocate
);
```

但 Win32 process 可擁有多個 (local) heaps 不是嗎？以 HeapCreate() 就可以創建出。

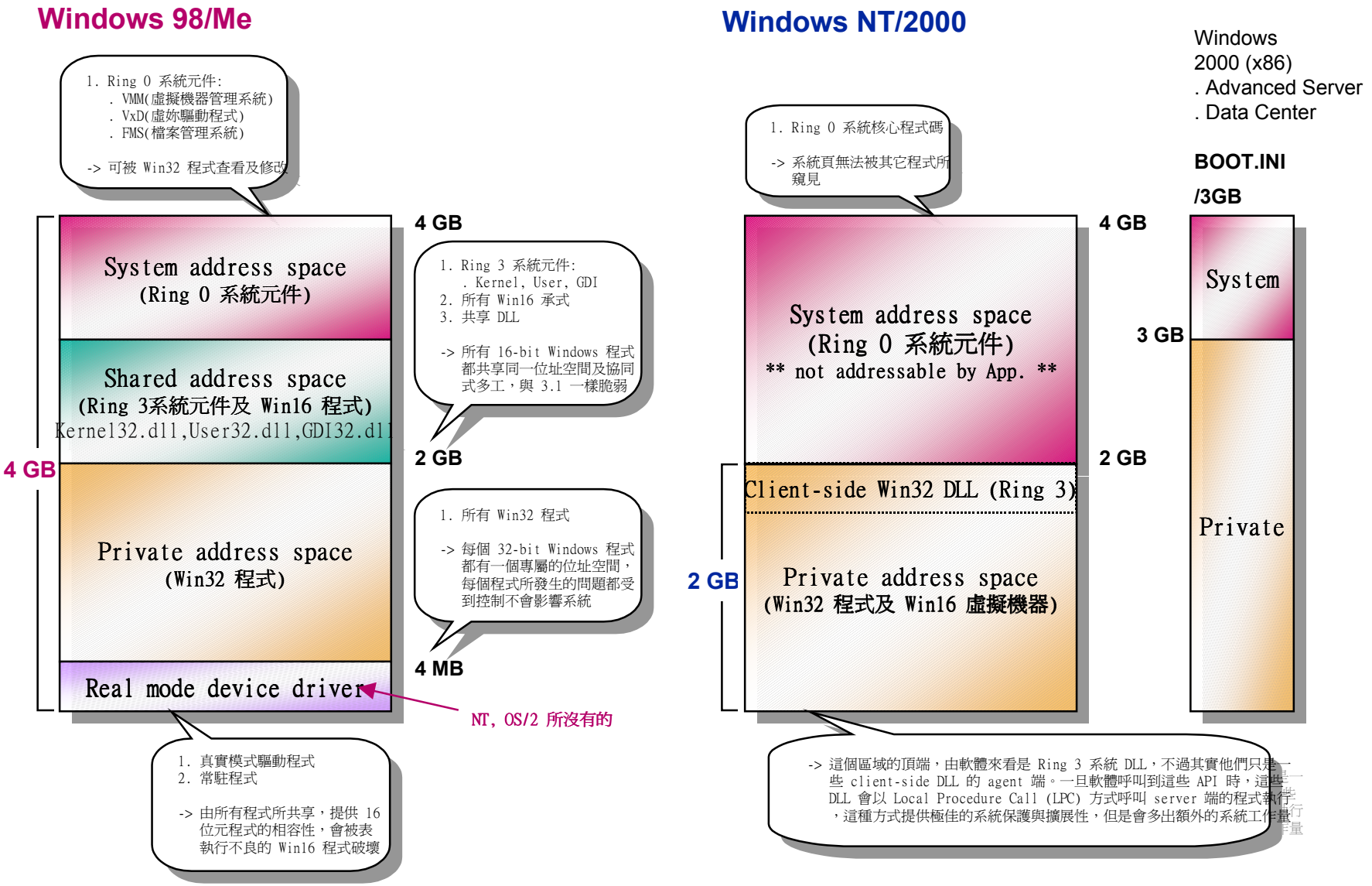
LocalAlloc

The LocalAlloc function allocates the specified number of bytes **from the heap**. Win32 memory management does not provide a separate local heap and global heap.

This function is provided only for compatibility with 16-bit versions of Windows.

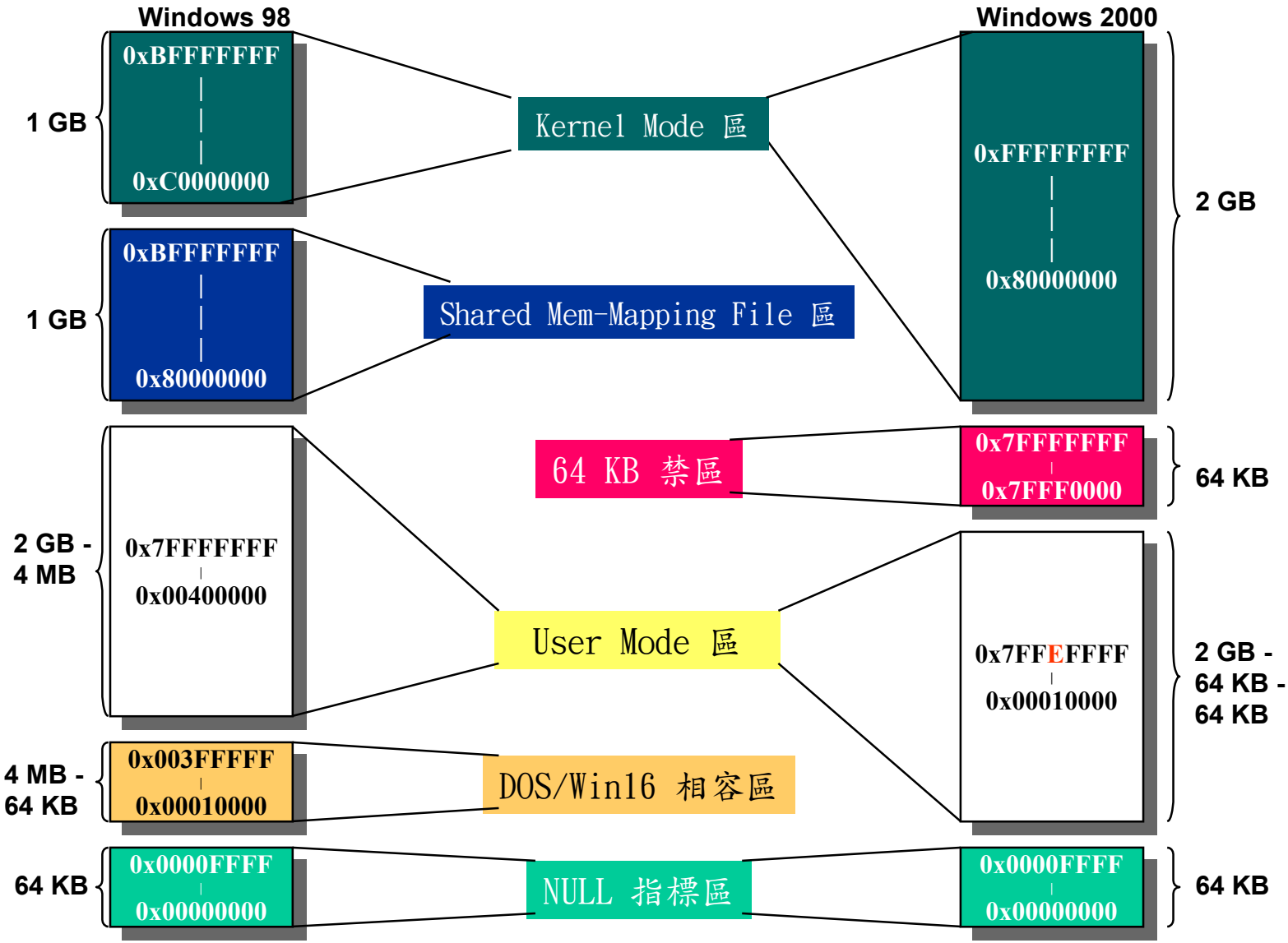
```
HLOCAL LocalAlloc(
  UINT uFlags,        // allocation attributes
  UINT uBytes         // number of bytes to allocate
);
```

Windows Address Space



Windows Address Space

行程之虛擬位址空間分割方式



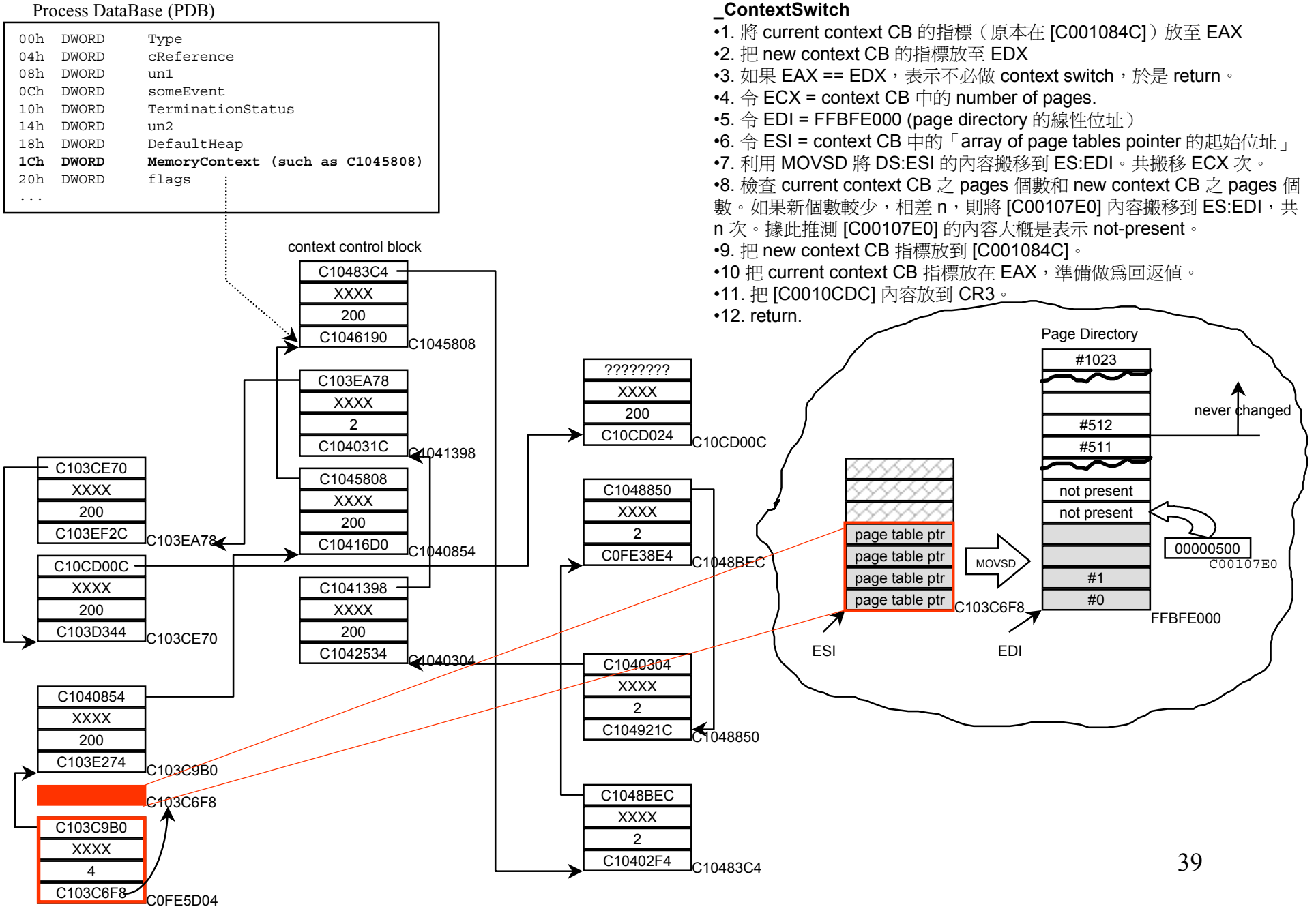
Memory Context Switch

Process DataBase (PDB)

```
00h  DWORD     Type
04h  DWORD     cReference
08h  DWORD     un1
0Ch  DWORD     someEvent
10h  DWORD     TerminationStatus
14h  DWORD     un2
18h  DWORD     DefaultHeap
1Ch  DWORD     MemoryContext (such as C1045808)
20h  DWORD     flags
...
```

_ContextSwitch

- 1. 將 current context CB 的指標（原本在 [C001084C]）放至 EAX
- 2. 把 new context CB 的指標放至 EDX
- 3. 如果 EAX == EDX，表示不必做 context switch，於是 return。
- 4. 令 ECX = context CB 中的 number of pages.
- 5. 令 EDI = FFBFE000 (page directory 的線性位址）
- 6. 令 ESI = context CB 中的「array of page tables pointer 的起始位址」
- 7. 利用 MOVSD 將 DS:ESI 的內容搬移到 ES:EDI。共搬移 ECX 次。
- 8. 檢查 current context CB 之 pages 個數和 new context CB 之 pages 個數。如果新個數較少，相差 n，則將 [C00107E0] 內容搬移到 ES:EDI，共 n 次。據此推測 [C00107E0] 的內容大概是表示 not-present。
- 9. 把 new context CB 指標放到 [C001084C]。
- 10 把 current context CB 指標放在 EAX，準備做爲回返值。
- 11. 把 [C0010CDC] 內容放到 CR3。
- 12. return.



GetSystemInfo

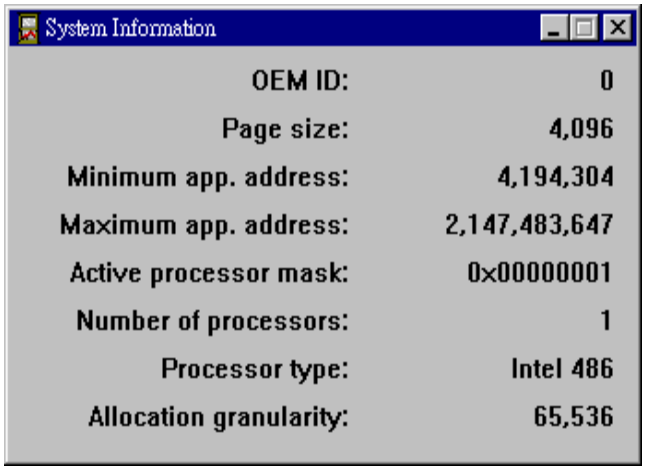
取得系統相關資訊 –分頁大小、區域之配置單位大小

SYNTAX:

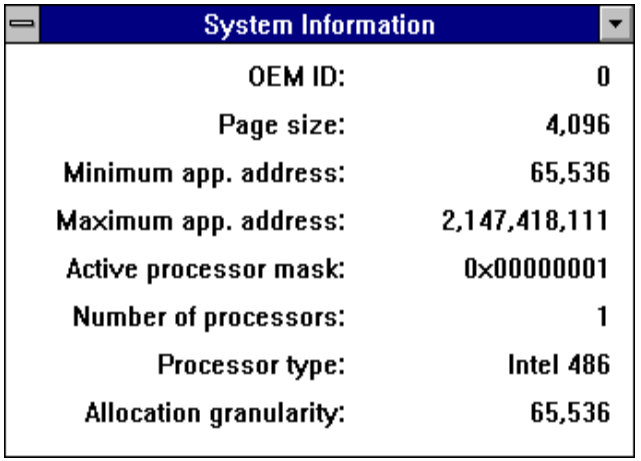
VOID GetSystemInfo (LPSYSTEM_INFO pSystemInfo**);** // 取得系統資訊(分頁大小、區域之配置單位大小...)

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD   dwOemID;
        struct {
            WORD    wProcessorArchitecture
            WORD    wReserved;
        };
    };
    DWORD       dwPageSize;
                // 4KB for x86, MIPS, PowerPC
                // 8KB for Alpha
    LPVOID      lpMinimumApplicationAddress;
                // 0x00400000 for Windows 95
                // 0x00010000 for Windows NT
    LPVOID      lpMaximumApplicationAddress;
                // 0x7FFFFFFF for Windows 95
                // 0x7FFEFFFF for Windows NT
    DWORD       dwActiveProcessorMask;
    DWORD       dwNumberOfProcessors;
    DWORD       dwProcessorType;
    DWORD       dwAllocationGranullarity;
                // 64KB
    WORD        wRProcessorLevel;
    WORD        wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

Windows 95



Windows NT on Intel x86

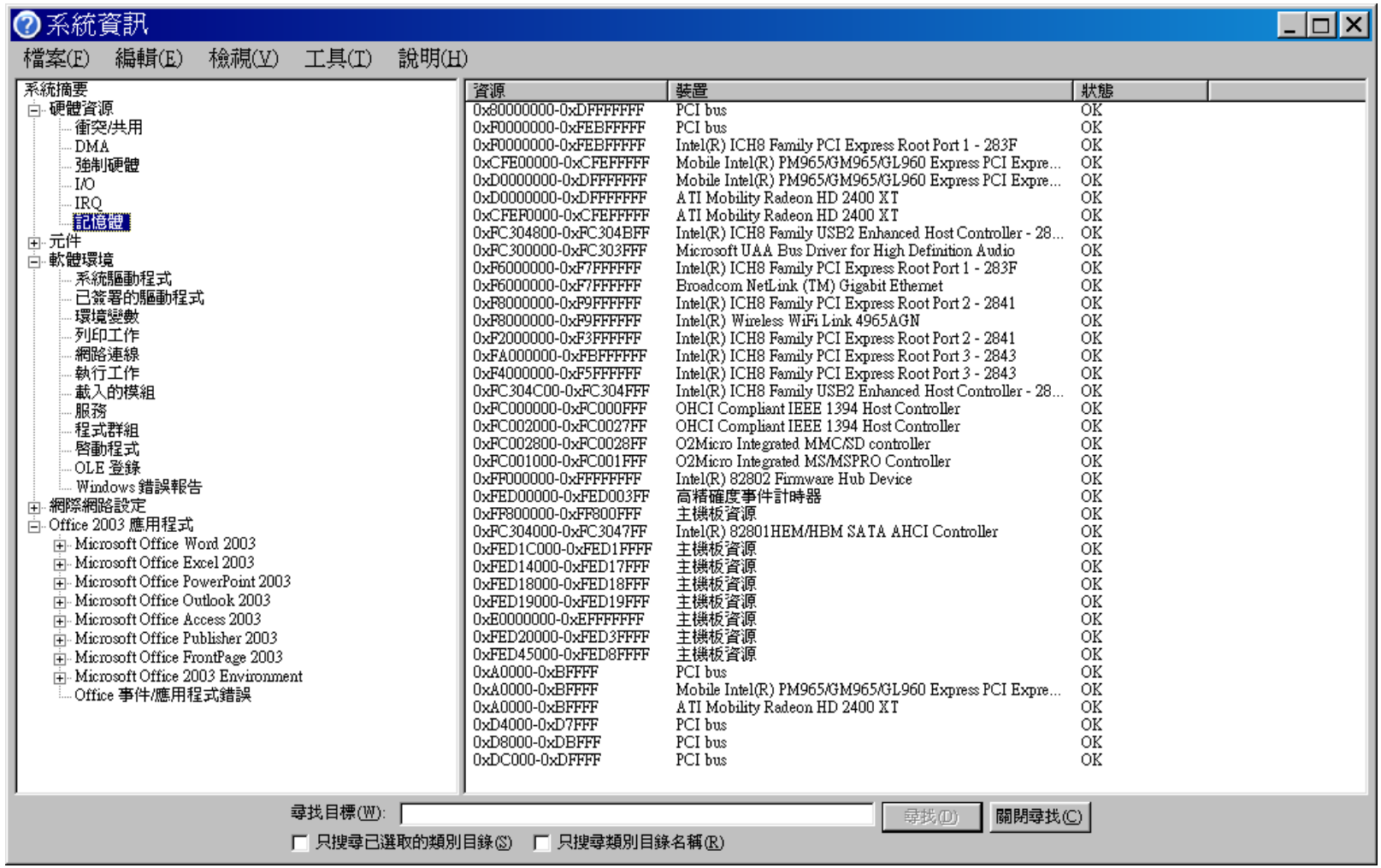


GetSystemInfo in Windows XP

Windows XP, v5.1 service pack2

```
dwPageSize: 4096
lpMinimumApplicationAddress: 00010000
lpMaximumApplicationAddress: 7FFEFFFF
dwActiveProcessorMask: 3
dwNumberOfProcessors: 2
dwProcessorType: 586
dwAllocationGranularity: 65536
wProcessorLevel: 6
wProcessorRevision: 5894
```

Windows XP


系統資訊
檔案(F) 編輯(E) 檢視(V) 工具(T) 說明(H)
系統摘要
硬體資源
衝突/共用
DMA
強制硬體
I/O
IRQ
記憶體
元件
軟體環境
系統驅動程式
已簽署的驅動程式
環境變數
列印工作
網路連線
執行工作
載入的模組
服務
程式群組
啓動程式
OLE 登錄
Windows 錯誤報告
網際網路設定
Office 2003 應用程式
Microsoft Office Word 2003
Microsoft Office Excel 2003
Microsoft Office PowerPoint 2003
Microsoft Office Outlook 2003
Microsoft Office Access 2003
Microsoft Office Publisher 2003
Microsoft Office FrontPage 2003
Microsoft Office 2003 Environment
Office 事件/應用程式錯誤
資源 裝置 狀態
0x80000000-0xDFFFFFFF PCI bus OK
0xF0000000-0xFEBFFFFF PCI bus OK
0xF0000000-0xFEBFFFFF Intel(R) ICH8 Family PCI Express Root Port 1 - 283F OK
0xCFE00000-0xCFEFFFFF Mobile Intel(R) PM965/GM965/GL960 Express PCI Expre... OK
0xD0000000-0xDFFFFFFF Mobile Intel(R) PM965/GM965/GL960 Express PCI Expre... OK
0xD0000000-0xDFFFFFFF ATI Mobility Radeon HD 2400 XT OK
0xCFEF0000-0xCFEFFFFF ATI Mobility Radeon HD 2400 XT OK
0xFC304800-0xFC304BFF Intel(R) ICH8 Family USB2 Enhanced Host Controller - 28... OK
0xFC300000-0xFC303FFF Microsoft UAA Bus Driver for High Definition Audio OK
0xF6000000-0xF7FFFFFF Intel(R) ICH8 Family PCI Express Root Port 1 - 283F OK
0xF6000000-0xF7FFFFFF Broadcom NetLink (TM) Gigabit Ethernet OK
0xF8000000-0xF9FFFFFF Intel(R) ICH8 Family PCI Express Root Port 2 - 2841 OK
0xF8000000-0xF9FFFFFF Intel(R) Wireless WiFi Link 4965AGN OK
0xF2000000-0xF3FFFFFF Intel(R) ICH8 Family PCI Express Root Port 2 - 2841 OK
0xFA000000-0xFBFFFFFF Intel(R) ICH8 Family PCI Express Root Port 3 - 2843 OK
0xF4000000-0xF5FFFFFF Intel(R) ICH8 Family PCI Express Root Port 3 - 2843 OK
0xFC304C00-0xFC304FFF Intel(R) ICH8 Family USB2 Enhanced Host Controller - 28... OK
0xFC000000-0xFC000FFF OHCI Compliant IEEE 1394 Host Controller OK
0xFC002000-0xFC0027FF OHCI Compliant IEEE 1394 Host Controller OK
0xFC002800-0xFC0028FF O2Micro Integrated MMC/SD controller OK
0xFC001000-0xFC001FFF O2Micro Integrated MS/MSPRO Controller OK
0xFF000000-0xFFFFFFFF Intel(R) 82802 Firmware Hub Device OK
0xFED00000-0xFED003FF 高精確度事件計時器 OK
0xFF800000-0xFF800FFF 主機板資源 OK
0xFC304000-0xFC3047FF Intel(R) 82801HEM/HBM SATA AHCI Controller OK
0xFED1C000-0xFED1FFFF 主機板資源 OK
0xFED14000-0xFED17FFF 主機板資源 OK
0xFED18000-0xFED18FFF 主機板資源 OK
0xFED19000-0xFED19FFF 主機板資源 OK
0xE0000000-0xEFFFFFFF 主機板資源 OK
0xFED20000-0xFED3FFFF 主機板資源 OK
0xFED45000-0xFED8FFFF 主機板資源 OK
0xA0000-0xBFFFF PCI bus OK
0xA0000-0xBFFFF Mobile Intel(R) PM965/GM965/GL960 Express PCI Expre... OK
0xA0000-0xBFFFF ATI Mobility Radeon HD 2400 XT OK
0xD4000-0xD7FFF PCI bus OK
0xD8000-0xDBFFF PCI bus OK
0xDC000-0xDFFFF PCI bus OK
尋找目標(W):
尋找(D)
關閉尋找(C)
只搜尋已選取的類別目錄(S)
只搜尋類別目錄名稱(R)

GlobalMemoryStatus

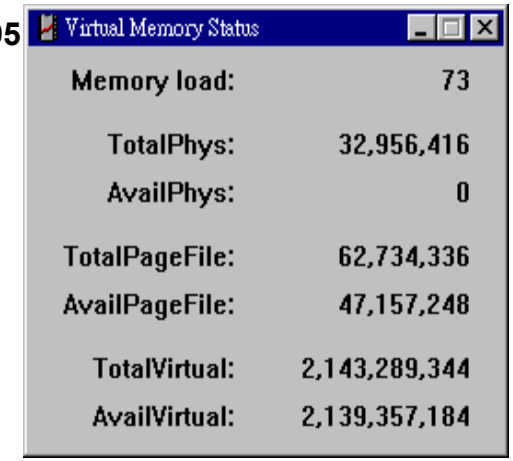
取得記憶體目前狀態

SYNTAX:

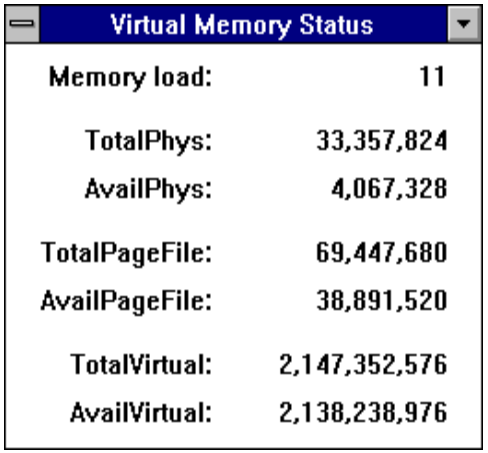
VOID GlobalMemoryStatus (LPMEMORYSTATUS pMemStatus**);** // 取得記憶體目前狀態

```
typedef struct _MEMORYSTATUS {
        DWORD    dwLenght;
                 // set to sizeof(MEMORYSTATUS) before call

        DWORD    dwMemoryLoad;
                 // 0..100 (對記憶體忙碌的程度, 意義不大)
        SIZE_T   dwTotalPhys;
        SIZE_T   dwAvailPhys;
        SIZE_T   dwTotalPageFile;
        SIZE_T   dwAvailPageFile;
        SIZE_T   dwTotalVirtual;
        SIZE_T   dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Windows 95



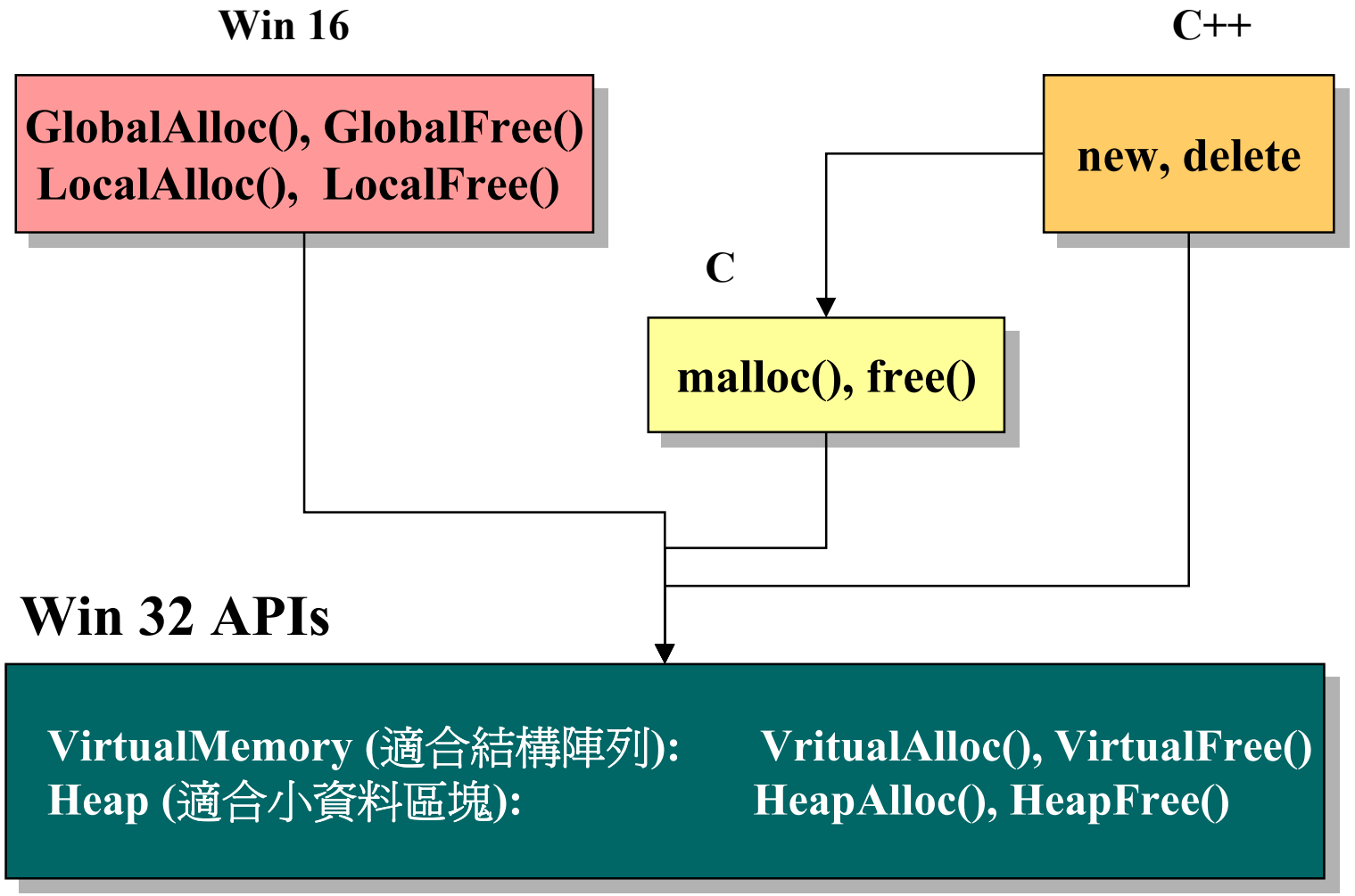
Windows NT on Intel x86



Windows XP

```
sizeof(MEMORYSTATUS): 32
percent of memory in use: 44
bytes of physical memory: 2145759232
free physical memory bytes: 1197162496
bytes of paging file: 4128817152
free bytes of paging file: 3200696320
user bytes of address space: 2147352576
free user bytes: 2139934720
```

Virtual Memory vs. Heap



Virtual Memory

虛擬記憶體技術上的優勢在哪裡 ？

撰寫一試算表應用程式：

程式特色 - 所需之資料儲存空間很大,但又只有極少數的資料格存放資料

方法一：(array)

```
CELLDATA  CellData[200][256];    // sizeof(CELLDATA) == 128
```

記憶體使用量太多 (約 6.25 MB)

方法二：(linking list)

尋找資料的速度太慢

方法三：(virtual memory)

有二維陣列存取便利的好處 + 連結串列節省實體儲存體的好處

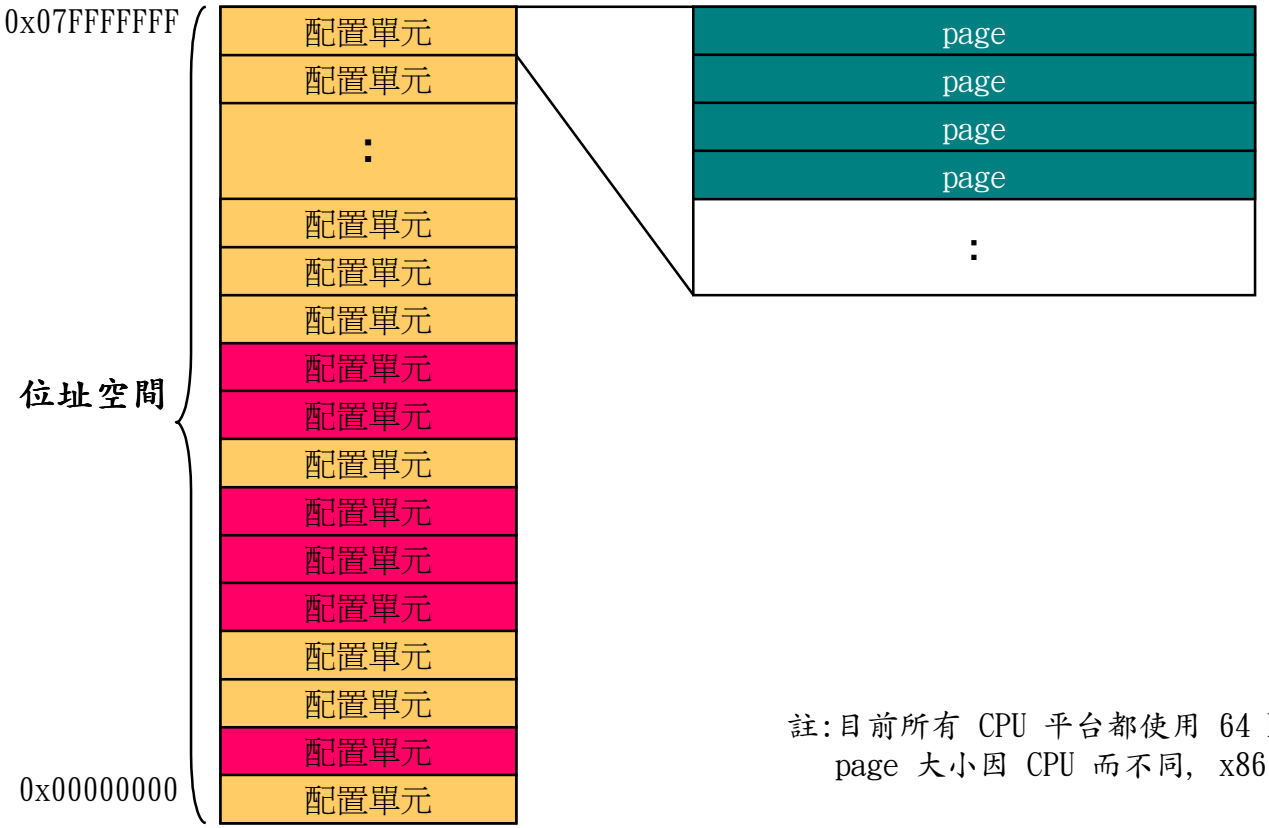
1. **保留一塊夠大的區域, 不要為它配置實際儲存體**
   VirtualAlloc(…, MEM_RESERVE, ...)
2. **有資料需要儲存時, 算出存放資料在區域中的真正位址, 引發“存取違規”異常**
   __except (...) {...}
3. **上述步驟中為該位址配置剛好夠用的實體儲存體**
   EXCEPTION_CONTINUE_EXECUTION
4. **在該位址上儲存資料**

Virtual Memory

位址空間的區域, 實際儲存體, 分頁檔 觀念-1

位址空間的區域: 系統建立一個行程時, 會給他一個私有的位址空間, 如果要使用此位址空間中的某一部份, 就必須向系統要求配置一塊區域, 不用時歸還.

VirtualAlloc() “保留”此一區域 (大小為 page 整數倍 eg. 10KB -> 12KB (x86))
VirtualFree() “釋放”此一區域



註:目前所有 CPU 平台都使用 64 KB 為一配置單元
page 大小因 CPU 而不同, x86:4KB, Alpha:8KB

Virtual Memory

位址空間的區域, 實際儲存體, 分頁檔 觀念-2

實際儲存體: 在使用一塊位址空間時, 必須為它安排實際儲存體
實際儲存體即現今作業系統硬碟上的的 page file(用時會移置RAM)
或 memory mapping file 之類的檔案(.EXE/.DLL/...)

VirtualAlloc() "配置"所需的 page file 空間
VirtualFree() "釋放"相關之 page file 空間

保護屬性	描述
PAGE_NOACCESS	不可對此 page 做任何讀取、寫入、 執行動作
PAGE_READONLY	只能讀取
PAGE_READWRITE	只能讀取、寫入
PAGE_EXECUTE	只能執行
PAGE_EXECUTE_READ	只能讀取、執行
PAGE_EXECUTE_READWRITE	任何動作均允許
PAGE_WRITECOPY	可讀取及寫入, 寫入時系統將給你複製一份私有的分頁資料
PAGE_EXECUTE_WRITECOPY	任何動作均允許, 寫入時系統將給你複製一份私有的分頁資料
bit OR with	
PAGE_NOCACHE	關閉 cache 功能
PAGE_WRITECOMBINE	給裝置驅動程式使用, 將對裝置的數個寫入動作結合, 加快執行效率
PAGE_GUARD	讓程式在此分頁有存取動作時, 會引發一例外狀況來穫得通知

Windows 98 僅支援此三類

Virtual Memory, Example

虛擬記憶體技術之使用例

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

#define PAGELIMIT   80
#define PAGESIZE    0x1000
#define REGIONSIZE(PAGELIMIT * PAGESIZE)
LPSTR      lpNxtPage;
DWORD    dwPages = 0;

int PageFaultFilter(DWORD dwCode)
{
    LPVOID    lpvResult;
    // the exception is not a page fault. skip
    if (dwCode != EXCEPTION_ACCESS_VIOLATION) {
          return EXCEPTION_CONTINUE_SEARCH;
    }
    printf("page fault exception\n");
    // otherwise, commit another page
    lpvResult = VirtualAlloc((LPVOID) lpNxtPage,
                  PAGESIZE, MEM_COMMIT,
                  PAGE_READWRITE);
    if (lpvResult == NULL) {
          return EXCEPTION_CONTINUE_SEARCH;
    }
    // page count+1, let lpNxtPage point to the next page
    dwPages++;
    lpNxtPage += PAGESIZE;
    // continue execution
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

```
void main(void)
{
    // reserve pages in the process's virtual address space
    LPTSTR    lpvBase;
    lpvBase = (LPTSTR) VirtualAlloc(NULL, REGIONSIZE,
                  MEM_RESERVE, PAGE_NOACCESS);
    if (lpvBase == NULL)     return;
    lpNxtPage = lpvBase;

    // use try-except structured exception handling
    // when accessing the pages. If a page falut occurs,
    // the exception filter is executed to commit
    // another page from the reserved block of pages.
    for (int i=0 ; i < REGIONSIZE ; i++) {
        __try {
              lpvBase[i] = 'a';
        }
        // if there is a page fault,
        // commit another page and try again...
        __except (PageFaultFilter(GetExceptionCode())) {
            // nothing to do...
        }
    }
    // release the entire block
    VirtualFree(lpvBase, 0, MEM_RELEASE);
}
```

Heap

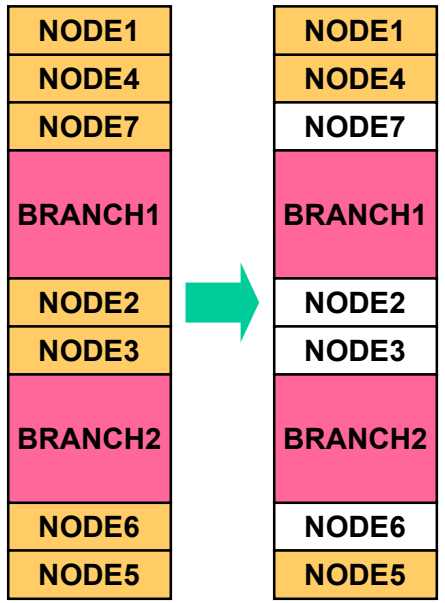
Heap 技術上的優勢在哪裡 ?

有關 Heap 的幾個觀念:

. process 初始會在其位址空間建立一個 Heap (Default Heap)
. 初始大小約為 1MB 可視需要動態加大
. 系統保證同一時間只有一個 thread 可以在 Default Heap 配置與釋放記憶體
. 一個process 可同時擁有多個 Heaps，process 執行期間可建立及摧毀 Heaps

自行建立並管理 Heap 的理由:

1. 如果資料不會被多緒存取，自行建立 Heap 可避免同步處理的負擔
2. 若需配置相同大小之資料物件，使用 Heap 可減少出現記憶體破碎情形，更有效率地管理記憶體
3. 集中存取相關物件(如一連結串列的所有元素)，可大幅減少RAM與pagefile的置換動作(swapping)
4. 如果為單一資料結構建立一專屬 Heap，可以很簡單地一次釋放整個 Heap，不需逐一釋放每個元件



Heap的優點是: programmer 無需理會配置單元大小、分頁邊界對齊等問題
Heap的缺點是: 記憶體的配置與釋放較慢，且無法控制實際儲存體的安排與解除

Heap, example

Heap 技術之使用範例

```
class CSomeClass
{
private:
        static HANDLE  hHeap;
        static UINT        uNumAllocsInHeap;
        . . .
public:
        void *operator  new (size_t size);
        void operator   delete (void *p);
        . . .
};

HANDLE   CSomeClass::hHeap = NULL;
UINT CSomeClass:: uNumAllocsInHeap = 0;
```

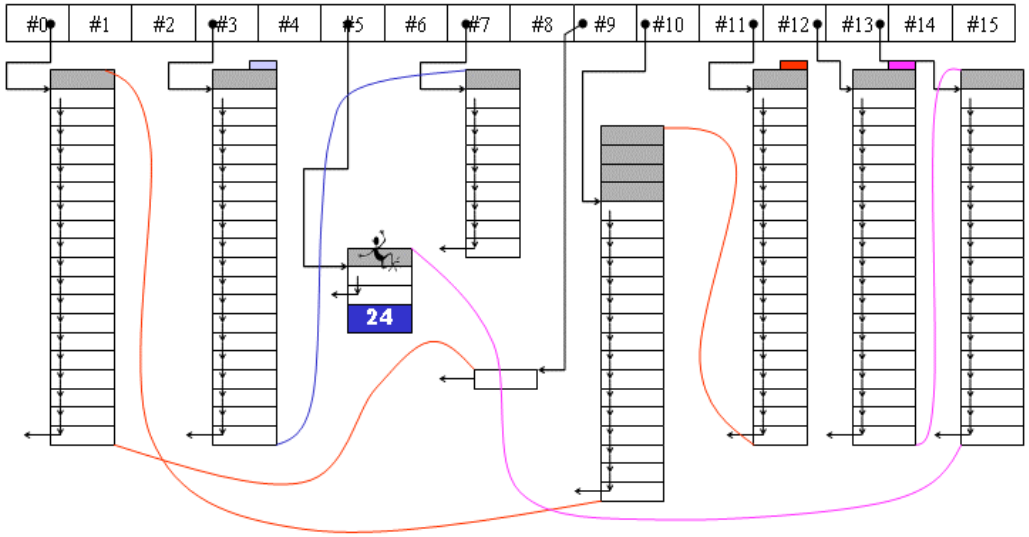
```
void* CSomeClass::operator new (size_t size)
{
        if (hHeap == NULL) {
                // Heap does not exist; create it.
                hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);
                if (hHeap == NULL)              return NULL;
        }
        // The heap exists for CSomeClass objects.
        void *p = HeapAlloc (hHeap, 0, size);
        if (p != NULL) {
                uNumAllocsInHeap++;
        }
        return p;
}
```

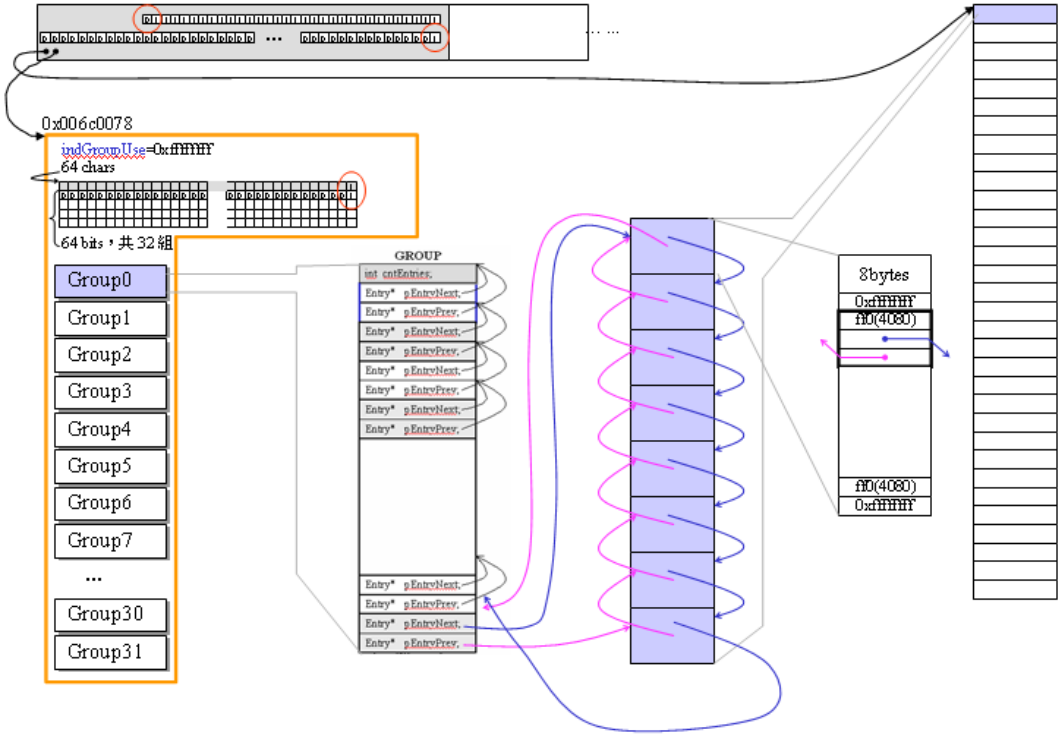
```
void CSomeClass::operator delete (void *p)
{
        if ( HeapFree(hHeap, 0, p) ) {
                // Object was deleted successfully.
                uNumAllocsInHeap--;
        }
        if (uNumAllocsInHeap == 0) {
                // If there are no more objects in
                // the heap, destroy the heap.
                if (HeapDestroy(hHeap)) {
                        // Set the heap handle to NULL so
                        // that the new operator will know
                        // to create a new heap if a new
                        // CSomeClass object is created.
                        hHeap = NULL;
                }
        }
}
```

```
CSomeClass  *pInstance = new CSomeClass;
delete   pInstance;
```

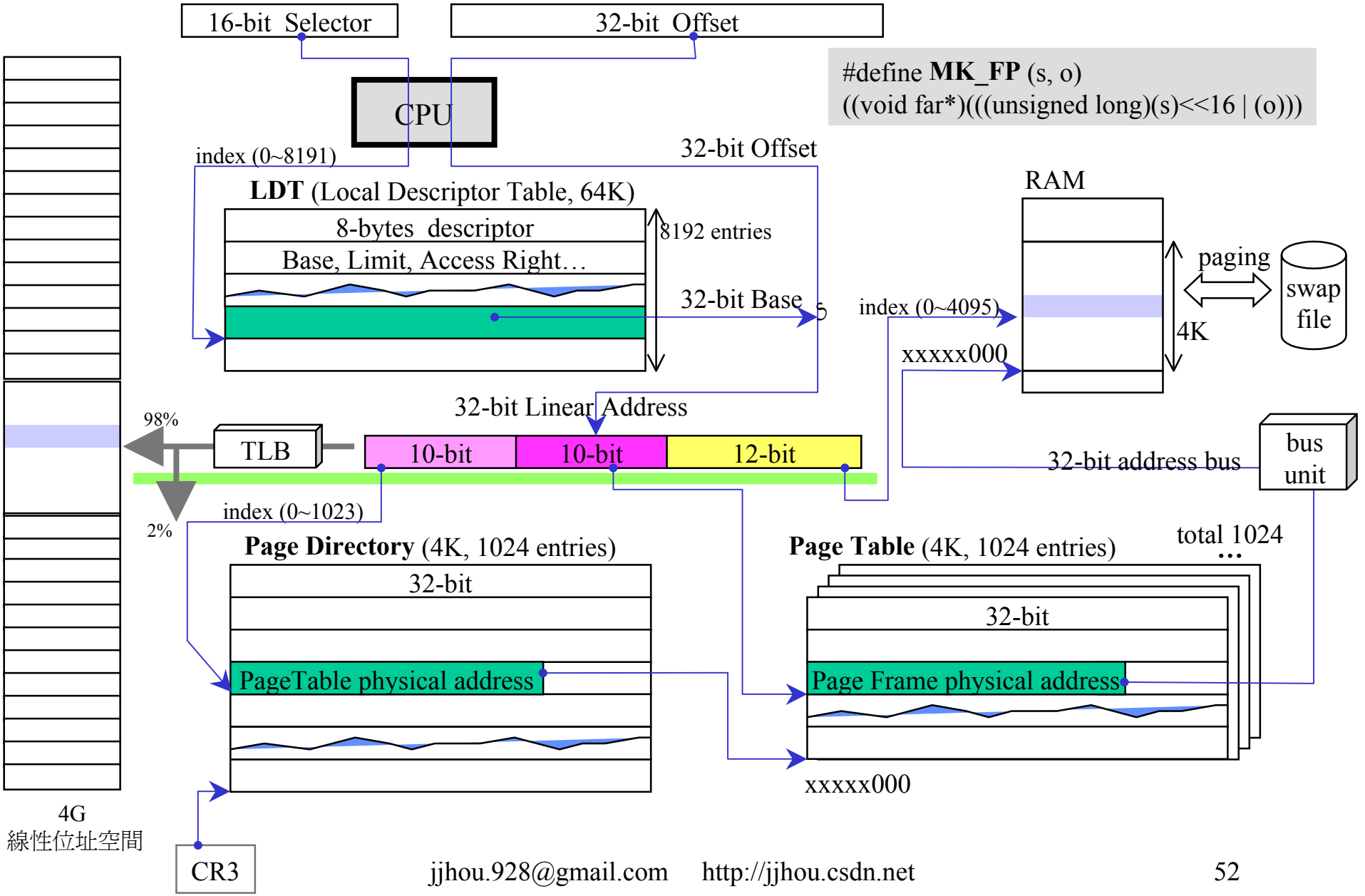
allocator ⇔ malloc

std::alloc 每次向 malloc 要求區塊，都是 size*20*2+roundup(heapsize>>4)，相當大。因此除了剛開始可能落入 malloc's sbh，其他都應該會 > 3F8 (這是 _sbh_threshold)，因而直接由 HeapAlloc() 處理。
(見 _heap_alloc_base())

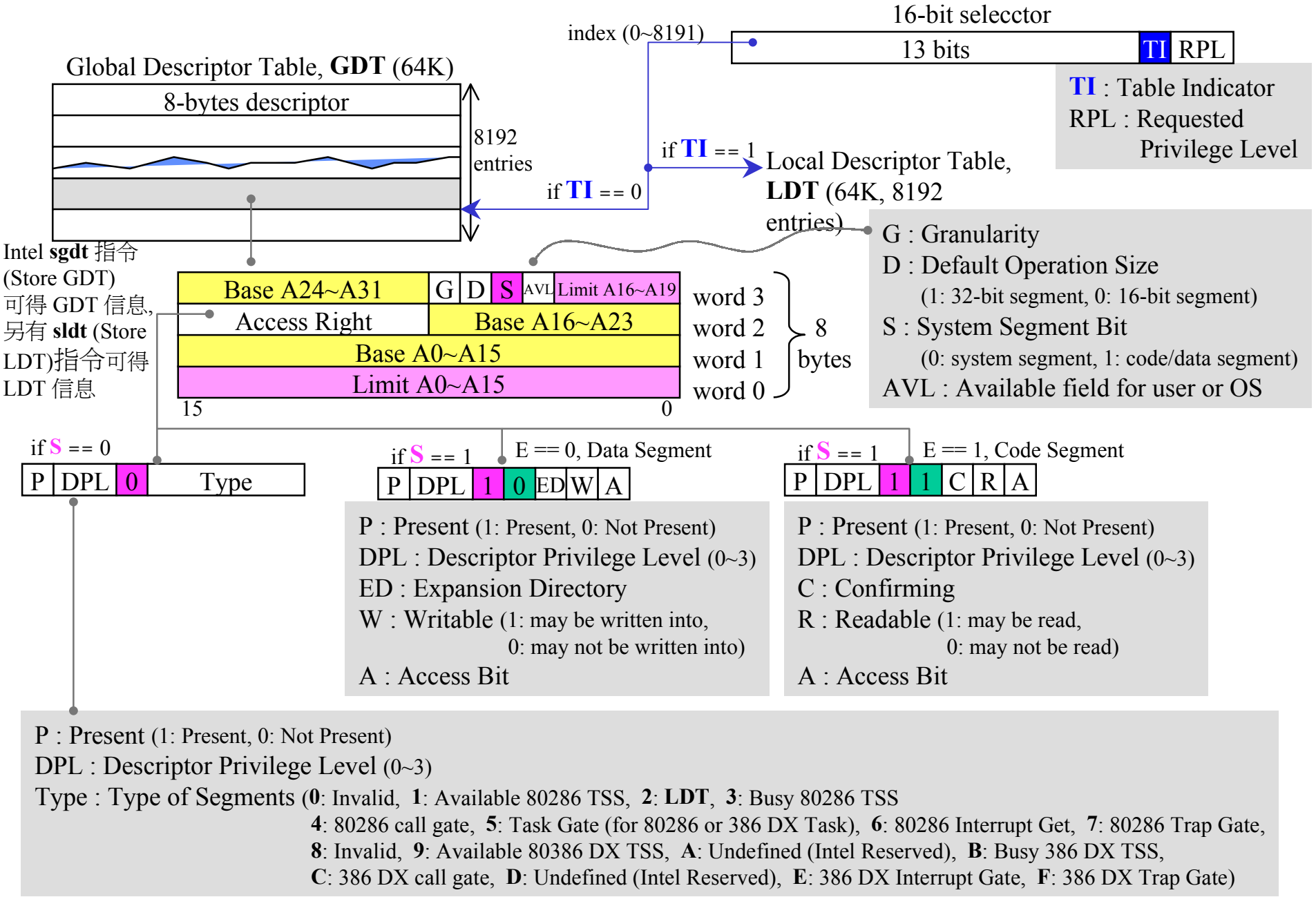




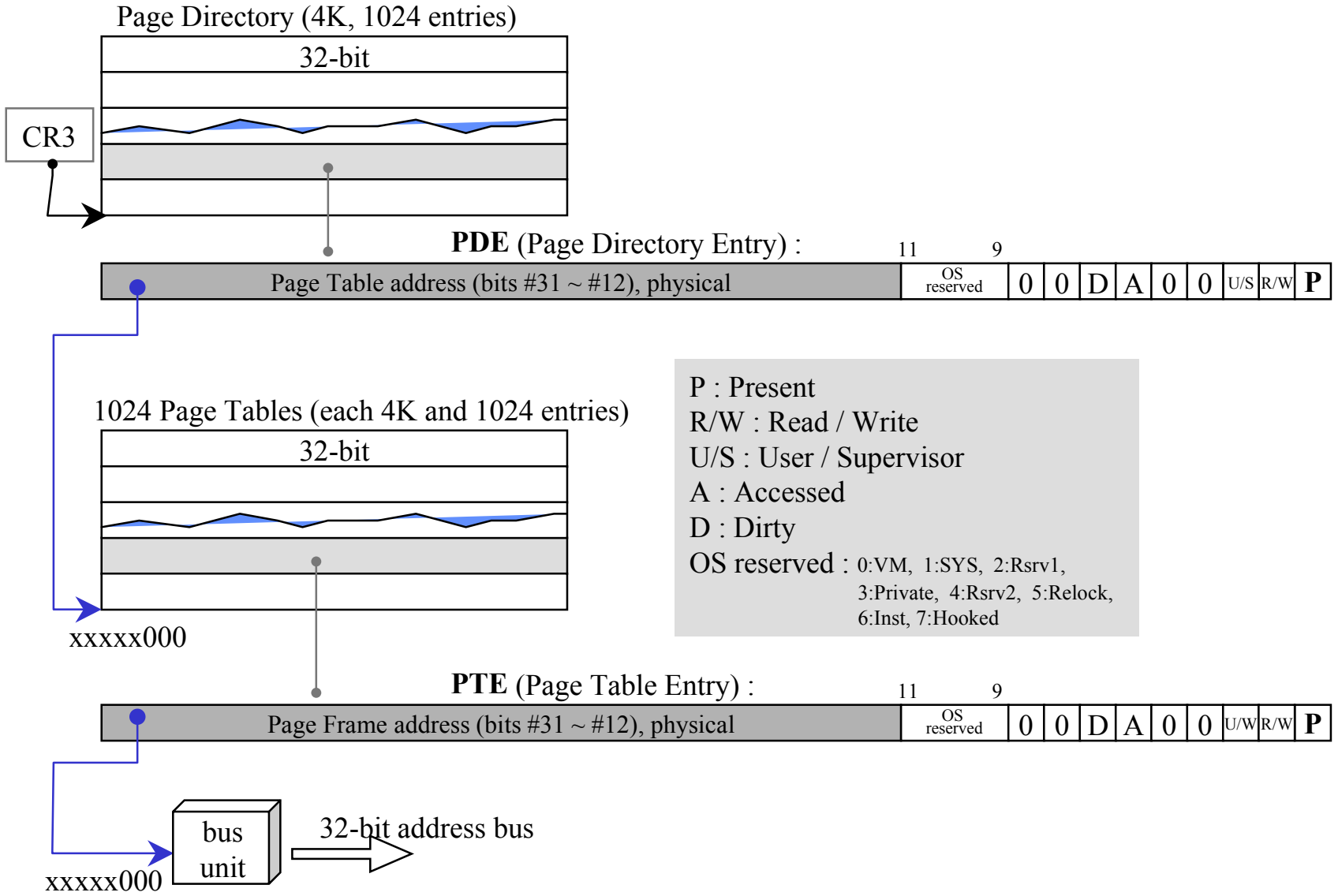
32-bit Addressing



Global Descriptor & Local Descriptor

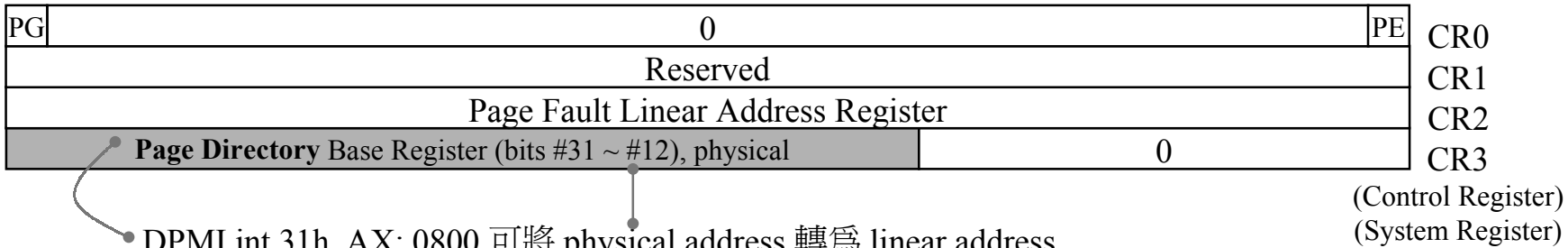


Page Directory & Page Tables



CR0 ~ CR3

PG : Paging Enable
PE : Protection Enable



DPMI int 31h, AX: 0800 可將 physical address 轉爲 linear address
欲將 linear addr. 轉爲 pointer 可用 Selector 函式（但 Win95 之後都不支援）

Win32 Modules

三個緊密相關的資料結構，構成 Windows 95 的核心：module（模組）、process（行程）和 thread（執行緒）。很難找到哪一個重要的 Windows API 函式沒有與它們有關聯。

Windows 3.x 有所謂 module 和 task。在 Win32 中，task 的觀念被分割爲兩部份：processes 和 threads。

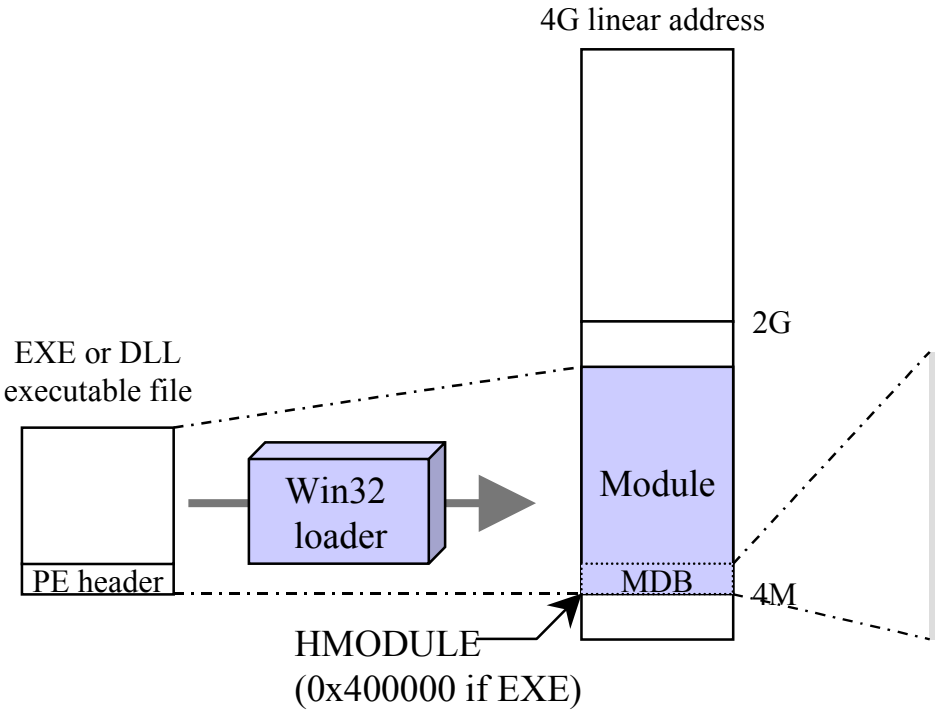
一個 Win32 module 代表一個被 Win32 loader 載入的 EXE 或 DLL 的程式碼、資料、資源。因此，記憶體中的一個 module 對應磁碟中的一個 file。Loader 利用記憶體映射檔，將 PE 檔中的某一段落映射到線性記憶體中。作業系統把一個 loaded module 的所有高階信息保持在一個結構之中，此結構稱爲 **module database**。

HMODULEs 用來代表 loaded module。Win32 讓一個 HMODULE 成爲 Win32 loader 映射 PE 檔時的起始線性位址。例如大部份 EXE 程式被載入於位址 0x400000（4MB）處，所以它們的 HMODULE 就是 0x400000。這意味多個 EXE 同時執行時擁有相同的 HMODULE。這不是問題，因爲Windows 95 和 NT 爲每一個行程維護了一個分離位址空間。

Module database 非常靠近 EXE 或 DLL 被載入後的記憶體起頭處，內含像是檔案中的 code/data sections 被載入到記憶體何處等等的資訊。Module 的程式碼和資料不僅止於編譯器爲程式所生的碼，還包括 import table、export table、resource directory...等等。

Win32 Modules

Win32 的 module database 其實就是 EXE 或 DLL 的 PE 表頭。WINNT.H 內有 IMAGE_NT_HEADERS 結構，由一個 DWORD 和兩個子結構組成。其內信息就是 Windows 95 內部用來尋找被載入之 EXE 或 DLL 的程式碼、資料、資源用的。




```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32,
  *PIMAGE_NT_HEADERS32;
```

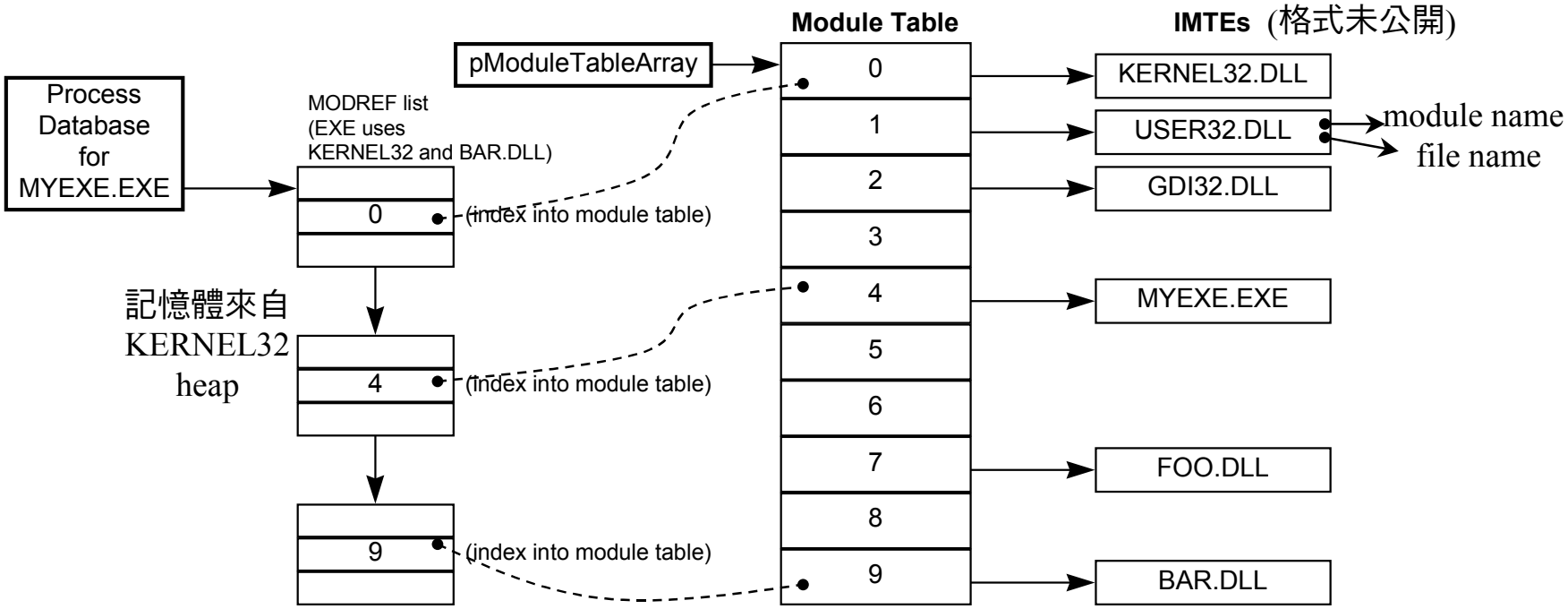
Win32 要求每一個 process 有自己的 module list。從應用程式眼光來看是不錯，但從 KERNEL32 的角度來看，單一 module list 比較容易達到程式碼和資源的共享。KERNEL32 利用兩個資料結構來維護一個 global module list，並使它看起來好像每一個 process 有自己的一個 module list。第一個資料結構是 **IMTE**（Internal Module Table Entry），第二個資料結構是 **MODREF**。

IMTE (Internal Module Table Entry)

Global KERNEL32 module list 其實只是 IMTEs 指標所組成的 array。其所使用的記憶體是從 KERNEL32 heap 中配置而來，那是一般的 *HeapAlloc* 所得。當新模組被載入或踢出記憶體，KERNEL32 利用 *HeapReAlloc* 動態擴張或縮減 array 大小。當 KERNEL32 產生一個新的 IMTE，它會搜尋 pModuleTableArray 中的空白元素；找到了一個，就把 IMTE 指標放進去。pModuleTableArray 的第一個元素（索引爲 0）用來表示 KERNEL32.DLL 模組。

KERNEL32 以一個 global array（內含 IMTEs 指標）維護所有 modules。把 all process private module lists 和 global module list 連接在一起的就是 **MODREF**。每個 process（除了奇怪的KERNEL32）私有的 module list 其事是個 MODREF list，其中一個 MODREF 針對 process 自己（因它自己也構成一個 module），其他 MODREFs 針對該 process 所使用的每一個 Win32 DLLs。MODREFs 的記憶體來自 KERNEL32 heap，處於 2GB 之上，是一塊可共享區。MODREFs list 的頭放在 **process database** 內。每一個 MODREFs 內含一個索引，指向 pModuleTableArray 表格。

MODREF



Global KERNEL32 module table
i.e., array of pointers to IMTE
array size 可動態增減.
記憶體來自KERNEL32 heap

K32 objects

K32 物件型態：

K32OBJ_SEMAPHORE(0x1)
K32OBJ_event(0x2)
K32OBJ_mutex(0x3)
K32OBJ_CRITICAL_SECTION(0x4)
K32OBJ_**PROCESS**(0x5)
K32OBJ_**THREAD**(0x6)
K32OBJ_FILE(0x7)
K32OBJ_CHANGE(0x8）
K32OBJ_CONSOLE(0x9)
K32OBJ_SCREEN_BUFFER(0xA)
K32OBJ_MEM_MAPPED_FILE(0xB）
K32OBJ_SERIAL(0xC)
K32OBJ_DEVICE_IOCTL(0xD）
K32OBJ_PIPE(0xE)
K32OBJ_MAILSLOT(0xF)
K32OBJ_TOOLHELP_SNAPSHOT(x10）
K32OBJ_SOCKET(0x11)

佔用 KERNEL32 heap 的任何東西都可以說是 KERNEL32 物件，但另一套定義是：

K32 物件是關鍵性的系統資料結構，放在 KERNEL32 heap 中。各式各樣的 K32 物件都以相同的表頭開始。決定是否爲一個 K32 物件的方法就是詢問：應用程式中可有 handles 代表此一物件？例如程式可以擁有 file handles 或 event handles，所以 file 和 event 都是 K32 物件。我不曾看過任何程式碼擁有 MODREF 或 IMTE 的 handles，所以它們不是 K32 物件。

每一個 K32 物件都以一個共通的表頭開始。此表頭擁有以下欄位：

00h　DWORD　物件類型
　　此值決定後續成員如何解釋
04h　DWORD　物件的參用次數

一個 process database 其實就是一個 K32_PROCESS 物件，一個 thread database 其實就是一個 K32_THREAD 物件。一個 process handle table 其實就是一個 array of pointer to 各式各樣的 K32 物件。KERNEL32 和 VWIN32.VXD 程式碼會先檢查物件的第一個 DWORD，確定它所處理的物件型態。

Windows 95 Process

Process 其實就是 a unit of ownership。也就是說 process 擁有 things。它可以擁有 memory（更精確說是 memory context）、file handles、threads、a list of DLL modules（被載入於此 process 的位址空間）…。

行程並不表現出 execution（執行緒才表現 execution of code）。行程也不是 EXE 檔。被載入前 EXE 檔只不過是個 program，被載入後 Windows 95 才爲它產生一個 process。換句話說一個行程關係到一個 disk file（但 KERNEL32 是個例外）。

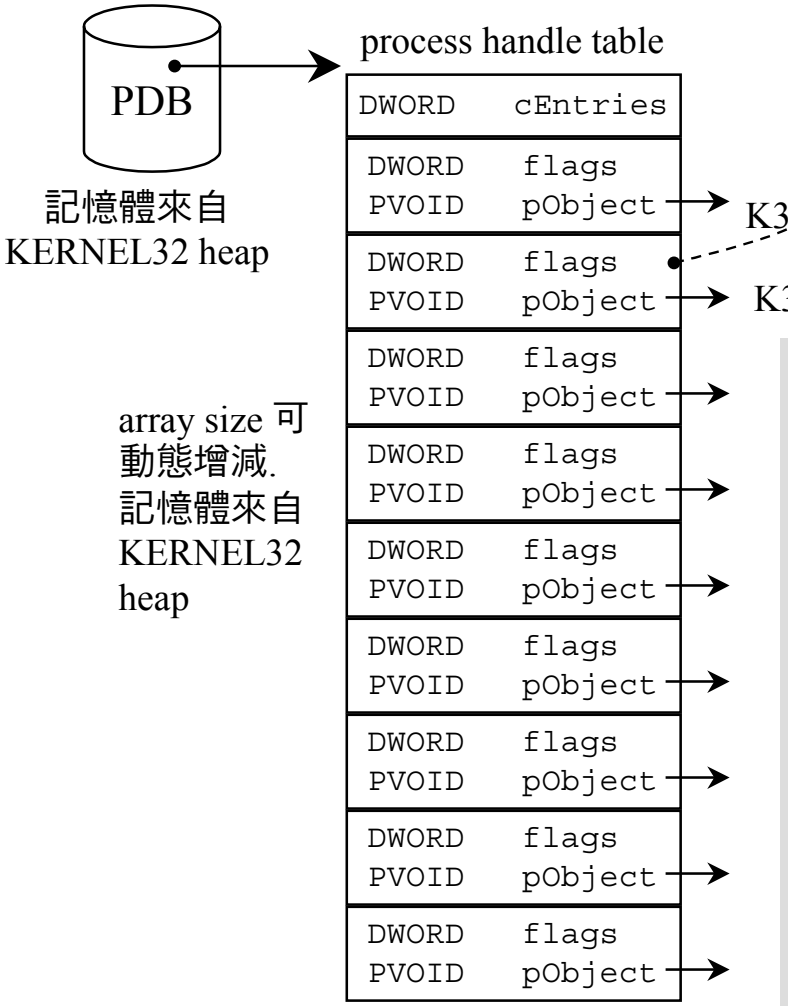
一旦 Windows 95 產生一個 process，它也產生一個 memory context，容納行程的執行緒在其中執行，並爲之產生第一個執行緒，用來執行行程本身。如有必要，行程可以再產生執行緒。系統還會產生一個 file handle table，行程可以在其中持有一些開啓的檔案。最後也是最重要的，Windows 95 產生一個 **process database**，用以表現這個行程。

Process database 是一種 K32 物件，內含與行程有關的大量資訊。Process database 所使用的記憶體來自 KERNEL32 heap，因此所有的 process database 都可以被其他行程看到。

Process database 含有 a list of threads、a list of loaded modules、default process heap handle、pointer to process handle table、以及 pointer to memory context，以及更多東西。

Process handle vs. Process ID
GetCurrentProcess, GetCurrentProcessId

Process database 含有 pointer to **process handle table**



兩個看起來類似的 Win32 函式：*GetCurrentProcess* 和 *GetCurrentProcessId*。前者獲得 handle 而後者獲得 id.

Process handle 不是任何東西的指標。程式不需要直接處理 handle table，所以 process handle 對程式而言無用。

不同的行程在各自位址空間中擁有相同的 process handle 絕對可能。process handle 不是個可賴以辨識行程的資料。

process handle 就像 file handle 一樣在其行程之外別無意義。process ID 則是各行程間獨一無二不會衝突的數值，它是 pointer to process database，但微軟在其中加了點料（Obsfucator）

TOOLHELP32 的 *Process32First* 和 *Process32Next* 所使用的 PROCESSENTRY32 結構中有 th32ProcessID 欄位，和 *GetCurrentProcessID* 傳回的東西有關係。

Windows 95 Process DataBase (PDB)

Windows 95 的每一個 process database 都從 KERNEL32 heap 配置而來。每一個 PDB 就是一個「第一字組爲 5（K32OBJ_PROCESS）」的 K32 物件。

節錄

00h DWORD **Type**
此值必爲 5（K32OBJ_PROCESS）

04h DWORD **cReference**
參用次數（reference count）。也就是此一 PDB 被使用的次數。

10h DWORD TerminationStatus
當你呼叫 *GetExitCodeProcess*，傳回的就是這個值。所謂 exit code 就是 *main* 或 *WinMain* 的回返值，可被 *ExitProcess* 或 *TerminateProcess* 指定。當一個行程還在執行時，此欄位爲 0x103（STILL_ACTIVE）。

1Ch DWORD **MemoryContext**
一個指標，指向行程的 memory context。

2Ch WORD **cThreads**
記錄此一行程擁有的執行緒個數。

34h HANDLE HeapHandle
一個 Heap handle，該 Heap 內含屬於這個行程的表格（或可能其他東西）。

PDB

3Ch DWORD MemMapFiles
一個指標，指向「此一行程所使用之記憶體映射檔所組成的串列」中的第一個節點。每一個記憶體映射檔都出現爲串列中的一個節點。

40h PENVIRONMENT_DATABASE **pEDB**
一個指標，指向 environment database，內含目前的子目錄、環境變數、行程命令列、標準 handles 如 stdin 以及其他項目。

44h PHANDLE_TABLE **pHandleTable**
一個指標，指向 process handle table。所有 handles 都在其中，包括file handles、event handles、process handles 等等。

48h struct _PROCESS_DATABASE * ParentPDB
一個指標，指向父行程的 PROCESS_DATABASE。對一般程式而言父行程是Windows 95 的檔案總管（Explorer）。MSGSRV32 則又是 Explorer 和 initial “service” processes 的父行程。

4Ch PMODREF **MODREFlist**
這個欄位指向行程的 MODREF list 起頭。

50h DWORD **ThreadList**
一個指標，指向此一行程所擁有的 Thread list。

54h DWORD DebuggeeCB
這是一個被除錯程式的 context。當一個行程被除錯，這個欄位即指向 2GB 以上的一個區域，該區域包含一個指標，指向被除錯者的 process database。

PDB

58h DWORD **LocalHeapFreeHead**
這個指標指向「process default heap」的 free blocks list 起頭。

60h CRITICAL_SECTION crst
這是一個 CRITICAL_SECTION，被各式各樣的 API 用來同步控制同一行程中的各執行緒。

84h DWORD pConsole
如果行程使用 console（亦即它是個文字模式程式），此欄位指向一個用於輸出的 console 物件（K32OBJ_CONSOLE）。

88h DWORD tlsInUseBits1
這 32 個位元表示最低的 32 個 TLS（Thread Local Storage）的索引。某位元設立，表示對應的 TLS 索引被用掉了。

8Ch DWORD tlsInUseBits2
表示 TLS 之中第 32~63 個索引的狀態。

A0h DWORD **BasePriority**
行程的排程優先權。

A4h DWORD **HeapOwnList**
指向「行程的所有 heaps」形成的 list。預設情況下每個行程只有一個 heap，可由 *GetProcessHeap* 取得。行程也可以呼叫 *HeapCreate* 產生另一個 heap。這些 heaps 都放在這個 list 中。

Process Handle Table

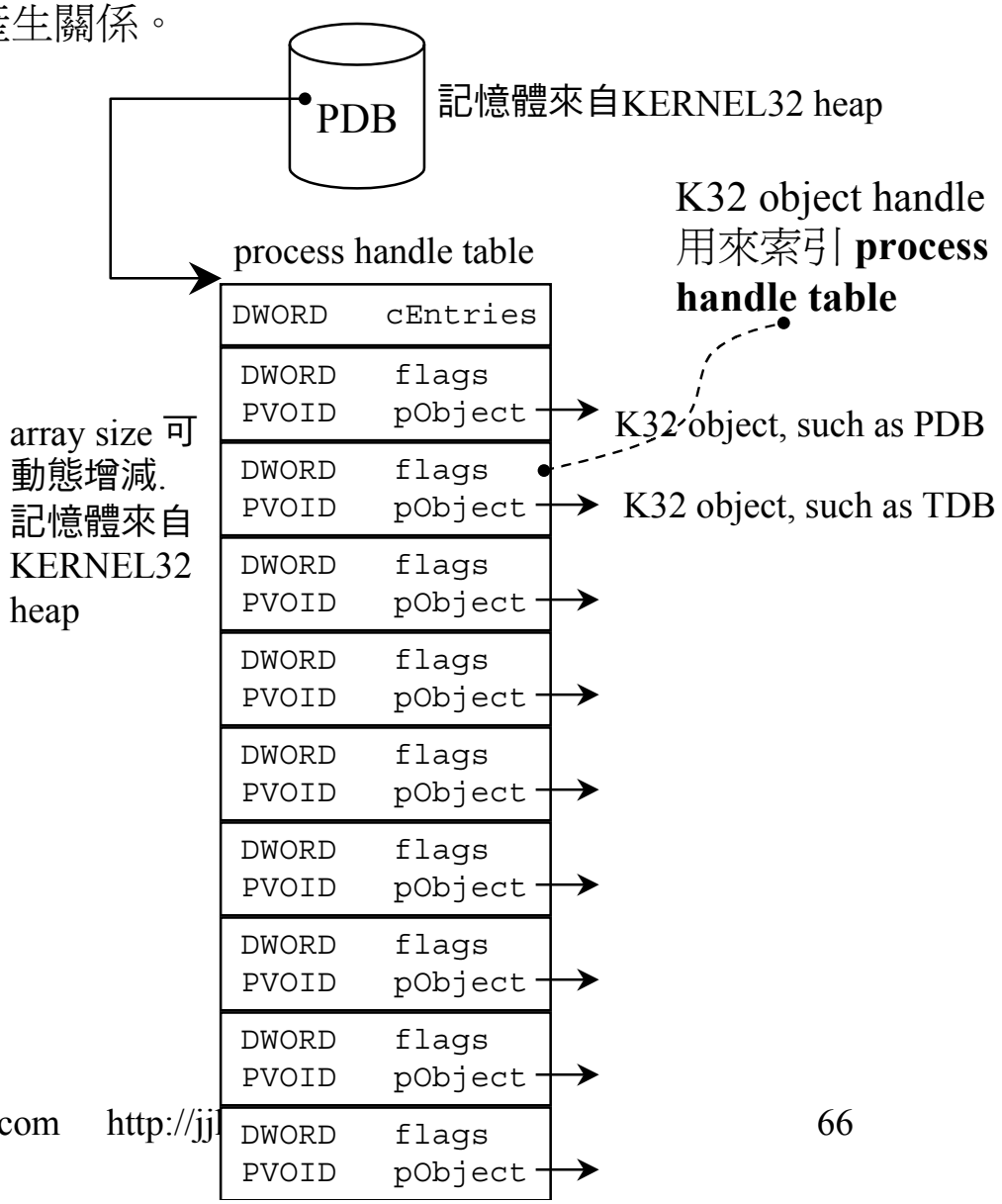
handle 一詞代表「KERNEL32 物件」。一旦了解 Windows 95 process handle tables，就可以輕易地將一個 handle 值和其參考到的資料產生關係。

第一個 DWORD 放的是表格的最大容量（entries 個數）。初始爲 0x30（48）。當行程需要更多的 handles，KERNEL32 會重新配置一塊記憶體，使表格有成長空間。每次增加 0x10 個 handles。似乎沒有上限。

之後是由結構組成的陣列。每一個結構都由兩個 DWORD 構成：

- DWORD flags 物件的access control flags，其意義視物件類型而定。
- DWORD pK32Object 指標，指向 K32 物件。

handle 是個 index into process handle table。一個 unused handle，其兩個 DWORD 一定都填滿 0。當程式配置一個 new handle，KERNEL32 就使用 handle table 中的第一筆 empty slot 的索引值做爲該 handle 值。

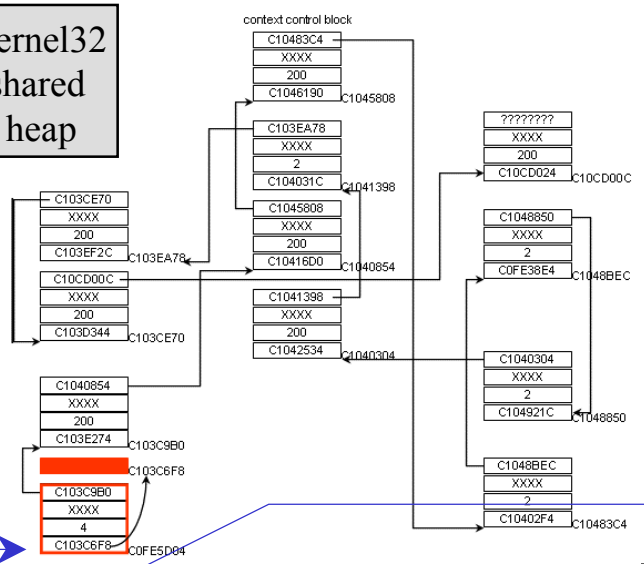


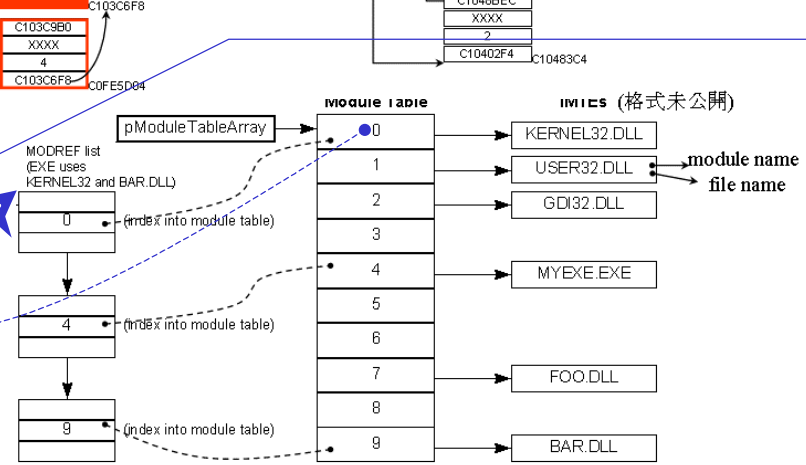
PDB

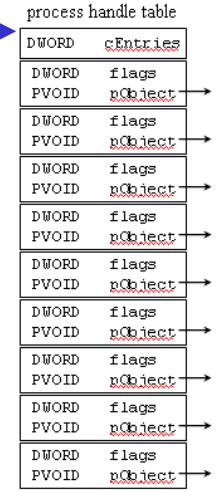
```
00h  DWORD      Type
04h  DWORD      cReference
08h  DWORD      un1
0Ch  DWORD      someEvent
10h  DWORD      TerminationStatus
14h  DWORD      un2
18h  DWORD      DefaultHeap
1Ch  DWORD      MemoryContext
20h  DWORD      flags
24h  DWORD      pPSP
28h  WORD       PSPSelector
2Ah  WORD       MTEIndex
2Ch  WORD       cThreads
2Eh  WORD       cNotTermThreads
30h  WORD       un3
32h  WORD       cRing0Threads
34h  HANDLE     HeapHandle
38h  HTASK      W16TDB
3Ch  DWORD      MemMapFiles
40h  PENVIRONMENT_DATABASE    pEDB
44h  PHANDLE_TABLE    pHandleTable
48h  struct _PROCESS_DATABASE* ParentPDB
4Ch  PMODREF    MODREFlist
50h  DWORD      ThreadList(格式未知)
54h  DWORD      DebuggeeCB
58h  DWORD      LocalHeapFreeHead
5Ch  DWORD      InitialRing0ID
60h  CRITICAL_SECTION    crst
78h  DWORD      un4[3]
84h  DWORD      pConsole
88h  DWORD      tlsInUseBits1
8Ch  DWORD      tlsInUseBits2
90h  DWORD      ProcessDWORD
94h  struct _PROCESS_DATABASE* ProcessGroup
98h  DWORD      pExeMODREF
9Ch  DWORD      TopExcFilter
A0h  DWORD      BasePriority
A4h  DWORD      HeapOwnList
A8h  DWORD      HeapHandleBlockList (for moveable bl
ACh  DWORD      pSomeHeapPtr
B0h  DWORD      pConsoleProvider
B4h  WORD       EnvironSelector
B6H  WORD       ErrorMode
B8h  DWORD      pevtLoadFinished
BCh  WORD       UTState
```

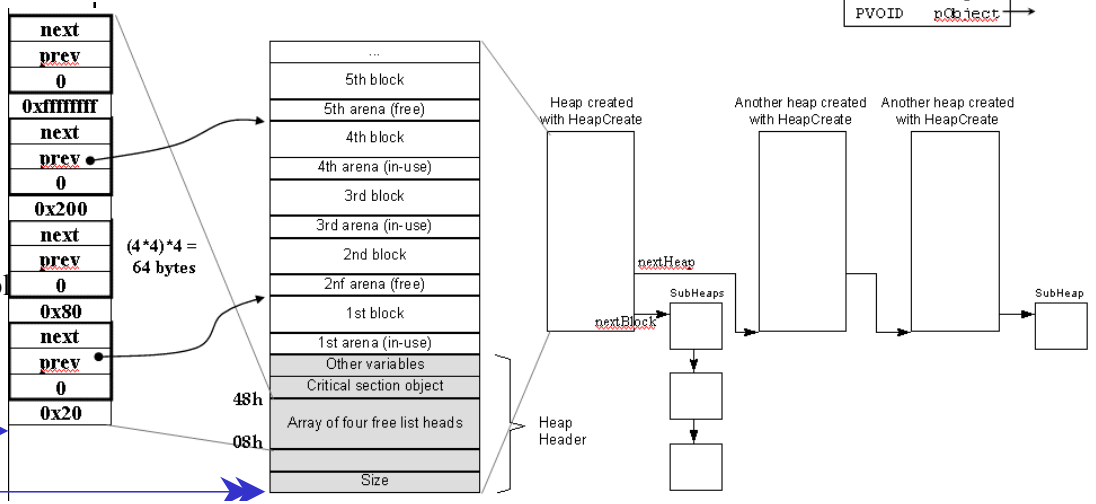
kernel32 shared heap

如何找出 process 所有 threads?
若從 **50h ThreadList**，格式未知呀!









Threads

執行緒主要表達 the execution code through modules。

從抽象層面來說，執行緒是 (a convenient way to keep various portions of your program running while other portions are waiting for some external action to occur) 一種便利的方式，讓你的某一部份程式碼執行 -- 當其他部份正在等待某些外部事件發生時。

任何時候，執行緒可能處於三種狀態之一。第一種是 **running** state。這個時候 CPU 暫存器內容就是執行緒的暫存器值。此時其他執行緒被虛懸 (suspended)。

第二種情況稱爲 **ready to run** state。這種狀態下的執行緒沒什麼理由不被執行 -- 時間早晚而已。它終有一刻能夠控制 CPU。

第三種情況稱爲 **blocked** state。執行緒如果被阻塞，表示它正在等待某件事發生。在那之前排程器不會配置執行緒執行起來。引起執行緒阻塞的東西稱爲同步控制物件（synchronization objects）。Windows 95 的同步控制物件有 critical sections、events、semaphores、mutexes 四種。

TDB (Thread DataBase) & THCB (Thread Control Block)

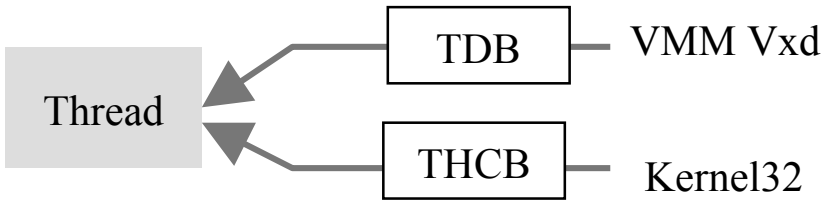
每個行程以一個執行緒開始，若需要可產生更多執行緒，使 CPU 得以在同一時間執行行程中不同區段的碼。

只有一顆 CPU 的機器不可能同時執行兩件工作。「許多個執行緒同時執行」的幻覺是靠 VMM 排程器提供。它使用一個硬體計時器和一組複雜規則，在不同的執行緒之間快速切換。

微軟宣稱 Windows 95 的時間切片（timeslice）是 20 毫秒（milliseconts）。也就是說如果不考慮其他因素（例如執行緒優先權），每一個執行緒執行 20 毫秒，然後換別人執行。

和行程一樣，執行緒是以一塊從 KERNEL32 共享記憶體中配置而來的記憶體表現出來。這塊記憶體稱爲 thread database 或 TDB，持有所有必要資訊讓 KERNEL32 用來維護一個執行緒。thread database 是個 KERNEL32 物件，其第一個 DWORD 值爲 6 (K32OBJ_THREAD)。

另一個與執行緒有關的資料結構，名爲 THCB（Thread Control Block）。在 Windows 95，執行緒呈現 ring0 和 ring3 兩份資料結構。ring0 碼如 VMM VXD 者經由 THCB 來處理執行緒。ring3 碼如 KERNEL32 者經由 thread database 來處理執行緒。



與執行緒有關的若干東西

執行緒本身擁有若干東西。第一樣東西是 **register set**。當執行緒正在執行，其 register set 內容被放到 CPU 暫存器中。當執行緒不在執行狀態，它的暫存器必須儲存在記憶體某處。每個執行緒有一個指標指向該處。

與執行緒有關係的另一樣東西是行程 **process**。行程的每一個執行緒分享行程的每一樣東西，例如行程擁有 memory context 和一個 private address space，其轄下所有執行緒都在相同的位址空間中執行。行程有個 handle table，轄下所有執行緒也共享相同的這些 handles。

執行緒還擁有許多其他東西。每個執行緒有一個專屬 **stack**，一個專屬的 **window msg queue**，一個專屬的 Thread Local Storage（**TLS**）以及一個專屬的 **structure exception handling chain**。此外，執行緒在執行過程中可能會索求或釋放同步控制物件的擁有權 **ownership of the various synchronization object**。

Thread Handle, Thread ID
GetThreadHandle, GetCurrentThreadId

thread handle 對應用程式而言無用。thread handle 並不是一個可賴以辨別執行緒的資料。Thread handle 只在自己的 context 中才有作用。

KERNEL32 使用同一個 ObsfucatorDWORD 把 pointer to process database 和 pointer to thread database 轉換爲 IDs。一旦知道 ObsfucatorDWORD 的值，就可以把 process ID 或 thread ID 轉換爲有用的指標了。

Thread Database

Thread Database 是個 K32 物件（K32OBJ_THREAD），從 KERNEL32 共享資料區中配置而來。和 process database 一樣，thread database 並不是直接成爲一個 linked-list 形式。thread database 的格式如下：

節錄

00h DWORD Type
此欄爲 6，表示 K32OBJ_THREAD 物件。

04h DWORD cReference
此欄內含執行緒的參用次數。

08h PPROCESS_DATABASE **pProcess**
指標，指向執行緒所屬的行程。

0Ch DWORD someEvent
指標，指向 K32OBJ_EVENT 物件。這個物件正是你呼叫 *WaitForSingleObject* 時給予的 event。

34h PDWORD **pCurrentPriority**
指向一個 DWORD，內含執行緒優先權。該 DWORD 位於 0xC0000000 之上，正是 VxD 的勢力範圍。

38h DWORD **MessageQueue**
low WORD 放置一個 Win16 global heap handle，用作執行緒的 msg queue，存放系統轉來的視窗訊息。

48h DWORD TerminationStatus
此欄位將被 *GetExitCodeThread* 傳回；其值可由 *ExitThread* 或*TerminateThread* 指定。如果執行緒尚在執行，此欄位將爲 0x103（STILL_ACTIVE）。

Thread Database

4Ch WORD **TIBSelector**
此欄位十分重要，內含一個 selector 代表 **current thread**'s **TIB**。TIB 內含極重要信息，像是此執行緒的 head of the exception handler chain。Windows 95 切換執行緒時會令 FS 暫存器存放此值，於是執行緒能夠經由 FS 獲得這些重要資料。

54h DWORD WaitNodeList
如果執行緒正在等待一個（以上）的 event 被激發，這個欄位就會指向 VxD 區域中一個由 event 節點組成的 list。每個節點內含一個指標指向一個 event 物件；另含一個指標指向那正在等待 event 的執行緒。

5Ch DWORD Ring0Thread
這個欄位內含指標，指向執行緒的 ring0 THCB（Thread Control Block）。

60h PTDBX pTDBX
這個欄位內含指標，指向 TDBX 結構。TDBX 是執行緒在 VWIN32.VXD 中的呈現。

7Ch PCONTEXT **ThreadContext**
指標，指向 Intel 的 CONTEXT 結構（定義於 WINNT.H），內放不處於 running state 的執行緒的暫存器值。結構內容可以經由 *GetThreadContext* 和 *SetThreadContext* 讀寫之。

98h DWORD **TLSArray[64]**
這是 64 個 DWORDs 組成的一個陣列。每個 DWORD 是「*TLSGetValue* 函式根據已知之 TLS ID 傳回的值」。第一個 DWORD 是 *TLSGetValue*(0) 的傳回值，第二個 DWORD 是 *TLSGetValue*(1) 的傳回值，依此類推。

Thread Database

198h DWORD **DeltaPriority**
內含「此執行緒之優先權」與其「所屬行程之優先權等級」之間的差異。可能是：

THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_HIGHEST	2

1E8h DWORD pSomeCritSect1
指向一個 K32OBJ_CRITICAL_SECTION 物件，其目的尚未清楚。

1ECh DWORD pWin16Mutex
這個指標指向 KRNL386.EXE 中的 Win16Mutex。

1F0h DWORD pWin32Mutex
這個指標指向 KERNEL32.DLL 中的 Win32Mutex。

TIB (Thread Information Block)

thread database 中有些欄位對於執行中的程式極爲有用，以至於 Win32 架構讓它們可以立刻被取用，不需經過 thread database。這些欄位被放置到一個名爲 Thread Information Block（**TIB**）的結構中。Thread database 的 10h~3Ch 欄位統統被放到 TIB 之中。

應用程式如何取用 TIB？Win32 程式的組語碼呈現出 FS 暫存器被頻繁使用。Win32 底層（包括 Windows NT、Windows 95、Win32s）都奉獻出 FS 暫存器，用以指向目前執行中的執行緒的 TIB。當 Windows 95 切換執行緒，排程器必須更改 FS 暫存器值，讓它內含一個 selector，指向另一個 TIB。

FS 暫存器和 TIB 的主要用途就是增加結構化異常處理串鏈（structured exception handling chain）的項目個數。串鏈的頭放在TIB 的 0 偏移處，所以當你看到組合語言碼使用 FS:[0]，你就知道它正在做某些與結構化異常處理有關的動作。

TIB

Current thread 的 msgQueue 的 handle，此欄位常被 USER.EXE 的視窗系統使用

TIB 結構佈局：

Windows 95 的 TIB 內容如下：

00h	DWORD	pvExcept
04h	DWORD	TopOfStack
08h	DWORD	StackLow
0Ch	WORD	W16TDB
0Eh	WORD	StackSelector16
10h	DWORD	SelmanList
14h	DWORD	UserPointer
18h	PTIB	**pTIB**
1Ch	WORD	TIBFlags
1Eh	WORD	Win16MutexCount
20h	DWORD	DebugContext
24h	PDWORD	**pCurrentPriority**
28h	DWORD	**MessageQueue**
2Ch	DWORD	**pTLSArray**

FS register

Thread database 的 10h~3Ch 欄位統統被放到 TIB 之中。

如果想知道每一個欄位的意義，把其偏移值加 10h，然後看看 thread Database 結構內容，便知道了。其中只有一些欄位對於其他 Win32 平台是共通的。

jjhou.928@gmail.com http://jjhou.csdn.net

00h	DWORD	**Type**
04h	DWORD	**cReference**
08h	PPROCESS_DATABASE	**pProcess**
0Ch	DWORD	someEvent
10h	DWORD	pvExcept
14h	DWORD	TopOfStack
18h	DWORD	StackLow
1Ch	WORD	W16TDB
1Eh	WORD	StackSelector16
20h	DWORD	SelmanList
24h	DWORD	UserPointer
28h	PTIB	**pTIB**
2Ch	WORD	TIBFlags
2Eh	WORD	Win16MutexCount
30h	DWORD	DebugContext
34h	PDWORD	**pCurrentPriority**
38h	DWORD	**MessageQueue**
3Ch	DWORD	**pTLSArray**
40h	PPROCESS_DATABASE	**pProcess2**
44h	DWORD	Flags
48h	DWORD	TerminationStatus
4Ch	WORD	**TIBSelector**
4Eh	WORD	EmulatorSelector
50h	DWORD	cHandles
54h	DWORD	WaitNodeList
58h	DWORD	un4
5Ch	DWORD	Ring0Thread
60	PTDBX	pTDBX
64h	DWORD	StackBase
68h	DWORD	TerminationStack
6Ch	DWORD	EmulatorData
70h	DWORD	GetLastErrorCode
74h	DWORD	DebuggerCB
78h	DWORD	DebuggerThread
7Ch	PCONTEXT	**ThreadContext**
80h	DWORD	Except16List
84h	DWORD	ThunkConnect
88h	DWORD	NegStackBase
8Ch	DWORD	CurrentSS
90h	DWORD	SSTable
94h	DWORD	ThunkSS16
98h	DWORD	**TLSArray[64]**
198h	DWORD	**DeltaPriority**
19Ch	DWORD	un5[7]
1B8h	DWORD	pCreateData16
1BCh	DWORD	APISuspendCount
1C0h	DWORD	un6
1C4h	DWORD	WOWChain
1C8h	WORD	wSSBig
1 Ah	WORD	un7

Thread Priority

Windows 95 的 VMM 核心排程系統並不在乎行程的優先權，它只在乎執行緒的優先權，而不管執行緒屬於哪一個行程。換句話說，行程不真正擁有優先權。

任何時候，擁有最高優先權的執行緒，而它又沒有什麼東西要等待的話，會是即將執行的一個。爲確保平順地讓系統運轉並避免許多問題，執行緒優先權可以動態地被系統改變。例如當一個執行緒正在進行 I/O 動作，其優先權可被暫時提高。

Windows 95 的 VMM 排程器支援 32 種優先權，區分爲四個等級，稱爲優先權級（**priority classes**），每個等級關係到一組優先權。在等級中，優先權可以上下增減 2。也有特殊情況如 THREAD_PRIORITY_LEVEL，可使優先權完全跳出其等級規範之外。作業系統產生一個行程時給予的優先權級預設是 NORMAL_PRIORITY_CLASS。

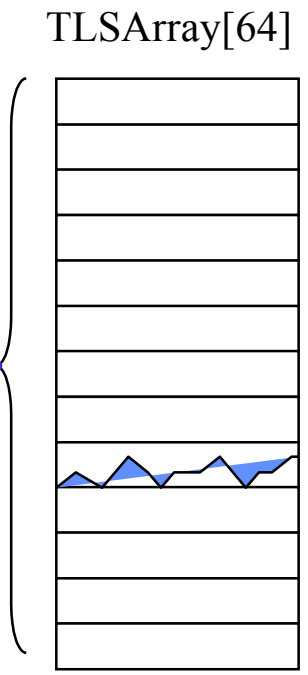
四個優先權級的預設優先權及上下擺盪範圍是：

優先權	預設值	上下擺盪範圍
IDLE_PRIORITY_CLASS	4	2~6
NORMAL_PRIORITY_CLASS	9 或 7 （前景爲 9，背景爲 7）	6~10
HIGH_PRIORITY_CLASS	13	11~15
REALTIME_PRIORITY_CLASS	24	16~31

TLS

陣列中的項目被用於 *TlsSetValue* 函式家族。

```
00h   DWORD   Type
04h   DWORD   cReference
08h   PPROCESS_DATABASE pProcess
...
3Ch   DWORD   pTLSArray
...
98h   DWORD   TLSArray[64]
...
200h  DWORD   LastTlsSetValueEIP[64]
```

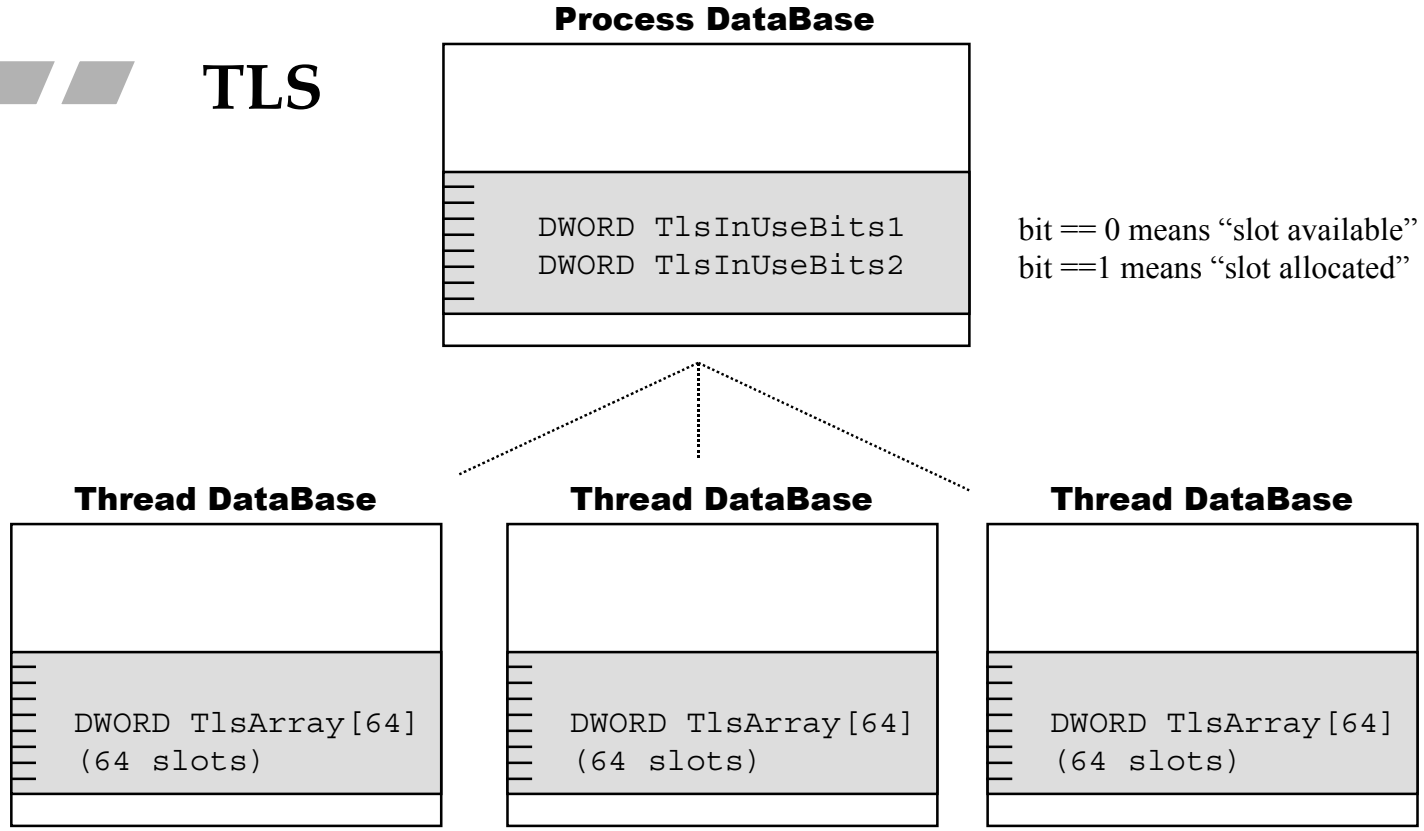


每個 DWORD 將是*TLSGetValue* 函式根據給定之 TLS ID 的傳回值。例如第一個 DWORD 是 *TLSGetValue*(0) 的傳回值，第二個 DWORD 是 *TLSGetValue*(1) 的傳回值。

爲了在每個執行緒中保留一個slot，呼叫 TlsAlloc，傳回一個可被所有執行緒使用的索引值。這個索引值常被儲存在全域變數中。當執行緒要對一個 slot 寫入資料，使用 TlsSetValue，交待一個 TLS 索引和一筆資料。日後當執行緒要取出此值，呼叫 TlsGetValue 再次交待一個 TLS 索引。最後，程式呼叫 TlsFree 並交待一個 TLS 索引，將slot 釋放掉。這麼一來當然也就讓 slot 不再能夠被任何執行緒使用，因爲TLS 索引值在各執行緒之間是共通的。

經由 TLS，程式可以擁有全域變數，但處於「每一執行緒各不相同」的狀態。也就是說行程中的所有執行緒都可以擁有全域變數，但這些變數其實是特定對某個執行緒才有意義。例如你可能有一個多緒程式，每一個執行緒對不同的檔案寫檔（也因此它們使用不同的檔案 handle）。這種情況下把每一個執行緒使用的檔案 handle 儲存在 TLS 會十分方便。當執行緒需要知道所使用的 handle，可從 TLS 獲得。重點在於：執行緒用來取得檔案 handle 的那一段碼在任何情況下都相同，而從 TLS 中取出的 file handle 卻各不相同。有全域變數的便利又分屬各執行緒。

TLS



in DLL

```
WORD TlsIndex;  // global variable

//---when DLL_THREAD_ATTACH ---------------------
if (firsttime)
    TlsIndex = TlsAlloc();
pPerThreadData = malloc(sizeof(PER_THREAD_DATA));
TlsSetValue(TlsIndex, pPerThreadData);

//--- DLL function -----------------------------
ptr = (PER_THREAD_DATA)TlsGetValue(TlsIndex);
...

//----when DLL_THREAD_DETACH --------------------
ptr = (PER_THREAD_DATA)TlsGetValue(TlsIndex);
free(ptr);
```

ref. chap10, Spy DLL p.708, p.730

ToolHelp32

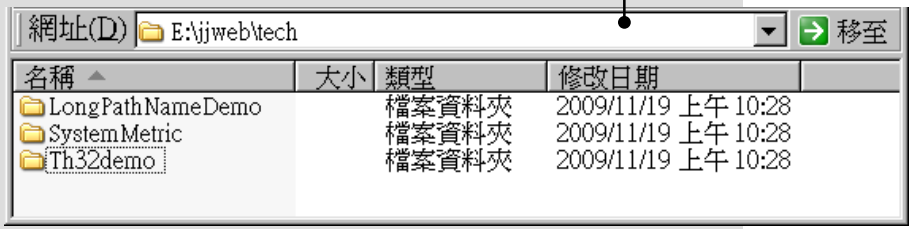
ToolHelp APIs 自 Windows 3.1 開始提供。到了 Windows 95，開始提供 ToolHelp32。經測試，Windows XP 也支援。

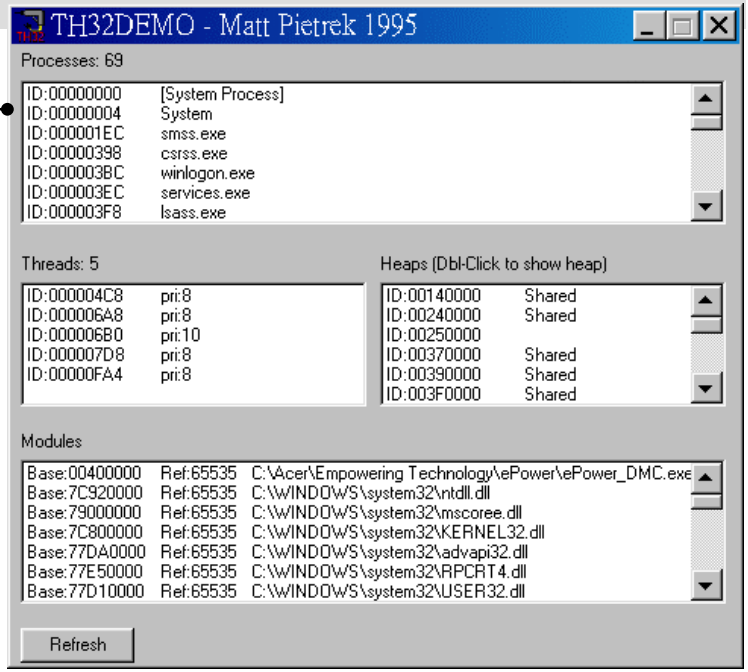
Windows NT 4 不支援。對應的作法是，查詢 register 取得所謂的 performance information。直到 Windows NT 5 才開始支援 ToolHelp32。

ToolHelp32 in Windows XP

Windows XP：c:\windows\system32**toolhelp.dll**		13,888	2007/12/21
	c:\...\vc98\include**tlhelp32.h**	8,669	1998/04/24
	c:\...\vc98\lib**th32.lib**	9,542	1998/05/13

2009/11/19 在 http://www.microsoft.com/msj/1197/nt5dll.aspx 下載 NT51197.exe (39KB)，自解壓得：
其中的 Th32demo 得：





ToolHelp32 in Windows XP

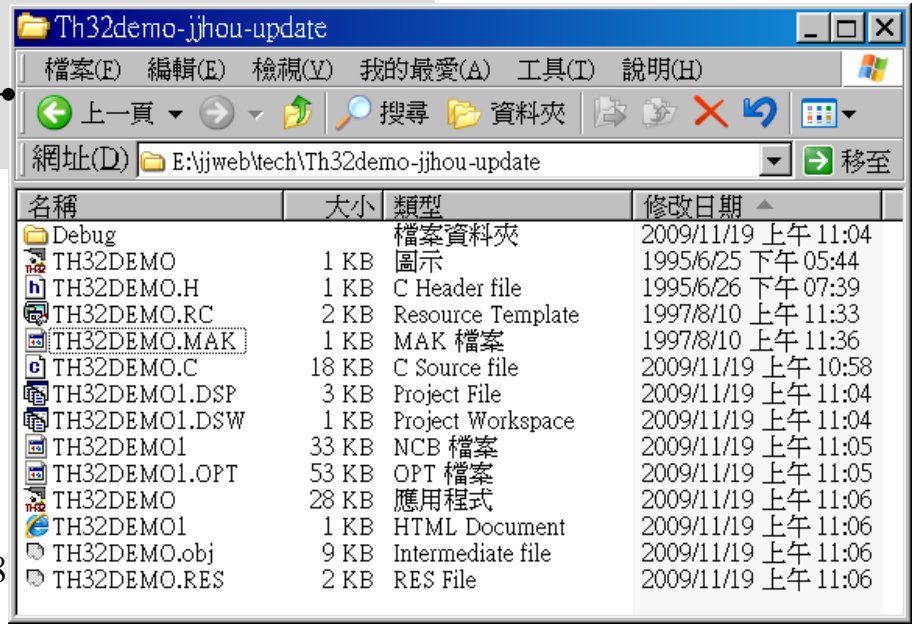
在 DOS box 下執行 nmake th32demo.mak
得到錯誤：

```
E:\temp>nmake th32demo.mak

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        CL /W3 /O1 /DNDEBUG /DWIN32_LEAN_AND_MEAN /c TH32DEMO.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

TH32DEMO.c
TH32DEMO.c(145) : warning C4013: 'AnimateWindow' undefined; assuming extern retu
rning int
TH32DEMO.c(145) : error C2065: 'AW_BLEND' : undeclared identifier
TH32DEMO.c(145) : error C2065: 'AW_ACTIVATE' : undeclared identifier
                                                  ' : return code '0x2'
```

修改：將以下二行 mark 起來：

```
#define _WIN32_WINNT 0x0500     // Needed for call to AnimateWindow

AnimateWindow( hWndDlg, 10, AW_BLEND | AW_ACTIVATE );
```

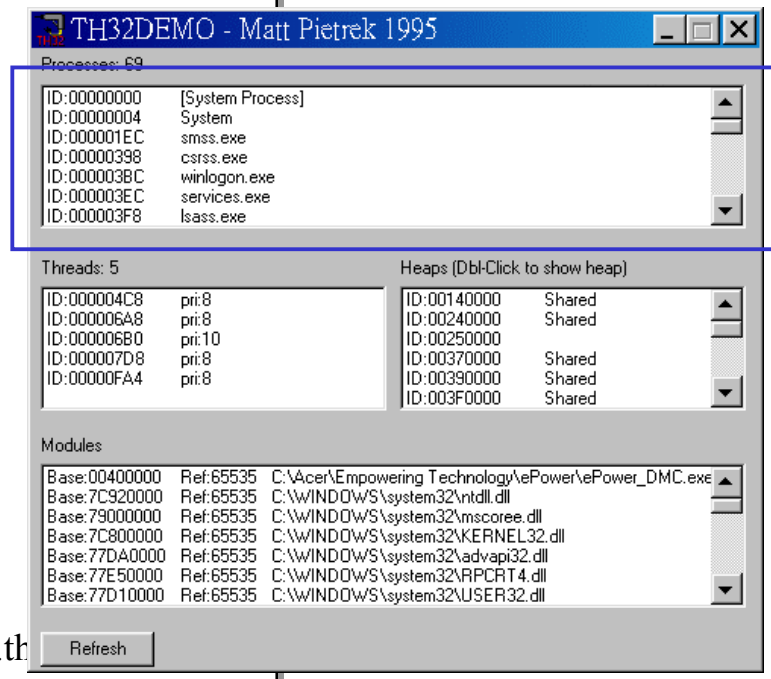
然後在 VC6 中載入 tl32demo.mak，build it，
VC6 告知要產生 dsw, dsp，順其意思，
然後再 build 一遍，便成功。



名稱	大小	類型	修改日期
Debug		檔案資料夾	2009/11/19 上午 11:04
TH32DEMO	1 KB	圖示	1995/6/25 下午 05:44
TH32DEMO.H	1 KB	C Header file	1995/6/26 下午 07:39
TH32DEMO.RC	2 KB	Resource Template	1997/8/10 上午 11:33
TH32DEMO.MAK	1 KB	MAK 檔案	1997/8/10 上午 11:36
TH32DEMO.C	18 KB	C Source file	2009/11/19 上午 10:58
TH32DEMO1.DSP	3 KB	Project File	2009/11/19 上午 11:04
TH32DEMO1.DSW	1 KB	Project Workspace	2009/11/19 上午 11:04
TH32DEMO1	33 KB	NCB 檔案	2009/11/19 上午 11:05
TH32DEMO1.OPT	53 KB	OPT 檔案	2009/11/19 上午 11:05
TH32DEMO	28 KB	應用程式	2009/11/19 上午 11:06
TH32DEMO1	1 KB	HTML Document	2009/11/19 上午 11:06
TH32DEMO.obj	9 KB	Intermediate file	2009/11/19 上午 11:06
TH32DEMO.RES	2 KB	RES File	2009/11/19 上午 11:06

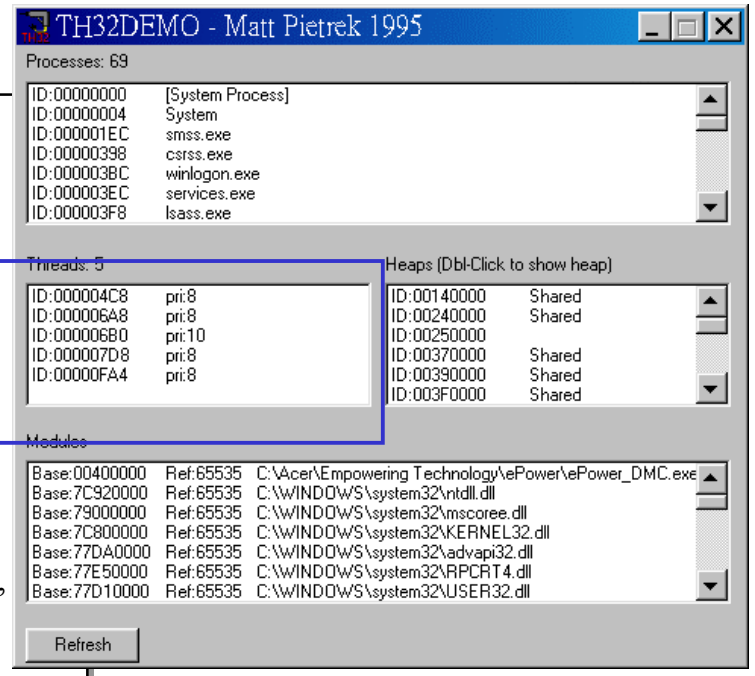
TH32DEMO, about Processes

```
HWND hWndLb;
 HANDLE hSnapshot;
 BOOL fOK;
 PROCESSENTRY32 pe32;
 unsigned cProcesses = 0;

 // create a process snapshot and walk through the process list, adding
 // each process to the top listbox
 hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
 if ( !hSnapshot )
    return;

 pe32.dwSize = sizeof(pe32);     // Gotta initialize dwSize!
 for ( fOK = Process32First( hSnapshot, &pe32 );
       fOK;
       fOK = Process32Next( hSnapshot, &pe32 ) )
 {
    unsigned lbIndex;
    lbIndex = lbprintf(hWndLb,"ID:%08X\t%s",
                    pe32.th32ProcessID, pe32.szExeFile);

    // Use listbox item data to associate the process ID with each
    // line that we add to the listbox
    SendMessage( hWndLb, LB_SETITEMDATA, lbIndex, pe32.th

    cProcesses++;   // Keep track of how many processes we saw
 }
 CloseHandle( hSnapshot );   // Done with this snapshot.  Free it
```



TH32DEMO, about Threads

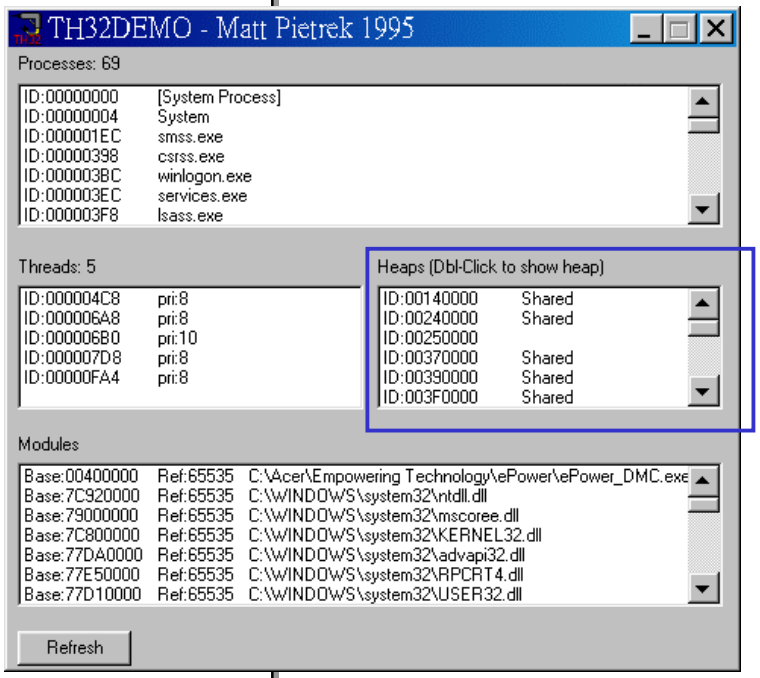
```
void UpdateThreadsListbox( HWND hWndDlg, DWORD processID )
{
   HANDLE hSnapshot;
   BOOL fOK;
   THREADENTRY32 te32;
   char szBuffer[32];
   unsigned cThreads = 0;
…
   // create a threads snapshot and walk through the thread list, adding
   // each thread with the correct process ID to the threads listbox
   hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPTHREAD,
   if ( !hSnapshot )
      return;

   te32.dwSize = sizeof(te32);     // Gotta initialize dwSize!
   for ( fOK = Thread32First( hSnapshot, &te32 );
         fOK;
         fOK = Thread32Next( hSnapshot, &te32 ) )
   {
      if ( te32.th32OwnerProcessID != processID )
         continue;

      lbprintf( hWndThreadsLb,"ID:%08X\tpri:%u",
               te32.th32ThreadID, te32.tpBasePri + te32.tpDeltaPri );

      cThreads++;     // Keep track of how many processes we saw
   }
   CloseHandle( hSnapshot );   // Done with this snapshot.  Free it
…
```



注意：不是「取某個 process 轄下所有 threads，而是取系統中的所有 threads 再比對其 process 是否爲吾人關注的這個 process。

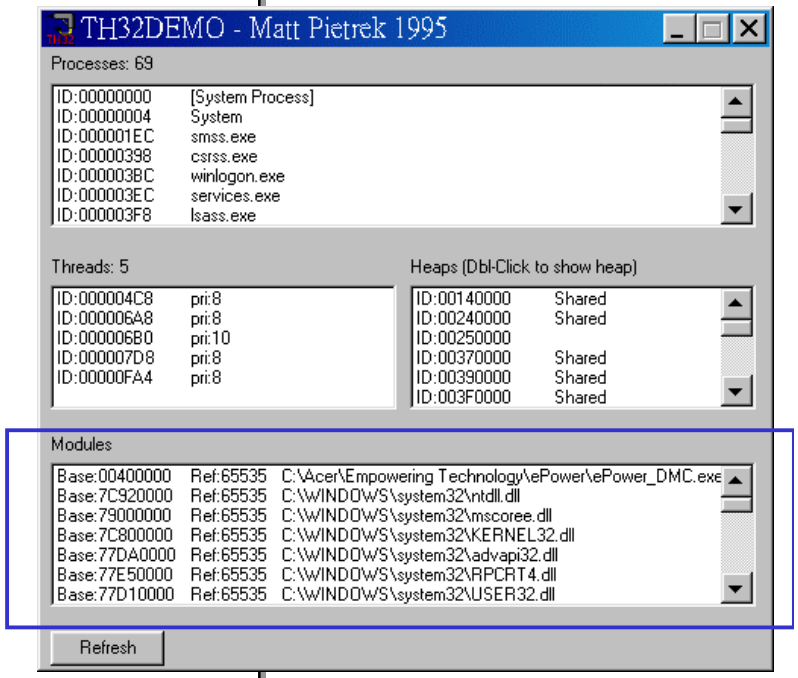
TH32DEMO, about Heap List

```
void UpdateHeapsListbox( HWND hWndDlg, DWORD processID )
{
   HWND hWndHeapsLb;
   HANDLE hSnapshot;
   BOOL fOK;
   HEAPLIST32 hl32;
…
   // create a process heap list snapshot and walk through the list, adding
   // each heap to the heaps listbox
   hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPHEAPLIST, processID );
   if ( !hSnapshot )
      return;

   hl32.dwSize = sizeof(hl32);     // Gotta initialize dwSize!
   for ( fOK = Heap32ListFirst( hSnapshot, &hl32 );
         fOK;
         fOK = Heap32ListNext( hSnapshot, &hl32 ) )
   {
      unsigned lbIndex;
      lbIndex = lbprintf( hWndHeapsLb,"ID:%08X\t%s %s",
                     hl32.th32HeapID,
                     hl32.dwFlags & HF32_DEFAULT ? "default" : "",
                     hl32.dwFlags & HF32_SHARED ? "Shared" : "" );
      …
   }
   CloseHandle( hSnapshot );   // Done with this snapshot.  Free it
}
```



TH32DEMO, about Modules

```
void UpdateModulesListbox( HWND hWndDlg, DWORD processID )
{
   HWND hWndModsLb;
   HANDLE hSnapshot;
   BOOL fOK;
   MODULEENTRY32 me32;
…
   // create a process module snapshot and walk through the list, adding
   // each module to the Modules listbox
   hSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, processID );
   if ( !hSnapshot )
      return;

   me32.dwSize = sizeof(me32);     // Gotta initialize dwSize!
   for ( fOK = Module32First( hSnapshot, &me32 );
         fOK;
         fOK = Module32Next( hSnapshot, &me32 ) )
   {
      if ( 0 == me32.hModule )
         break;

      lbprintf( hWndModsLb,"Base:%08X\tRef:%02u\t%s",
               me32.hModule, me32.ProccntUsage, me32.szExePath );
   }

   CloseHandle( hSnapshot );   // Done with this snapshot.  Free it
}
```



TH32DEMO, about Heap

```
void Handle_HeapWalkDlg_WM_INITDIALOG(  HWND hWndDlg,
                                        WPARAM wParam, LPARAM lParam )
{
   BOOL fOK;
   PPROCESSHEAP_IDS ph_ids = (PPROCESSHEAP_IDS)lParam;
   HEAPENTRY32 he32;
   HWND hWndLb;
   DWORD tabStops = 20;
…
   // Walk through all the blocks in the specified heap, addin
   // each block's information to the top listbox
   he32.dwSize = sizeof(he32);     // Gotta initialize dwSize!
   for (fOK=Heap32First(&he32, ph_ids->processID, ph_ids->heapID);
        fOK;
        fOK = Heap32Next(&he32) )
   {
      unsigned lbIndex;
      lbIndex = lbprintf( hWndLb, "Addr:%08X\tHndl:%08X\t%04u bytes %s"
                     he32.dwAddress, he32.hHandle, he32.dwBlockSize,
                     he32.dwFlags & LF32_FREE ? "Free":"");

      // If it's an in-use block, allocate memory and store off info
      // about the block so that we can display it's contents later.
      if ( 0 == (he32.dwFlags & LF32_FREE) )
      {
         PIN_USE_HEAP_BLOCK pBlock;
         if ( pBlock = malloc(sizeof(IN_USE_HEAP_BLOCK)) )
         {
            pBlock->processID = ph_ids->processID;
            pBlock->address = he32.dwAddress;
            pBlock->dwSize = he32.dwBlockSize;
            SendMessage( hWndLb, LB  SETITEMDATA, lbIndex, (LPARAM)pBlock );
```

```
Toolhelp32ReadProcessMemory(  pBlock->processID,
                              (PVOID)pBlock->address,
                              data,
                              cbMax,
                              &cbMax ) )
```