- Introduction : ErrorHandling ( What? 왜 ErrorHandling이 필요한가? )

- Unicode ( DBCS에서 Unicode 의 전환 , 사용 방법 )

- PE : Windows 기반의 Exe, Dll 파일 포멧

- Object 🡺 Kernel Object( 다른 오브젝트와의 차별성, 특징 )

- Process ( 생성 , 소멸, 기타 관련 함수)

- Thread( 생성, 소멸, 기타 관련 함수)

Thead synchronization ( 동기화 : CriticalSection / Mutext / Shemapore / Event )

- Memory( 가상 메모리, 가상 메모리 주소, 다양한 형태의 메모리 사용 기법 )

MMF(Memory Mapping File) : 파일의 가상 메모리 화

IPC(SendMessage, WM\_COPYDATA, Pipe) : 프로세스간 데이터 교환

- DLL

DLL Injection( DLL을 이용한 Hooking 기법)

\_ SEH / STH ( 구조화된 예외처리 기법)

일반적으로 하나의 함수를 어떻게 컴파일 되도록 하느냐(어떤 기계어 코드로 바꿀것이냐)에 따라 5가지의 호출 규약이 있다.

**\_\_cdecl**

일반적으로 C/C++의 함수( printf, strcpy등.)이 사용하는 방식이다.

파라미터는 Stack을 통해서 전달된다. 스택에서 파라미터를 제거하는 것은 호출자 함수의 책임이다.

**\_\_stdcall**

윈도우 프로그램에서 사용하는 대다수의 API가 사용하는 방식이다.

파라미터는 Stack을 통해서 전달된다. 스택에서 파라미터를 제거하는 것은 호출된 함수의 책임이다(\_\_cdecl 과의 차이점). \_\_cdecl방식보다는 메모리의 사용이 효율적이다. Pascal 언어가 사용하던 방식이라서 Pascal 호출 규약이라고도 불린다.

**\_\_fastcall**

함수의 호출을 보다 효율적(빠르게)하기 위해 처음 2개의 파라미터는 레지스터를 사용해서 전달하고 나머지 파라미터는 Stack을 사용한다.

다른 CPU와의 이식성이 없기 때문에 거의 사용하지는 않는다.

**this**

C++의 멤버함수를 컴파일할때 사용된다. C++의 멤버함수를 호출할때는 객체의 주소(this)가 함수로 전달되어야 하는데, 이때 ECX레지스터를 사용해서 전달한다.

**naked**

컴파일러 기본적으로 만들어주는 Prolog와 Epilog를 변경할때 사용된다. 다른 CPU와의 이식성이 없기 때문에 일반 응용프로그램에서는 거의 사용되지 않는다. 주로 디바이스 드라이버를 만들때 사용한다.

대다수의 윈도우 프로그램에서 사용하는 함수는 \_\_stdcall 방식을 사용한다. 또한 WINAPI, CALLBACK, APIENTRY, PASCAL등의 매크로는 \_\_stdcall방식을 가르키는 매크로 이다.

void **WINAPI** SomeFunck(); // WINAPI -> #define WINAPI \_\_stdcall

일반적으로 VC++를 사용하면 아래의 2가지 방식의 실행 파일을 만들수 있다.

**CUI ( Console User Interface ) – Win32 Console Application**

**- Entry Point 수행전에 Console 창이 실행된다.**

**- Entry Point : main(), wmain()**

**- Link Option : /subsystem:console**

**GUI ( Graphic User Interface ) – Win32 Application**

**- Console 창이 생성되지 않는다. 대부분 사용자가 윈도우를 만들어야 한다.**

**- Entry Point : WinMain (), wWinMain()**

**- Link Option : /subsystem:windows**

VC++에서 처음에 선택하는 Project의 Type에 따라(그림 참고) 결정된다.

**하지만 결국 이 2가지 type의 차이는 Link Option 에 따라 달라 지게된다.**

**/subsystem:windows**

라는 Option을 주면 Link가 GUI형의 Exe 파일을 생성한다.

Project->Setting 메뉴를 선택한후 Project Setting Dialog 의 Link Tab에서 Project Option안의 /subsystem을 변경함으로써 쉽게 exe의 type을 변경할 수 있다(그림 참고).

또한 /subsystem Option을 완전히 제거해 버리면 Link는 사용자가 main, WinMain중 무엇을 만들었는지를 보고 실행 파일의 type을 결정한다.

**#pragma 지시어를 사용해서 Linker에게 지시하면 프로젝트 Type에 상관없이 Linker에게 Exe의 Type을 지정할수 있다**

**#pragma comment ( linker, “/subsystem:windows”)**

**#pragma comment ( linker, “/subsystem:console”)**

**#pragma comment ( linker, “/subsystem:windows,5”) // window 2000의 추가 기능을 사용할경우.**

**ErrorHandling**

|  |  |
| --- | --- |
| 윈도우 프로그램을 작성할 때 많은 함수를 호출한다. 임의의 API 함수를 호출할 때 해당 함수가 실패할 경우가 있다. 다음의 표는 함수의 리턴type과 실패했을 때의 값을 설명한다. | |
| **Return Value Data Type** | **Value to Indicate Failure** |
| VOID | 이 함수는 실패하지 않을 것이다. 일부 함수만 VOID를 리턴한다. |
| BOOL | 함수가 실패 하면 FALSE( 0 )을 리턴한다. |
| HANDLE | NULL, 또는 INVALID\_HANDLE\_VALUE(-1)을 리턴한다. |
| PVOID | 실패할경우 0을 리턴한다. |
| LONG/DWORD | 대개 0, -1 을 리턴하면 실패이다. |
|  | |
| * **Last Error Code** | |
| Windows는 스레드를 하나 생성할때 마다 내부적으로 TEB(Thread Environment Block, TIB라고도 불린다.)라는 data structure를 한 개 씩 생성하는 데 이 안에 Last Error Value를 보관한다. 즉, Error Code 값은 스레드당 1개씩 존재한다. | |
| 어떤 함수의 호출이 실패 했을 때 아래 함수를 사용하면 Error Code값을 얻을 수 있다. | |
| **DWORD GetLastError();** | |
| 또한 다음 함수를 사용하면 Error Code값을 변경할 수도 있다. | |
| **VOID SetLastError(DWORD *dwErrCode*** **);** | |

|  |
| --- |
| * **FormatMessage** |
| GetLastError()가 리턴한 에러 값은 FormatMessage() 함수를 사용해서 문자열로 변환할 수 있다.  또한, VC++ 6.0 의 메뉴중 Tools -> Error Lookup을 선택한 후 Value에 값을 넣고 Lookup 버튼을 누르면 에러에 설명을 얻을 수 있다.  아래 예제는 실행해 보자. |
| #include <windows.h>  #include <TCHAR.h>  VOID ReportError( LPTSTR userMsg )  {  DWORD dwLastError = GetLastError();  LPTSTR lpszMsg;  DWORD dwLen = FormatMessage( FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |  FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, dwLastError,  MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),  (LPTSTR)&lpszMsg, 0, NULL);  MessageBox( 0, lpszMsg, userMsg, MB\_OK | MB\_ICONERROR);  LocalFree( lpszMsg );  }  int WINAPI \_tWinMain( HINSTANCE hInstance, HINSTANCE, LPTSTR lpsz, int nCmdShow )  {  HWND hwnd = CreateWindowEx(0,0,0,0,0,0,0,0,0,0,0,0);  if ( hwnd == 0)  ReportError( \_T("윈도우 생성실패") );  return 0;  } |

에러 코드의 값과 설명은 프로그램의 디버깅 시에도 얻을 수 있다. 디버깅 중에 Watch 창에

**@err**

라고 입력하면 현재 스레드의 Last Error Value를 얻을 수 있다. 또한.

**@err,hr**

라고 입력하면 Last Error 값 뿐아니라 FormatMessage를 수행한 결과의 문자열도 바로 확인할 수 있다.

또한,

**@TIB**

라고 입력하면 Thread Environment Block의 주소를 얻을수 도 있다.

|  |
| --- |
| 함수호출이 실패 했을 때만 GetLastError()를 호출하는 것은 아니다. 때로는 함수 호출이 성공을 하고 원하는 결과를 얻었을때는 GetLastError()을 호출하곤한다. |
| Event 객체라는 것은 System에서 실행중인 모든 프로세스 안에서 동일한 이름으로 1개 이상 만들 수는 없다. 즉, A가 “countEvent”라는 이름으로 event객체를 만들었다고 하자. 그리고 B가 역시 동일한 이름으로 CreateEvent()함수를 호출할 경우 동일한 이름의 event 객체가 있으므로 새롭게 만들어지지는 않는다. 대신, 이미 존재 하던 event 객체의 사용횟수가 2로 증가 하고 핸들이 리턴 된다. |
| 이때, B의 입장에서는 자신이 얻은 핸들이 새롭게 만들어진 핸들인지 아니면 이미 존재하는 객체의 핸들인지를 확인하기 위해서 GetLastError()를 호출하면 된다.  **즉, 함수 호출이 성공 했을 때, 성공의 이유를 알기 위해서도 GetLastError()함수를 호출한다.** |
| * **오직 한 개의 인스턴스 만을 생성할 수 있는 프로그램** |
| MSN 메신저나 미디어 플레이어 같은 프로그램은 2개의 실행 인스턴스를 만들수가 없다. 오직 1개만 실행할 수 있다.  아래 프로그램은 event 객체를 사용해서 이와 같이 1개의 인스턴스 만을 만들 수 있는 방법을 보여준다. |
| #include <stdio.h> #include <conio.h> #include <windows.h>  int main() {  DWORD error = 0;  HANDLE hEvent = CreateEvent( 0, 0, 0, "countEvent");  if ( hEvent == 0 ) {  printf("Fail Create Event..!! \n");  return 0;  }  error = GetLastError();    if ( error == ERROR\_ALREADY\_EXISTS ) {  printf("이미 프로그램이 실행중입니다.\n");  return 0;  }  // 프로그램... 수행...  printf("현재 프로그램이 실행중입니다.\n");  printf("종료 하려면 아무키나 누르세요\n");  getch();  } |

|  |
| --- |
| * **DBCS ( Double Byte Character Set )** |
| 먼저 아래 코드를 보자. (A) 의 결과는 얼마가 나올까 ? |
| #include <stdio.h>  void main() {     char msg[] = "ABCD가나다라";         printf("%d\n", sizeof( msg ) ); // ( A ) } |
| 답은 13이다.  현재 C에서는 영어 1 문자를 보관하기 위해서는 1Byte를 사용하고 한글 1문자를 보관하기 위해서는 2Byte를 사용한다. |
| 이와 같이 어떤 경우는 1Byte를 어떤 경우는 2Byte를 사용해서 문자를 나타내는 것을 **DBCS(Double Byte Character Set)**이라고 한다. 하지만 DBCS의 단점은 Pointer연산을 사용하기가 힘들다는 점이다. 영문자일 경우는 다음 글자를 얻기 위해 시작 주소에 1을 더하면 되지만 한글일 경우는 2를 더해야 한다. 다음 코드는 다음 글자를 얻기 위해 Pointer에 1을 더하는 코드이다.  오른쪽은 실행결과이다. |

|  |  |
| --- | --- |
| #include <stdio.h>  void main() {  char msg[] = "ABCD가나다라";  char\* pmsg = msg;  while ( \*pmsg )  {  printf("%s\n", pmsg );  pmsg = pmsg + 1;  } } | ABCD가나다라 BCD가나다라 CD가나다라 D가나다라 가나다라 〕ご帽? 나다라 ご帽? 다라 帽? 라 ? |
| 오른쪽 결과에서 보듯 한글의 경우는 2byte를 더해야 하는 데 1Byte씩 증가 하므로 문자가 깨어지는 것을 볼수 있다.  그래서 DBCS문자열들은 포인터 연산 대신에 아래의 함수들을 사용해야 한다.   * **windows 에서 DBCS를 다루기 위해 제공되는 함수들** | |
| * BOOL IsDBCSLeadByte( BYTE TestChar ): TestChar이 나타내는 값이 2Byte문자의 첫 번째 바이트라면 TRUE이다. * LPTSTR CharNext( LPCTSTR lpsz ) : lpsz가 가르키는 문자열에서 다음 문자의 Pointer를 얻는다. * LPTSTR CharPrev( LPCTSTR lpszStart, LPCTSTR lpszCurrent ) : lpszStart에서 시작하는 문자열에서 lpszCurrent가르키는 문자의 바로 앞 문자를 구한다. | |
| ==> 즉 한글일 경우 2byte 단위로 이동 영문일 경우 1byte 단위로 이동 | |
| * **아래 프로그램의 위의 함수 중 IsDBCSLeadByte() 함수를 사용한 예이다.(앞장의 예제와 비교)** | |
| #include <stdio.h> #include <windows.h>  void main() {  char msg[] = "ABCD가나다라";  char\* pmsg = msg;  while ( \*pmsg )  {  printf("%s\n", pmsg);  if ( IsDBCSLeadByte(\*pmsg) )   pmsg += 2;  else  pmsg += 1;  } }  // CharNext() 함수를 사용해서 만들어 보자.  void main()  {  char msg[] = "ABCD가나다라";  char\* pmsg = msg;  while ( \*pmsg )  {  printf("%s\n", pmsg);  pmsg = CharNext( pmsg );  }  } | ABCD가나다라 BCD가나다라 CD가나다라 D가나다라 가나다라 나다라 다라 라 |

|  |
| --- |
| * **이번에는 DBCS 문자열의 개수를 세는 프로그램을 만들어보자.**   **(한글, 영문 모두 1글자로 처리하는 함수)** |
| #include <stdio.h> #include <windows.h>  int DBCSLen(const char\* p) {  int count = 0;  while ( \*p )  {  ++count;          if ( IsDBCSLeadByte( \*p ) )  p += 2;  else  ++p;  }  return count; }  void main() {  char msg[] = "ABCD가나다라ABCD";  printf("%d\n", strlen( msg ) );  printf("%d\n", DBCSLen( msg ) ); } |
|  |
| **DBCS 의 문제점**  **한글 같은 경우 일부만 읽을 경우 깨진다.**  **문자 사이 이동시 추가적인 함수가 필요하다.**  **1바이트 or 2바이트 구분없이 사용할 수 있는 방법은 없을까?**  **[ 연습 문제 ]** |
| 1. DBCS 문자열을 뒤집을 수 있는 DBCSReverse(char\* p) 함수를 만드시오. |

세상의 다양한 문자들도 나타낼 수 있게 하고 Pointer연산도 쉽게 처리할 수 있도록 1문자를 위해 2Byte를 사용할 수 있다. 이를 WideCharacter라고 한다. Wide Character를 흔히 Unicode라고 한다. 아래 코드를 보자.

|  |
| --- |
| #include <stdio.h> void main() {  wchar\_t msg[] = L”ABCD가나다라”;  printf(“%d\n”, sizeof(msg)); } |

C에서 wchar\_t 는 1문자를 담을 때 사용하는 2byte type 이다.

(현재 VC++6.0에서 wchar\_t 는 unsigned short의 typedef이다.)

'L'지시어는 컴파일러에게 문자열을 컴파일할 때 2Byte씩 처리해달라는 지시어이다.

아래코드의 결과는 얼마 일까?

|  |
| --- |
| #include <stdio.h> #include <string.h>  void main() {     wchar\_t msg[] = L"ABCD가나다라";     printf("%d\n", strlen( (char\*)msg ) ); // ? } |

왜 이런 결과가 나올까? 아래 메모리 그림을 보자.



결국 Unicode에서는 'A'라는 문자를 2Byte로 표현하기 위해 첫 번째 바이트는 'A'의 아스키 코드인 41를 그리고 2번째 바이트는 '\0'으로 채운다. 이때 strlen()함수는 파라미터로 전달된 포인터부터 시작해서 '\0'을 찾을 때 까지의 갯수를 파악하므로 1이 나오게 된다.

이 때문에 Unicode전용 문자열 함수('/0/0'을 문자열 끝으로 판단 하는) 가 필요하다. C에서 ANSI 문자열 함수는 'str' 이라는 접두어를 갖는다. 또한 해당함수의 unicode버전인 'wcs'를 접두어로 갖는 함수도 지원한다.

**size\_t** **strlen(** **const** **char** **\****string* **);** DBCS문자열의 길이를 구한다.

**size\_t** **wcslen(** **const** **wchar\_t** **\****string* **);** strlen의 Unicode 버전

ANSI C의 표준 문자열함수는 접두어 **strxxx**을 지닌다. 동일한 기능을 하는 Unicode문자열 함수는 **wcsxxx(wide character string)의 형태를 지닌다.** 위의 코드가 올바르게 실행되게 하려면 strlen()대신 wcslen을 사용한다.(위 코드 참고).

* 위의 소스 참조

하나의 소스코드를 사용해서 DBCS용 버전의 Code 와 Unicode용 버전의 Code를 모두 만들고 싶다면 어떻게 할까 ? char, wchar\_t를 바로 사용할 수는 없을 것이다. 그래서 마이크로 소프트에서는 몇 가지 유용한 매크로를 제공한다.위 코드는 TCHAR.h 라는 헤더에 포함되어 있는 내용이다.

결국 TCHAR 이라는 type은 \_UNICODE라는 symbol이 정의 되어 있느냐에 따라 char, 혹은 wchar\_t로 바뀌게 된다. 또한 \_tcslen도 strlen혹은 wcslen으로 바뀌게 된다. 아래 코드를 보자.

|  |
| --- |
| #include <stdio.h> #define \_UNICODE  // 이 부분을 주석처리 할 때와 그렇지 않을 때를 비교해 보자. #include <TCHAR.h>  void main() {     TCHAR buf[] = \_TEXT("ABCD가나다라");  #ifdef \_UNICODE     printf("현재 Unicode입니다.\n"); #else     printf("현재 Unicode가 아닙니다.\n"); #endif      printf("메모리 크기  : %d Byte\n", sizeof( buf ));     printf("문자열의 길이 : %d\n",  \_tcslen( buf ) ); } |

결국 TCHAR Type을 사용하고 모든 문자열은 \_TEXT()또는 \_T()로 묶어서 사용하고, 문자열 함수는 \_tcsxxx()를 사용하면 \_UNICODE의 정의 여부에 따라 자동으로 UNICODE혹은 UNICODE가 아닌 코드로 변경할 수 있게 된다. 보통의 경우는 **\_UNICODE(C헤더 파일에서사용), UNICODE(윈도우 헤더파일에서 사용)**를 모두 정의 해주어야 한다.

MessageBox 라는 함수를 생각해 보자. 이 함수의 2번째 3번째 파라미터는 문자열이다. 그렇다면 Unicode가 넘어갈 수도 있고 Unicode가 아닌 문자열이 넘어 갈수 도 있을 것이다. C++이라면 동일한 이름의 함수가 2개 있을 수 있지만 API는 C로 만들어 져 있다.결국 우리가 아는 MessageBox라는 함수는 매크로 일뿐, 진짜 함수는 2가지의 버전이 있다.

결국 User32.dll 안에는 MessageBoxA, MessageBoxW라는 2가지의 함수가 있다.MessageBox는 WinUser.h라는 헤더에 있는 매크로 일뿐이다.Visual C++과 함께 제공되는 Depends.exe를 사용하면 DLL을 export하는 모든 symbol을 조사해 볼 수 있다.

위의 그림은 Depends.exe에서 user32.dll을 열어 보았을 때의 모습이다.(WIN XP 환경)

위 그림에서 끝이 A, W로 끝나는 함수는 파라미터로 문자열을 받는 함수들의 Unicode버전과 ANSI 표준 문자열 버전의 함수이다.

위의 2가지 함수를 사용하면 Ansi 표준 문자열과 Unicode사이의 변환을 수행할 수 있다.

아래 프로그램은 Unicode문자를 ANSI C표준 문자열로 변경 후 printf를 사용하여 출력하는 함수이다.

|  |
| --- |
| #include <stdio.h> #include <memory.h> #include <windows.h>  void uprint( const wchar\_t\* msg) {     int size = wcslen(msg)+1;     char\* temp = (char\*)malloc( size \* 2 );      WideCharToMultiByte( CP\_ACP, 0, msg, wcslen(msg)+1, temp, size \* 2, 0,0);    printf("%s\n", temp);     free(temp); }  void main() {     wchar\_t msg[] = L"ABCD가나다라";     uprint( msg ); } |

운영 체제에 따른 지원 현황

|  |
| --- |
| win 3.1 지원 안함  win95 win98 unicode 일부 지원 A버전 지원됨. W버전은 선언만...정의 안됨.  wind2000 이상 unicode 지원 A버전, W버전 둘 다 지원  Wind CE unicode만 지원 |

**타입 정의**

|  |
| --- |
| PSTR : = char\*  LPSTR : = L자가 붙은 경우 3.1에서 썼음 WIN32에서는 똑같다. = char\*  PWSTR : wchar\_t\*  PCSTR : = const char\*  // 두가지를 다 지원한다. 일반적으로 이 두 가지를 가장 많이 씀  TCHAR wchar\_t char 문자  LPTSTR wchar\_t\* char\* 문자열  PTSTR : TCHAR\*-> (char\*, wchar\_t\*)  LPCTSTR const wchar\_t\* const char\* 문자열 상수  PCTSTR : const TCHAR\* |

|  |  |
| --- | --- |
| /\* StringReverse.h \*/  #ifdef DLL\_SOURCE  #define DLLAPI \_\_declspec(dllexport)  #else  #define DLLAPI \_\_declspec(dllimport)  #endif  #ifdef \_\_cplusplus  extern "C" {  #endif  DLLAPI bool StringReverseW( wchar\_t\* );  DLLAPI bool StringReverseA( char\* );  #ifdef \_\_cplusplus  }  #endif  /\* UNICODE 매크로의 define 여부에 따라 함수가 결정된다. \*/  #ifdef UNICODE  #define StringReverse StringReverseW  #else  #define StringReverse StringReverseA  #endif // UNICODE | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  /\* StringReverseA : 실제로 문자열을 뒤집는 대신에  /\* ANSI 문자열을 Unicode로 변환한후  /\* ReverseStringW()로 전달해서  /\* 문자열을 뒤집은 후에 다시 ANSI  /\* 문자열로 변경한다. \*/  bool StringReverseA( char\* pStr )  {  int nLen = 0;  wchar\_t\* pWstr = 0;  bool fOk = FALSE;  // ANSI 문자열을 Unicode로 변경하기위해 필요한  // Unicode buffer의 크기를 계산  nLen = MultiByteToWideChar( CP\_ACP, 0, pStr, -1, NULL, 0 );  // ANSI 문자열을 Unicode로 변경하기 위해서 Unicode 문자  // Buffer 할당  pWstr = (wchar\_t\*)malloc( sizeof( wchar\_t) \* nLen);  if ( pWstr == NULL )  return fOk;  // ANSI -> Unicode  MultiByteToWideChar( CP\_ACP, 0, pStr, -1, pWstr, nLen );  // Unicode 버전의 함수를 호출한다.  fOk = StringReverseW( pWstr );  if ( fOk )  {  // 뒤집어진 Unicode문자열을 다시 ANSI 문자열로 변환  WideCharToMultiByte(CP\_ACP, 0, pWstr, -1, pStr,  strlen( pStr ), NULL, NULL );  }  free( pWstr );  return fOk;  } |
| /\* StringReverse.cpp \*/  #define DLL\_SOURCE  #include <windows.h>  #include "StringReverse.h"  /\* StringReverseW : Unicode 문자열을 뒤집는다. \*/  bool StringReverseW( wchar\_t \*pWstr )  {  wchar\_t \*pEnd = pWstr + wcslen( pWstr ) - 1;  wchar\_t temp;    while( pWstr < pEnd )  {  temp = \*pWstr;  \*pWstr = \*pEnd;  \*pEnd = temp;  pWstr++;  pEnd--;  }  return TRUE;  } |

윈도우즈 운영체제에서 하나의 실행파일(.exe)이나 DLL은 단순한 기계어코드의 덩어리는 아니다. **Portable Executable File Format** 이라고 알려진 형태의 파일이다.실행파일(PE File)은 크게 헤더 부분과 Section부분으로 나누어진다.(DLL도 마찬가지 이다.) 위 그림을 보자.

헤더는 다음과 같은 요소로 구성되어 있다.

|  |  |  |
| --- | --- | --- |
| IMAGE\_DOS\_HEADER MS DOS Stub | MS-DOS 시절 사용하던 헤더로 현재는 사용되지 않는다. | |
| IMAGE\_NT\_HEADER | Signature | 항상 'PE'로 되어 있다. |
| IMAGE\_FILE\_HEADER |  |
| IMAGE\_OPTIONAL\_HEADER | 가장 중요한 헤더 이다.  Entry Point의 주소, Image의 가상주소(Image base), stack, heap의 크기등의 정보를 가지고 있다. |
| IMAGE\_SECTION\_HEADER | 섹션의 갯수만큼 존재한다. 각 Section의 정보를 담고 있다. | |

헤더 다음은 각 section이라는 요소로 구분되어 있다.

우리가 만든 c 함수들의 기계어 코드는 .text라는 section에 놓인다. 또한 메뉴나 다이얼로그, Icon등은 .rsrc Section에 놓인다.또한 exe가 사용하는 각종 DLL에 대한 정보는 .idata section에 놓인다.

대부분의 표준 섹션들은 '.' 으로 시작된다. 위 표는 몇가지 대표적인 section 들이다.

PE(Portable Executable) 이란 **플랫폼(CPU)에 상관없이 Wn32 운영체제가 돌아 가는 시스템이면 어디든지 실행 가능하다는 의미에서** 붙여진 이름이다.

즉, Intel CPU기반의 윈도우 운영체제가 탑재된 시스템에서 돌아가는 PE 프로그램은 ALPHA CPU기반의 윈도우 운영체제가 탑재된 시스템에서도 실행이 가능하다는 것이다.

요즈음의 컴퓨터는 대략 512M정도의 물리메모리를 사용한다. 하지만 이 메모리양은 많은 프로그램을 동시에 실행하기 에는 그리 많은 메모리가 아니다. 그렇다고 값이 비싼 물리 메모리를 무조건 많이 사용할 수 도 없다.그래서 대부분의 운영체제는 하드 디스크의 특정 공간(PageFile.sys)을 마치 물리 메모리 인것처럼 사용한다. 이를 가상메모리라고 한다.물리메모리가 부족할 때 물리 메모리의 하드디스크로 잠시 저장하고 확보된 물리메로리를 계속 사용할 수 있게되는 것이다.(이를 Swapping이라고 한다.)

제어판에서 시스템을 선택후 고급탭에서 성능항목의 설정 버튼을 누른후 나타나는 윈도우에서 고급탭에 있는 가상메모리 변경을 선택하면 위의  그림 처럼 가상 메모리를 변경할 수 있다

하나의 프로그램이 실행될 때 운영체제는 해당하는 프로그램이 사용할 수 있는 4G의 가상 주소 공간을 만든다.아래 그림에서 프로그램 A가 실행 될 때 A는 4G의 자신만의 주소 공간을 갖게 된다. 이 주소는 Page Table과 Page directory 라는 것을 사용해서 실제 물리 주소로 변경된다.

위의 그림에서,

결국 A와 B라는 2개의 프로세스가 실행중인때 A프로세스의 1234'0000이라는 주소는 B프로세스의 1234'000주소와는 전혀 다른 내용을 가르키게 되는 것이다.

그러므로 결국 A가 아무리 잘못된 포인터 연산을 하더라도 B프로세스의 메모리 영역은 안전할 수 있는 것이다.

하나의 프로세스가 사용하는 4G의 주소공간중 상위 2G(0x8000'0000~0xFFFF'FFFF)는 System에의해 사용되어 진다. 주로 이영역에 운영체제 Code와 각종 Device Driver Code가 놓여진다. 또한 이공간은 모든 프로세스들이 공유하는 공간이다.하지만 하위 2G(0x0000'0000 ~ 0x7FFF'FFFF)는 해당 프로세스가 독립적으로 사용할 수 있는 영역이며 모든 프로세스가 독립적으로 사용하는 Private공간이다.

**GlobalMemoryStatus()**라는 함수를 사용하면 현재 물리메모리의 크기와 Paging파일의 크기, 가상주소 공간의 크기등을 구할 수 있다.

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

#define DIV (1024\*1024)

const char\* division = "MB";

#define dprint(expr) printf(#expr " = %d %s\n", (expr/DIV)+1, division)

void main()  
{  
    MEMORYSTATUS ms;  
    ms.dwLength = sizeof( ms);

    GlobalMemoryStatus( &ms );

    dprint( ms.dwTotalPhys );  
    dprint( ms.dwAvailPhys );  
    dprint( ms.dwTotalPageFile);  
    dprint( ms.dwAvailPageFile);  
    dprint( ms.dwTotalVirtual);  
    dprint( ms.dwAvailVirtual);  
}

결국 하나의 프로그램이 실행되면 모든 프로세스는

1. **4G의 가상주소가 공간을 가지게 되고**
2. **exe가 가상주소에 Mapping되고**
3. **Stack과 Heap이 만들어진다.**
4. **또한 exe의 .idata섹션에있는 DLL이 가상주소공간에 Mapping된다.**

위 그림은 하나의 프로세스가 실행중인때의 가상주소공간의 모양이다.

아래 함수를 사용하면 현재 프로세스에 Mapping 되어 있는 exe와 DLL들의 주소를 구할 수 있다.

**HMODULE** **GetModuleHandle(** **LPCTSTR** *lpModuleName* **);**

**2.1 Window Object**

**2.2 Object Category**

**2.3 Kernel Object**

|  |
| --- |
| **2.1 Window Object** |
| **2.1.1 윈도우 핸들(HWND)** |
| 현재 실행중인 **모든 윈도우에는 자신만의 고유한 번호가 있다**. 일반적으로 이 번호를 윈도우 **핸들**이라고 부른다. **HWND** 라는 데이터 타입을 사용하는데 32비트 정수이다. |
| 2 |
| * **윈도우 핸들 구하기** |
| 아래의 함수들을 사용하면 특정 윈도우의 핸들을 구할 수 있다. |
| **HWND FindWindow( LPCTSTR *lpClassName*, LPCTSTR *lpWindowName* );** |
| FindWindow 함수는 첫 번째 인자로 윈도우클래스 이름을, 두 번째 인자로 윈도우 캡션바의 내용을 받아서 해당하는 윈도우의 찾아서 핸들을 리턴 한다. |
|  |
| **HWND WindowFromPoint( POINT *Point* );** |
| WindowFromPoint 함수는 스크린 좌표 1개를 인자로 받아서 해당 좌표 아래 있는 윈도우의 핸들을 리턴 한다. |
| 아래는 계산기 윈도우의 핸들을 구하는 예제이다. |
| #include <windows.h>  #include <stdio.h>  int main(void)  {  HWND hwnd = FindWindow( 0, "계산기");  if ( hwnd == 0 )  printf("계산기를 먼저 실행해 주세요\n");  else  printf("계산기 윈도우 핸들 : 0x%08x\n", hwnd);  return 0;  }  윈도우 클래스를 알지 못할 경우 FindWindow의 첫 번째 인자로 0을 지정하면 캡션바의 내용만을 가지고 윈도우를 찾는다. |
| |  | | --- | | * **윈도우 관련 함수** | | Win32 API 함수 들 중에는 수많은 윈도우 관련함수가 있다. 이런 함수들의 대부분은 첫 번째 인자로 윈도우 핸들을 갖는다. | | BOOL DestroyWindow( **HWND *hWnd*** );  BOOL IsWindowVisible( **HWND *hWnd*** );  BOOL MoveWindow( **HWND *hWnd***, int *X*, int *Y*, int *nWidth*, int *nHeight*,  BOOL *bRepaint* );  BOOL SetWindowPos( **HWND *hWnd***, HWND *hWndInsertAfter*, int *X*, int *Y*, int *cx*, int *cy*, UINT *uFlags* );  int SetWindowRgn( **HWND *hWnd***, HRGN *hRgn*, BOOL *bRedraw* ); | | FindWindow 함수로 얻을 핸들을 위의 함수들에 전달하면 해당 윈도우를 이동하거나 사라지게 하는 등의 작업을 수행할 수 도 있다. | |
| #include <windows.h> #include <stdio.h> #include <conio.h>  int main(void) {  char name[256]; HRGN hRgn; HWND hwnd = FindWindow( 0, "계산기");  if ( hwnd == 0 )  printf("계산기를 먼저 실행해 주세요\n");  else {  printf("계산기 윈도우 핸들 : 0x%08x\n", hwnd);  GetClassName( hwnd, name, 256);  printf("계산기 윈도우의 클래스 이름 : %s\n", name);  getch(); ShowWindow( hwnd, SW\_HIDE );  getch(); ShowWindow( hwnd, SW\_SHOW );  getch();  hRgn = CreateEllipticRgn( 0,0,200,200);  SetWindowRgn( hwnd, hRgn, TRUE);  DeleteObject( hRgn);  }  return 0;  } |
| 위 예제는 계산기 윈도우의 핸들을 얻은 후에 윈도우 클래스 이름을 출력하고 윈도우를 사라 졌다가 나타나게 한 후 마지막으로 계산기윈도우를 동그랗게 변형하는 예제이다. |
| 결국 |
| “모든 윈도우는 저마다 고유한 번호(윈도우 핸들)가 있다. “  “대다수의 윈도우 관련 함수는 윈도우 핸들을 첫 번째 인자로 받는다.” |
| 연습문제 |
| 1. SetWindowPos 함수를 사용하면 윈도우를 최상위 윈도우로 만들거나 해제 할수 있다. 계산기를 최상위 윈도우로 변경해 보자. 2. 위 프로그램의 마지막 부분에 DestroyWindow( hwnd ) 코드를 추가해 보자. 계산기 윈도우가 파괴 되는가 ? 파괴되지 않는다면 왜 그런지 MSDN에서 찾아 보자. 3. 현재 계산기의 위치(스크린 좌표)와 크기를 구해보자. 4. 현재 실행중인 모든 윈도우를 열거해 보자.(EnumWindows) 5. 4에서 캡션바에 문자열이 없거나, 보이지 않거나, 부모(소유)윈도우가 있는 경우를 제외하고 열거해보자.   작업관리자와 비교해보자. |
| * 참고자료 – Spy++ |
| 캡션바가 없는 윈도우의 핸들은 구하려면 어떻게 해야 할까 ? 윈도우 클래스 이름을 알면 구할 수 있다.  Visual C++ 과 함께 제공되는 Spy++프로그램을 사용하면 윈도우 클래스 이름을 구할 수 있다. |
| spyspyaa |
| SPY++의 Spy메뉴에서 Find Window항목을 선택 한 후 찾기 도구를 사용해서 윈도우의 클래스의 이름을 구할 수 있다. |
| Spy++같은 프로그램의 어떻게 만들까 ? |
| 아래 예제는 마우스 캡쳐와 WindowFromPoint()함수를 사용해서 Spy++ 처럼 윈도우 클래스의 이름을 얻어 내는 프로그램이다. |
| LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)  {  POINT pt;  char str[256];  HWND hwndDest;  switch (message)  {  case WM\_LBUTTONDOWN:  SetCapture( hwnd );  break;  case WM\_LBUTTONUP:  if ( GetCapture() == hwnd )  {  ReleaseCapture();  GetCursorPos( &pt ); /\* 커서의 위치를 스크린 좌표로 얻는다 \*/  hwndDest = WindowFromPoint( pt ); /\* 스크린 좌표 pt 아래 있는 윈도우의 핸들을 얻는다. \*/  GetClassName( hwndDest, str, 256 ); /\* 윈도우 클래스 이름을 얻는다. \*/  MessageBox( hwnd, str, "Class Name", MB\_OK);  }  break;  case WM\_DESTROY:  PostQuitMessage(0);  break;  default:  return DefWindowProc(hwnd, message, wParam, lParam);  }  return 0;  } |
|  |
| 시작 버튼이 있는 작업바(TaskBar)의 클래스 이름을 구해보자. |

|  |
| --- |
| **2.1.2 윈도우 객체(Window Object)** |
| * **윈도우 객체(Windows Object)란 ?** |
| CreateWindow()함수를 사용하면 윈도우를 만들 수 있다. 그럼, 윈도우의 정확한 정체는 무엇일까 ? 사용자가 윈도우를 작게 했다가(minimize) 다시 복구 했을 때 윈도우는 항상 동일한 위치에 동일한 크기를 가지고 동일한 스타일로 복구 되는 것을 볼 수 있다. 그럼 어딘가에 해당 윈도우의 모든 정보를 보관해야 되지 않을까 ? |
| 윈도우는 결국 하나의 구조체 형식의 객체이다. 즉, 사용자가 CreateWindow()함수를 사용해서 윈도우를 만들 때 시스템은 **구조체형식의 변수를 하나 생성한다. 그리고**  CreateWindow()의 **인자로 초기화** 한다. 이 변수를 흔히 **윈도우 객체** 라고 부른다. 이 윈도우 객체의 내용으로 화면에 윈도우를 그리는 함수가 ShowWindow()이다. |
|  |
| 결국 윈도우 핸들도 윈도우 객체를 가르키는 번호이다. |
|  |
| **참고** |
| Matt Pietrick 이 쓴 “Windows 95 System Programming Secrets” 라는 책에 보면 윈도우을 관리하기 위해 시스템(Windows 95)이 사용하는 구조체를 소개하고 있다. 비록 오래된 책이지만 읽어 볼 만한 책이다. |
| typedef struct \_WND32  {  struct \_WND32 \*hWndNext; // 00h (GW\_HWNDNEXT) HWND of next sibling window  struct \_WND32 \*hWndChild; // 04h (GW\_CHILD) First child window  struct \_WND32 \*hWndParent; // 08h Parent window handle  struct \_WND32 \*hWndOwner; // 0Ch Owning window handle  RECTS rectWindow; // 10h Rectangle describing entire window  RECTS rectClient; // 18h Rectangle for client area of window  WORD hQueue; // 20h Application message queue handle  WORD hrgnUpdate; // 22h window region needing an update  WORD wndClass; // 24h handle to an INTWNDCLASS  WORD hInstance; // 26h hInstance of creating application  WNDPROC lpfnWndProc; // 28h Window procedure address  DWORD dwFlags; // 2Ch internal state flags  DWORD dwStyleFlags; // 30h WS\_XXX style flags  DWORD dwExStyleFlags; // 34h WS\_EX\_XXX extended style flags  DWORD moreFlags; // 38h flags  HANDLE ctrlID; // 3Ch GetDlgCtrlId or hMenu  WORD windowTextOffset; // 40h Offset of the window's text in atom heap  WORD scrollBar; // 42h DWORD associated with the scroll bars  WORD properties; // 44h Handle for first window property  WORD hWnd16; // 46h Actual HWND value for this window  struct \_WND32\* lastActive; // 48h Last active owned popup window  HANDLE hMenuSystem; // 4Ch handle to the system menu  DWORD un1; // 50h  WORD un2; // 54h  WORD classAtom; // 56h See also offs. 2 in the field 24 struct ptr  DWORD alternatePID; // 58h  DWORD alternateTID; // 5Ch  } WND32, \*PWND32; |
| **2.2 Object Category** |
| * **Question** |
| 다음 그림을 보자. |
|  |
| * A프로세스가 윈도우를 생성했다. 윈도우 핸들은 0x00001234이다. * A프로세스가 B프로세스에게 SendMessage()를 사용해서 윈도우 핸들을 전달했다. * B프로세스에서 MoveWindow() 함수를 사용해서 A에서 전달받은 윈도우를 움직이려고 한다. |
| B는 A가 만든 윈도우를 움직일 수 있을까 ?? |
|  |
| 답은 움직인다. **윈도우의 핸들은 모든 프로세스에 대해 전역적이다. (Public to All Process)** |
|  |
| 그럼 이번에는 파일을 생각해 보자. |
|  |
| * A프로세스가 파일을 생성했다. 파일의 핸들은 0x000007F0 이다. * A프로세스가 B프로세스에게 SendMessage() 함수를 사용해서 파일의 핸들을 전달했다. * B프로세스에서 A에서 전달 받은 파일의 핸들을 사용해서 WriteFile()로 임의에 data를 파일에 쓰려고 한다. |
|  |
| B는 A가 만든 파일에 Data를 쓸 수 있을까 ? |
|  |
| 답은 실패한다. 파일은 모든 프로세스에 **대해 상대적 핸들을 사용한다**.(상대적 핸들에 대해서는 뒷장에서 설명한다.) |
|  |
| 그럼 A가 Pen을 만들어서 펜의 핸들을 B에게 보냈다면 B는 A가 만든 펜을 사용할 수 있을까 ?  아이콘은 ? 커서는 ? 메뉴는 ? 도대체 어떤 차이가 있는 걸까 ? |

|  |
| --- |
| * **객체(Object)와 핸들(Handle)** |
| CreateWindow() 함수를 사용해서 윈도우를 만들 때 운영체제는 윈도우를 관리하기 위해 윈도우 객체(구조체)를 생성한다.  또한, CreatePen()을 사용해서 펜을 만들 때 운영체제는 펜을 관리하기 위해 펜 객체를 생성한다. 마찬가지로 파일을 만들기 위해 CreateFile()함수를 호출하면 파일을 관리하기 위한 객체를 생성한다. |
|  |
| 윈도우프로그램의 다양한 구성요소를 관리하기 위해 구조체를 사용하고 이것을 객체라고 부른다.  객체는 그래픽이미지나, 파일, 스레드, 펜, 윈도우등의 시스템 자원을 나타내는 자료구조 이다.  흔히 윈도우를 객체 기반 운영체제라고 이야기한다.  API 함수중 Createxxx 로 시작하는 함수는 전부 객체를 생성한다. 또한 객체를 파괴하기 위해 Destroyxxx, Deletexxx, CloseHandle()등의 함수를 사용한다.  또한, 이와 같은 객체를 가리키는 번호를 핸들이라고 한다. |
| * **Object Category** |
| 윈도우 시스템이 제공하는 객체(Object)는 크게 3가지로 분류할 수 있다. |
| * **User Object** : 윈도우와 관련된 작업을 수행하기 위한 Object.   프로세스에 전역적(public) 핸들을 갖는다.  파괴하는 함수는 대부분 Destroy로 시작한다. |
| Accelerator table, Caret, Cursor, DDE conversation, Hook, Icon, Menu, Window, Window position |
|  |
| * **GDI Object** : 그래픽 작업과 관련된 Object.   프로세스에 지역적(private) 핸들을 갖는다.  파괴하는 함수는 대부분 Delete로 시작한다. |
| Bitmap, Brush, DC, Enhanced metafile, Enhanced-metafile DC, Font, Memory DC, Metafile, Metafile DC, Palette,  Pen and extended pen, Region |
|  |
| * **Kernel Object** : 파일, 메모리, 프로세스, IPC 등과 같은 작업에 관련된 Object.   프로세스에 **상대적(Specific) 핸들을** 갖는다.  CloseHandle()함수로 파괴한다.(실제로는 파괴가 아니고 참조개수 감소, 뒤에서 설명) |
| Access token, Change notification, Communications device, Console input, Console screen buffer, Desktop, Event,  Event log, File, File mapping, Heap, Job, Mailslot, Module, Mutex, Pipe, Process, Semaphore, Socket, Thread,  Timer, Timer queue, Timer-queue timer, Update resource, Window station |

window, Menu, Icon, Cursor 등과 같은 UI 요소를 나타내는 객체를 User Object라고한다.

User Object를 생성하기 위해서는 User32.DLL 안에 있는 함수를 사용한다.

User Object는 객체당 1개의 핸들을 사용한다. 즉 특정 윈도우에 접근하기 위해서 모든 프로세스는 동일한 핸들 값을 사용한다. 또한 핸들은 DuplicateHandle을 사용해서 복사하거나 자식 프로세스에 상속 될 수 없다.(복사, 상속할 필요가 없다.)

CreateWindow()함수는 Window Object를 생성하고 DestroyWindow()는 Window Object를 파괴한다.

|  |  |  |
| --- | --- | --- |
| **User object** | **Creator function** | **Destroyer function** |
| Accelerator table | CreateAcceleratorTable | DestroyAcceleratorTable |
| Caret | CreateCaret | DestroyCaret |
| Cursor | CreateCursor, LoadCursor, LoadImage | DestroyCursor |
| DDE conversation | DdeConnect, DdeConnectList | DdeDisconnect, DdeDisconnectList |
| Hook | SetWindowsHookEx | UnhookWindowsHookEx |
| Icon | CreateIconIndirect, LoadIcon, LoadImage | DestroyIcon |
| Menu | CreateMenu, CreatePopupMenu, LoadMenu, LoadMenuIndirect | DestroyMenu |
| Window | CreateWindow, CreateWindowEx, CreateDialogParam,  CreateDialogIndirectParam, CreateMDIWindow | DestroyWindow |
| Window position | BeginDeferWindowPos | EndDeferWindowPos |

|  |
| --- |
| C/C++ 실행환경(Run-time)은 많은 미리 정의된 전역 변수를 가지고 있다. |
|  |
| #include <iostream>  #include <windows.h>  using namespace std;  #define dprint(x) cout << #x << " : " << x << endl  int main()  {  dprint( \_osver );  dprint( \_winmajor );  dprint( \_winminor );  dprint( \_winver );  dprint( \_\_argc );  dprint( \_\_argv[0] );  dprint( \_environ[0]);  dprint( \_pgmptr );  } |

|  |
| --- |
| * **Process Current Directory** |
| 모든 프로세스는 현재 디렉토리 정보를 보관한다. 어떤 함수가 파일 이름를 인자로 받을 때 전체 경로를 지정하지 않으면 현재 디렉토리 안에 있는 파일이라고 가정하고 작업을 하게된다.  프로세스의 현재 디렉토리 를 얻거나 변경할 때는 아래의 함수를 사용한다. |
| * **BOOL** **SetCurrentDirectory(** **LPCTSTR** *lpPathName* **);** * **DWORD** **GetCurrentDirectory(** **DWORD** *nBufferLength***, LPTSTR** *lpBuffer* **);** |
|  |
| * **Process Error-Mode** |
| 프로그램을 수행하다가 몇가지의 심각한 에러가 발생했을 때 시스템이 처리할지 프로세스가 처리할지를 아래 함수로 지정할수 있다. |
| **UINT** **SetErrorMode(** **UINT** *uMode* **);** |

**CreateProcess 인자 설명**

|  |  |
| --- | --- |
| BOOL CreateProcess( LPCTSTR *lpApplicationName*,  LPTSTR *lpCommandLine*,  LPSECURITY\_ATTRIBUTES *lpProcessAttributes*,  LPSECURITY\_ATTRIBUTES *lpThreadAttributes*,  BOOL *bInheritHandles*,  DWORD *dwCreationFlags*,  LPVOID *lpEnvironment*,  LPCTSTR *lpCurrentDirectory*,  LPSTARTUPINFO *lpStartupInfo*,  LPPROCESS\_INFORMATION *lpProcessInformation* ); | |
| * **lpApplicationName** | |
| 실행할 exe화일의 이름. 전체경로를 주지 않을 경우 프로세스의 현재 드라이버와 현재 디렉토리를 사용한다. 열려진 Path를 검색하지는 않는다. 이 파라미터로 NULL을 넘길 수 있다. 하지만 이 경우에는 lpCommandLine인자에 실행할 파일의 이름이 포함되어 있어야 한다. | |
| * **lpCommandLine** | |
| 생성될 자식 프로세스에 전달할 Commnad line을 지정한다. lpApplicationName이 NULL 일 경우 lpCommandLine의 첫 번째 문자열을 exe 파일의 이름으로 가정한다. 전체 경로를 지정하지 않고 파일의 이름만 지정한 경우 아래의 순서대로 검색한다. | |
| 1. Application이 Load된 디렉토리 2. 부모 프로세스의 현재 디렉토리 3. 윈도우 시스템 디렉토리 4. 윈도우 디렉토리 5. PAHT 환경변수에 포함되어 있는 디렉토리 | |
| * **lpProcessAttributes** | |
| 생성될 자식 프로세스의 프로세스 객체의 보안 속성. | |
| * **lpThreadAttributes** | |
| 생성될 자식 프로세스의 스레드 객체의 보안 속성. | |
| * **bInheritHandles** | |
| Object Table의 핸들을 자식프로세스에게 상속할 것인가를 지정. TRUE이면 Object Table의 핸들중 상속가능한 핸들만 자식프로세스에게 상속된다. | |
| * **dwCreationFlags** | |
| 프로세스의 우선 순위클래스와 생성 Flag를 지정한다. 아래와 같은 상수 값을 지정할 수 있다. | |
| 우선 순위 클래스 | Description |
| ABOVE\_NORMAL\_PRIORITY\_CLASS |  |
| BELOW\_NORMAL\_PRIORITY\_CLASS |  |
| HIGH\_PRIORITY\_CLASS |  |
| IDLE\_PRIORITY\_CLASS |  |
| NORMAL\_PRIORITY\_CLASS |  |
| REALTIME\_PRIORITY\_CLASS |  |

|  |
| --- |
| **lpEnvironment** |
| 생성될 자식 프로세스에 지정할 환경 블록의 포인터. NULL을 지정하면 부모의 환경 블록이 전달된다. |
| **lpCurrentDirectory** |
| 새롭게 생성되는 자식 프로세스를 위해 현재 드라이브와 현재 디렉토리를 나타내는 문자열. 반드시 드라이브명을 포함한 전체경로를 지정해야 한다. 만약 NULL을 전달하면 부모 프로세스와 동일한 드라이브와 디렉토리를 사용한다. |
| **lpStartupInfo** |
| STARTUPINFO 구조체의 포인터. 생성될 프로세스의 window, station 객체, 표준입출력 핸들, 윈도우의 크기와 위치등을 지정한다. |
| **lpProcessInformation** |
| PROCESS\_INFORMATION 객체의 주소. 생성된 프로세스와 스레드의 ID와 핸들을 얻어온다. |

|  |  |  |
| --- | --- | --- |
| * **cb** | | |
| 구조체의 크기를 전달한다. | | |
| * **lpReserved** | | |
| 예약됨. CreateProcess()를 호출하기전에 반드시 0으로 설정하고 호출해야 한다. | | |
| * **lpDesktop** | | |
| window station과 desktop 객체의 이름을 전달한다. | | |
| * **lpTitle** | | |
| console Application을 실행할 경우 새로운 Console 창의 caption에 전달할 문자열. NULL을 전달할 경우 실행파일의 완전한 경로가 캡션바에 놓인다. GUI Application 을 실행할 경우 NULL로 전달해야 한다. | | |
| * **dwX, dwY** | | |
| Console Application를 실행할 경우 Console 창의 위치를 지정한다.  GUI Application을 실행할 경우 CreateWindow()함수의 CW\_USEDEFAULT값으로 전달된다.  **dwFlags**인자로 **STARTF\_USEPOSITION** 를 지정할 때만 적용된다. | | |
| * **dwXSize, dwYSize** | | |
| Console Application를 실행할 경우 Console 창의 크기를 지정한다.  GUI Application을 실행할 경우 CreateWindow()함수의 CW\_USEDEFAULT값으로 전달된다.  **dwFlags**인자로 **STARTF\_USESIZE**  를 지정할때만 적용된다. | | |
| * **dwXCountChars, dwYCountChars** | | |
| Console Application을 실행할 때 새로운 Console 창의 스크린 버퍼의 너비와 높이를 지정한다.  **dwFlags**인자로 **STARTF\_USECOUNTCHARS** 를 지정할때만 적용된다. | | |
| * **dwFillAttribute** | | |
| dwFlags의 인자로 **STARTF\_USEFILLATTRIBUTE** 가 지정될때만 적용된다.  Console Application을 실행할때 새롭게 생성되는 Console창의 글자색과 배경색을 지정한다. 다음과 같은 상수값이 지정될수 있다. | | |
| FOREGROUND\_BLUE, FOREGROUND\_GREEN, FOREGROUND\_RED, FOREGROUND\_INTENSITY, BACKGROUND\_BLUE, BACKGROUND\_GREEN, BACKGROUND\_RED, BACKGROUND\_INTENSITY | | |
| 글자색을 빨간색으로, 배경색을 흰색으로 하기 위해서는 아래 처럼 지정한다. | | |
| **FOREGROUND\_RED| BACKGROUND\_RED| BACKGROUND\_GREEN| BACKGROUND\_BLUE** | | |
| * **wShowWindow** | | |
| dwFlags의 인자로 STARTF\_USESHOWWINDOW 가 지정될때만 적용된다.  GUI Application에서 첫번째로 ShowWindow를 호출할때 2번째 인자로 지정된다. GUI Application에서 첫번째로 호출하게 되는 ShowWindow()는 wShowWindow항목이 가 있을 경우에는 무시된다. | | |
| * **cbReserved2, lpReserved2** | | |
| C Runtime에 의해 예약되어 있다. 반드시 0을 지정하다. | | |
| * **hStdInput, hStdOutput, hStdError** | | |
| 새롭게 생성되는 프로세스의 표준 입력, 출력, 에러 핸들을 지정한다.  dwFlags의 인자로 STARTF\_USESTDHANDLES 가 지정될때만 적용된다. | | |
| * **dwFlags** | | |
| STARTUPINFO 구조체중에서 어떤 멤버를 적용할 지를 지정한다. 아래와 같은 값들이 OR해서 지정할 수 있다. | | |
| **Value** | **Meaning** |
| **STARTF\_FORCEONFEEDBACK** |  |
| **STARTF\_FORCEOFFFEEDBACK** |  |
| **STARTF\_RUNFULLSCREEN** | Console Application을 Full-Screen Mode로 실행되게 한다. |
| **STARTF\_USECOUNTCHARS** | dwXCountChars, dwYCountChars 멤버를 적용한다.  (95/98/ME 는 지원되지 않는다.) |
| **STARTF\_USEFILLATTRIBUTE** | dwFillAttribute 멤버를 적용한다. |
| **STARTF\_USEPOSITION** | dwX, dwY 멤버를 적용한다. |
| **STARTF\_USESHOWWINDOW** | wShowWindow 멤버를 적용한다. |
| **STARTF\_USESIZE** | dwXSize, dwYSize 멤버를 적용한다. |
| **STARTF\_USESTDHANDLES** | hStdInput, hStdOutput, hStdError 멤버를 적용한다. |

|  |
| --- |
| * **프로세스의 종료 코드** |
| 프로세스 커널 오브젝트 안에는 프로세스 종료코드가 포함되어 있다. |
|  |
| 아래 함수를 사용하면 특정 프로세스의 종료 코드를 얻어올 수 있다. |
| **BOOL** **GetExitCodeProcess(** **HANDLE** *hProcess***, LPDWORD** *lpExitCode* **);** |
| 첫 번째 인자로 프로세스의 핸들을 두 번째 인자로 종료 코드 값을 담아올 DWORD값의 주소를 전달한다. |
| 프로세스가 현재 실행 중 일 경우 STILL\_ACTIVE(0x103, 10진수 259) 값이 나온다. |
| 프로세스가 종료된 경우 다음의 값이 종료 코드 값으로 나온다. |
| * main 또는 WinMain함수의 리턴값. * TerminateProcess또는 ExitProcess함수로 전달한 값. |

|  |
| --- |
| * **TerminateProcess** |
| 실행중인 프로세스를 죽이기 위해서는 아래의 함수를 사용한다. |
| **BOOL** **TerminateProcess(** **HANDLE** *hProcess***, UINT** *uExitCode* **);** |
| 첫 번째 인자로는 프로세스의 핸들을 두 번째 인자로는 종료 코드를 전달한다. 이 함수는 다른 프로세스를 죽이거나 스스로 죽을 수도 있다.  TerminateProcess 함수는 비동기 함수이다. 즉, 이 함수가 리턴 할 때 해당 하는 프로세스가 바로 죽은 것은 아니다. |
| TerminateProcess( hProcess, 0);  GetExitCodeProcess( hProcess, &code ); // hProcess가 완전히 종료 되었다고 확신할 수는 없다. |
| 프로세스가 죽을 때 프로세스 커널 오브젝트는 시그널 된다. 아래 함수는 특정 커널 오브젝트가 시그널 될 때 까지 현재 스레드 를 대기 시킨다. |
| **DWORD** **WaitForSingleObject(** **HANDLE** *hHandle***, DWORD** *dwMilliseconds* **);** |
| TerminateProcess를 호출한 후 프로세스가 완전히 종료 될 때 까지 대기하려면 아래와 같이 만들면 된다. |
| TerminateProcess( hProcess, 0 );  **DWORD ret = WaitForSingleObject( hProcess, 5000);**  if ( ret == WAIT\_TIMEOUT) // 5초가 지난 경우 – 프로세스를 종료 할 수 없다.  else if ( ret == WAIT\_OBJECT\_0) // 프로세스 커널 Object 가 Signal 되었다. 프로세스가 종료 되었다.  Else // hProcess 핸들이 잘못된 경우 |
| TerminateProcess는 프로세스를 강제로 종료 시킨다. 강제로 종료되는 프로세스는 자신이 할당한 자원을 정리하지 못하게 될 것이다. 그러므로 TerminateProcess는 최후의 수단으로 사용해야 한다. |
|  |
| * **WaitForSingleObject** |
| 모든 커널 오브젝트에는 Signal 항목이 있다. 커널 오브젝트가 언제 시그널 되는지는 커널 오브젝트의 종류 마다 다르다. 프로세스 커널 오브젝트는 프로세스가 종료 될 때 시그널 된다. 파일 커널 오브젝트는 파일을 쓰거나 읽기 작업을 마쳤을 때 시그널 된다. 특정 커널 오브젝트가 시그널 될 때 까지 대기 하려면 아래 함수를 사용한다. |
| DWORD WaitForSingleObject( HANDLE hHandle, DWORD dwMilliseconds ); |
| 첫 번째 인자로 커널 오브젝트의 핸들을 두 번째 인자로 시간을 전달한다. 2번째 인자로 0을 전달하면 현재 커널 오브젝트의 시그널 상태를 즉시 알아온다. 그리고 INFINITE를 전달하면 시그널 될 때 까지 무한히 대기한다. |
| * WaitForSingleObject( hProcess, 0); // hProcess의 상태의 시그널 상태를 조사한다.(대기하지 않는다.) * WaitForSingleObject( hProcess, 5000); // hProcess가 시그널 되거나, 5초가 지나면 리턴 된다. * WaitForSingleObject( hProcess, INFINITE); // hProcess가 시그널 될 때 까지 무한히 대기한다. |
|  |
| **WaitForSingleObject 는 다음 중 1개의 값을 리턴 한다.** |
| * WAIT\_OBJECT\_0 : 커널 오브젝트가 시그널 된 경우. * WAIT\_TIMEOUT : 2번째 인자로 지정된 시간이 다 된 경우. * WAIT\_ABANDONED : 뮤텍스 커널 오브젝트에서만 발생. 뮤텍스를 소유한 스레드가 Release하지 않고 종료한 경우. * WAIT\_FAILED : 함수 호출에 실패한 경우.(주로 잘못된 핸들) |

|  |
| --- |
| * **프로세스를 종료 하는 방법** |
| 프로세스는 다음의 4가지 방법으로 종료 할 수 있다. |
| * **주 스레드가 엔트리포인트 함수에서 리턴 한다.(가장 좋은 방법)** |
| 주 스레드가 엔트리포인트 함수에서 리턴 하면 프로세스는 자신이 사용하던 모든 자원을 반납하고 종료된다.  가장 좋은 방법으로 항상 프로세스가 이와 같이 종료 되도록 설계해야 한다. |
| * **프로세스내의 하나의 스레드에서 ExitProcess 함수를 호출.(좋지 않은 방법)** |
| 프로세스내의 어떠한 스레드라도 ExitProcess를 호출하면 프로세스는 종료된다. 이 방법은 C/C++프로그램의 자원 반납을 하지 못한채 종료 하게 된다. 특히 C++ 객체의 소멸자가 호출되지 않는다. 그러므로 이 방법으로 프로세스를 종료하는 것은 좋지 않다. |
| * **다른 프로세스 안에 있는 스레드에서 TerminateProcess함수 호출.(좋지 않은 방법)** |
| 이 방법은 다른 프로세스를 강제로 죽일 때 사용한다. 이 방법에 의해 강제로 죽게 되는 프로세스는 자신이 사용하던 자원을 정리할 기회를 갖지 못하므로 가장 안좋은 방법이다. 이 방법은 응답하지 않은 프로세스를 죽이기 위한 최후의 수단으로 사용해야 한다. 일반 적으로 다른 프로세스를 종료하게 하려면 WM\_CLOSE메세지를 전달한다. |
| * **프로세스내의 모든 스레드가 스스로 죽음.** |
| 주 스레드가 시작함수(main, WinMain)에서 리턴하거나, 함수의 끝에 도달하면 프로세스 내의 모든 스레드가 종료되고 프로세스가 종료된다. 하지만 주 스레드가 ExitThread()함수를 사용해서 종료가 되었다면 다른 스레드는 계속 실행 될 수 있다. 이때 다른 모든 스레드가 종료 되면 프로세스는 종료된다. 이 경우 마지막 스레드가 소멸될때 전달된 exit code가 프로세스의 종료 코드가 된다. |
|  |
| * **프로세스가 종료 될때** |
| 프로세스가 종료 될 때는 아래와 같은 일들을 수행된다. |

* + 프로세스내의 모든 스레드가 종료된다.
  + 프로세스가 할당한 모든 User, GDI Object 가 파괴된다.
  + 모든 커널Object에 대해 핸들을 닫는다.(사용횟수가 0이 되면 파괴된다.)
  + 프로세스의 exit코드가 STILL\_ACTIVE에서 ExitProcess 또는 TerminateProcess로 부터 전달된 값으로 변경
  + 프로세스 Kernel Object가 Signal 된다.
  + 프로세스 Kernel Object의 사용횟수가 1감소한다.

|  |
| --- |
| * **프로세스(커널 오브젝트)의 핸들은 상대적이다.** |
| 커널오브젝트의 핸들은 각 프로세스에 상대적이다. 가령 A라는 커널 오브젝트가 있을 때 프로세스 x는 자신의 object table에 있는 인덱스 값을 A에 접근하기 위해 사용한다. 또한 프로세스 y는 A에 접근하기 위해 자신의 object Table에 있는 인덱스 값을 사용한다. 결국 동일한 A 커널 오브젝트를 접근할 때 프로세스 x, y는 서로 다른 핸들 값을 사용한다. |
| * **Process ID** |
| 모든 프로세스는 자신만의 ID를 갖는다. 이 ID는 시스템내의 모든 프로세스에서 Unique한다.  그래서 프로세스 X, Y 에서 프로세스 A에 접근할때는 모두 200 번이라는 동일한 ID를 사용하게 된다. |
| * **프로세스 ID를 가지고 핸들을 구하기** |
| 윈도우 핸들을 알 때 아래 함수를 사용하면 해당 윈도우를 만든 프로세스(스레드)의 ID를 얻을수 있다. |
| **DWORD GetWindowThreadProcessId( HWND***hWnd***,     LPDWORD***lpdwProcessId* **);** |
| 또한 아래 함수를 사용하면 프로세스(스레드)ID를 가지고 프로세스(스레드)의 핸들을 얻을수 있다.  **HANDLE** **OpenProcess(** **DWORD** *dwDesiredAccess***, BOOL** *bInheritHandle***, DWORD** *dwProcessId* **);** |
|  |
|  |