1. Unicode

1. 1 Double Byte Character Set(DBCS)

DBCS는 문자셋(Character Sets) 중 하나로써 언어에 따라서 1byte 나 2byte를 사용하는 방법이다. 예를 들어 한글은 2byte, 영문은 1byte로 저장된다.
이 방법은 일반적으로 사용해 왔던 방법인데 다음과 같은 문제점을 가지고 있다.

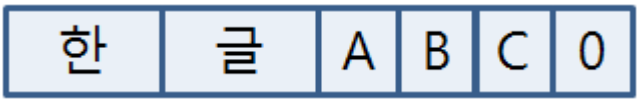
- 만약 "한글ABC" 라는 문자열에 몇 개의 문자가 저장되었는지 알고 싶다면 간단히 해결할 수 없다. 한글 문자가 몇 개인지, 영문이 몇 개인지를 체크할 수 있어야 실제 문자의 개수를 파악할 수 있다.
- "한글ABC" 라는 문자열을 한 문자씩 이동하면서 출력하고자 한다면 이 또한 간단히 해결할 수 없다. 현재 위치의 문자가 한글인지, 영문인지를 확인해서 한글이면 2byte 이동, 영문이면 1byte 이동을 해야 한다.
- 국제적인 범용 프로그램 제작시 "한글ABC" 는 한글 윈도우에서는 그대로 보이지만 중국어 윈도우나 일본어 윈도우에서는 한글 코드 영역에 한자 코드 또는 히라가나가 맵핑되어 있으므로 제대로 보이지 않는다.

아래 코드를 통해서 DBCS 문자셋의 성질을 이해해 보도록 하자.

```
void main(){
        char str[] = "한글123";
        printf("%d₩n", sizeof(str));
}
```

```
8
```

[예제 1-1] DBCS



DBCS 방식은 위 그림처럼 한글은 2byte 영문은 1byte로 저장한다. 따라서 sizeof()연산자 결과 8의 값이 리턴된다.

아래 소스는 문자열을 1byte 씩 이동하면서 출력하고 있다. 문제는 한글의 경우 2byte를

이동해야 하지만 **1byte씩 증가하므로 문자가 깨지는** 것을 볼 수 있다.

```
void main(){
        char str[] = "한글ABC";
        char *pStr = str;
        while(*pStr){
                printf("%s₩n", pStr);
                pStr++;
        }
}
```

```
한글ABC
畸?BC
글ABC
?BC
ABC
BC
C
```

[예제 1-2] DBCS 이동

위의 문제를 해결하려면 아래 함수를 사용해야 한다.

BOOL IsDBCSLeadByte(BYTE ch)	ch 가 나타내는 값이 2byte 문자의 첫 번째 바이트라면 TRUE 값을 리턴한다.
LPTSTR CharNext(LPCTSTR lpsz)	Lpsz 이 가르키는 문자열에서 다음 문자의 주소를 얻는다.(즉 2byte or 1byte를 이동한다.)
LPTSTR CharPrev(LPCTSTR lpszS, LPCTSTR lpszC)	LpszS 에서 시작하는 문자열에서 lpszC 가 가르키는 문자의 바로 앞 문자를 구한다. (즉 2byte or 1byte를 이동한다.)

위의 코드를 수정해 보자.

```
#include <Windows.h>
void main(){
        char str[] = "한글ABC";
        char *pStr = str;
        while(*pStr){
```

```
                printf("%s₩n", pStr);
                if(IsDBCSLeadByte(*pStr))       pStr += 2;
                else                            pStr += 1;
        }
}
```

```
한글ABC
글ABC
ABC
BC
C
```

[예제 1-3] DBCS 이동 수정

1. 2 Unicode(Wide Character)

유니코드는 하나의 문자 코드로 전 세계의 모든 문자를 표현할 수 있는 코드 체계이다. 모든 문자는 16비트 즉 2byte로 표현되므로 최대 65536개의 문자를 표현할 수 있고 각 국가별로 코드 영역이 구분되어 있기 때문에 코드 페이지를 변경할 필요가 없다.
특정 문자는 세계의 어느 곳에서나 일정한 코드로 표현되며 따라서 유니코드로 작성된 프로그램은 별도의 재 컴파일 과정을 거칠 필요가 없다.
다음은 “한글ABC” 문자열이 유니코드로 메모리상에 구현된 모양이다. 모든 문자가 동일한 크기 즉 2byte로 구성되어 있다.

유니코드를 실제 적용하여 사용할 수 있도록 변형한 형태가 있는데 이를 UTF(UCS Transformaion format)이라고 한다. UTF-1, UTF-7, UTF-8 등, UTF-16 이 있는데 각 인코딩 방식은 유니코드 문자를 바이트 스트림에 어떻게 배치할 것인가가 다르다.
예를 들어 UTF-16은 유니코드의 표준대로 모든 문자를 16비트로 표현한다. UTF-8은 모든 문자를 8비트로 표현한다. 즉 UTF-16 보다 반정도의 메모리로 한 문자를 표현하는 방식이다.
UTF-8은 사람이 직접 읽기는 어렵지만 기계가 이 코드를 처리하기는 쉽다. 따라서 이

방식은 XML과 인터넷 기본 인코딩으로 널리 활용된다.

아래 코드는 Unicode로 작성한 프로그램이다.

```
void main(){
        wchar_t msg[] = L"한글ABC";
        printf("%d₩n", sizeof(msg));
}
```

```
12
```

[예제 1-4] 유니코드

wchar_t 는 문자를 저장할 때 2byte 로 저장한다.
'L' 지시어는 컴파일러에게 문자를 컴파일할 때 2byte씩 처리해달라는 지시어이다.
만약 해당 키워드를 삭제할 경우 아래와 같은 에러 메시지를 확인해 볼 수 있다.

error C2440: '초기화 중' : 'const char [8]'에서 'wchar_t []'(으)로 변환할 수 없습니다.

1. 3 String 함수

DBCS에서는 문자열을 다루기 위한 다양한 함수가 지원된다. 예를 들어 strcpy, strcmp, st
stlen, strcat… 위 함수들이 사용하는 알고리즘의 공통점은 문자열을 다룰 때 문자열의 끝
을 '₩o' 로 판단하는 것이다.
따라서 해당 함수들을 Unicode 방식의 문자열에서는 사용할 수 없다. 이유는 Unicode 방
식은 문자열의 끝을 나타내는 종결 문자도 2byte로 표현하기 때문이다.
즉, Unicode 관련 String 함수는 문자열의 끝을 '₩o' 이 2번 중복되었을 때로 인지하는
알고리즘을 갖는 새로운 함수를 필요로 한다.

아래 코드 결과를 확인해 보자.

```
#include <string.h>
void main(){
    wchar_t str[] = L"한글ABC";
    printf("%d₩n", strlen( (char*)str ) ); // ?
}
```

```
2
```

[예제 1-5] String 함수

strlen 함수는 '₩0' 을 찾을 때 까지 byte를 이동하며 개수를 파악한다. 따라서 문자열 중간에 '₩0' 이 들어간 경우 더 이상 이동하지 않고 리턴하게 된다.
이 문제 때문에 Unicode에서는 기존 String 함수를 사용하지 못하고 새로운 함수를 제공한다
.

size_t strlen(const char *_string_); DBCS문자열의 길이를 구한다.

size_t wcslen(const wchar_t *_string_); strlen의 Unicode 버전

위의 함수 선언부를 보면 파라미터 타입이 다른 것을 확인해 볼 수 있으며, 함수의 이름은 접두어가 달라지는 것을 볼 수 있다. 즉 새로 제공되는 Unicode 방식의 문자열 함수는 기존 스트링 함수의 접두어만 변경하면 된다.

```
#include <string.h>
void main(){
    wchar_t str[] = L"한글ABC";
    printf("%d₩n", wcslen( str ) ); // ?
}
```

```
5
```

[제 1-6] 유티코드 기반의 String 함수

1. 4 Macro 를 활용한 범용적 사용

지금까지 학습한 내용을 이해했다면 이제는 문자열을 다룰 때 DBCS 방식으로 처리할 것인가? 아니면 Unicode 방식으로 처리할 것인가? 하는 선택의 기로에 서게 될 것이다. 앞으로 만들어지는 프로그램의 경우 Unicode 방식으로 처리하는 것이 맞을 듯 한데 기존 프로그램과의 호환성 문제도 있고, 쉽게 결정할 내용은 아닌 것 같다.
그렇다면 하나의 소스코드를 사용해서 DBCS 용과 Unicode 용 버전의 Code를 모두 만드는 방법은 없을까? 이를 위해 마이크로 소프트에서는 몇 가지 유용한 매크로를 제공하고 있다.

1) TCHAR.h 파일을 포함하자.

```
//TCHAR.h
#ifdef UNICODE
```

```
    typedef wchar_t TCHAR;
    #define _TEXT(X)  L##X
    #define _T(X)     _TEXT(X)
    #define _tcslen   wcslen
#else
    typedef char TCHAR;
    #define _TEXT(X)  X
    #define _T(X)     _TEXT(X)
    #define _tcslen   strlen
#endif
```

2) 범용타입을 사용하여 변수를 선언하자.

DBCS	Unicode	범용타입
Char	wchar_t	TCHAR
char *	wchar_t *	LPTSTR
const char *	const wchar_t *	LPCTSTR

범용타입은 TCHAR.h 파일에 정의된 조건부 컴파일문에 따라 환경에 따라 적절하게 타입 변환이 발생한다.

3) 상수 문자나 상수 문자열을 사용할 때는 반드시 _TEXT 키워드를 사용한다.
상수 문자나 상수 문자열 사용시 컴파일러는 해당 문자열이 DBCS 형식의 문자(문자열인지) Unicode 형식의 문자(문자열)인지를 구분할 수 있어야 한다. 이 때 해당 키워드를 사용함으로 TCHAR.h 파일에 정의된 조건부 컴파일문에 따라 적절하게 변환된다.

```
#include <tchar.h>

void main(){
        TCHAR* str = _TEXT("TEST");
        TCHAR ch = _TEXT('A');
}
```

[예제 1-7] 매크로 사용

4) 문자열을 다루는 함수들도 범용함수를 사용한다.

DBCS	Unicode	범용함수
Strlen	wcslen	_tcslen
Strcpy	wcscpy	_tcscpy
Strcat	wcscat	_tcscat
Strcmp	wcscmp	_tcscmp

1. 5 API의 유니코드 지원

MessageBox 라는 함수를 생각해 보자. 이 함수의 2번째 3번째 파라미터는 문자열이다. 그렇다면 char* 형태로 전달될 수 있고 wchar* 형태로 전달 될 수 도 있을 것이다. C++이라면 오버로딩을 통해 동일함수 명으로 처리할 수 있지만, API는 C기반이기 때문에 다른 방식이 필요하다. 결론은 우리가 사용한 MessageBox라는 함수는 매크로 일뿐, 진짜 함수는 2가지 버전이 있다.
실제 함수는 MessageBoxA, MessageBoxW 라는 2가자의 함수이며, 매크로에서 환경에 따라 적절한 함수 호출 코드로 치환시켜 준다.

```
int WINAPI MessageBoxA(HWND hwnd, LPCSTR lpText, LPCTSTR lpC, UNIT uType);
int WINAPI MessageBoxW(HWND hwnd, LPCWSTR lpText, LPCWSTR lpC, UINT uType);

#ifdef UNICODE
#define MessageBox MessageBoxW
#else
#define MessageBox MessageBoxA
#endif //!UNICODE
```

또한 함수들이 사용하는 구조체들도 ANSI 버전과 UNICODE 버전이 각각 존재하며 UNICODE 매크로 상수에 따라 실제 정의가 달라진다.

1. 6 코드 변환

때때로 ANSI 문자열과 UNICODE를 변경할 경우들이 발생한다. 일부 API 함수들은 특정 타입의 문자열을 요구하며 COM 오브젝트나 쉘 라이브러리의 일부 요소는 반드시 유니코드 인수를 넘겨야 제대로 동작한다. 예를 들어 WinExec 함수는 대체되는 CreateProcess라는 새 함수가 별도로 제공되기 때문에 별도로 유니코드 버전이 제공되지 않는다. 하지만 WinExec는 그 간편함 때문에 아직도 많이 사용되며 이 함수를 쓰기 위해서는 인수는 반드시 ANSI로 넘겨야 한다.

아래의 두 가지 함수를 사용하면 유니코드를 ANSI로 혹은 ANSI를 유니코드 형태로 변환할 수 있다.

```
int MultiByteToWideChar(UINT CodePage, DWORD dwFlags, LPCSTR lpMultiByteStr,
    int cbMultiByte, LPWSTR lpWideCharStr, int cchWideChar);

int WideCharToMultiByte(UINT CodePage, DWORD dwFlags, LPWSTR lpWideCharStr ,
    int cchWideChar , LPCSTR lpMultiByteStr, int cbMultiByte, LPCSTR lpDefaultChar,
    LPBOOL lpUsedDefaultChar);
```

아래 코드는 간단한 사용 예이다.

```
#include <Windows.h>
#include <locale.h>
void main(){
        //ANSI ==> UNICODE
        _wsetlocale(LC_ALL, L"Korean");
        char ansi[] = "유니코드 문자열로 변환";
        wchar_t atow[250];
        MultiByteToWideChar(CP_ACP, 0, ansi, -1, atow, 250);
        fwprintf(stdout, L"%sWn", atow);

        char wtoa[250];
        WideCharToMultiByte(CP_ACP, 0, atow, -1, wtoa, 250, NULL, NULL);
        printf("%sWn", wtoa);
}
```

유니코드 문자열로 변환

유니코드 문자열로 변환

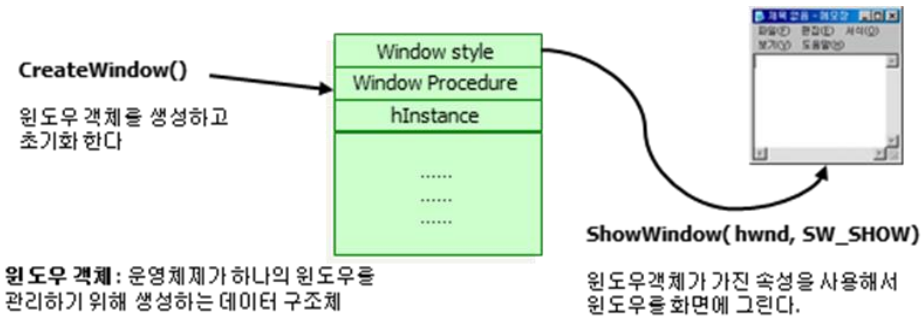
[예제 1-8] 코드 변환

2. Object

2. 1 Window Object

CreateWindow() 함수를 사용하면 윈도우를 만들 수 있다. 그럼 윈도우의 정확한 정체는 무었일까? 사용자가 윈도우를 작게 했다가 다시 복구 했을 때 윈도우는 항상 동일한 위치에 동일한 크기를 가지고 동일한 스타일로 복구되는 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 구조체 형식의 변수를 하나 생성한다. 그리고 Create Window() 인자로 초기화 한다. 이 변수를 흔히 윈도우 객체라고 부른다. 이 윈도우 객체의 내용으로 화면에 윈도우를 그리는 함수가 ShowWindow() 이다.



결국 윈도우 핸들도 윈도우 객체를 가리키는 번호이다.

2. 2 Window Object Handle과 Kernel Object Handler

다음 그림을 보자.

- A 프로세스가 윈도우를 생성했다. 윈도우 핸들은 0x0001234라 가정하자.
- A 프로세스가 B 프로세스에게 SendMessage()를 이용하여 윈도우 핸들을 전달 했다.
- B 프로세스는 MoveWindow()를 이용하여 A에서 전달받은 윈도우 핸들을 가지고 해당 윈도우를 움직이고자 한다.

과연 B는 A가 만든 윈도우를 움직일 수 있을까?

답은 움직이다. 이유는 윈도우의 모든 핸들은 모든 프로세스에 대해 전역적이다. 즉 모든 프로세스들이 특정 윈도우의 핸들값을 획득하면 동일한 핸들값을 얻게 된다는 의미이다. 따라서 다른 누군가에게서 윈도우의 핸들값을 전달 받았다고 해서 그 핸들값이 의미가 없는 것은 아니다.

이번에는 Kernel Object 중 하나인 파일을 생각해 보자.

- A 프로세스가 파일을 생성했다. 파일의 핸들은 0x000007F0 이라고 가정하자.
- A 프로세스가 B 프로세스에게 SendMessage() 함수를 사용해서 파일의 핸들을 전달 했다.
- B 프로세스에서 A에서 전달 받은 파일의 핸들을 사용해서 WrtieFile()로 임의의 data를 파일에 쓰려고 한다.

B는 A가 만든 파일에 data를 쓸 수 있을까?

답은 쓸 수 없다. 파일은 모든 프로세스에 대해 상대적인 핸들을 사용한다.
동일한 핸들임에도 불구하고 Window의 핸들은 가능하고 Kernel의 핸들은 불가능한 이유는 무었일까? 아래에서 정리해 보자.

윈도우 시스템이 제공하는 객체(Object0는 크게 3가지로 분류할 수 있다.

객체	설명
User 객체	윈도우와 관련된 작업을 수행한다. 프로세스에 전역적 핸들을 갖는다. 파괴하는 함수는 Destoy 로 시작한다. 예) Caret, Cursor, Menu, Window, Icon, Accelerator Table …
GDI 객체	그래픽과 관련된 작업을 수행한다. 프로세스에 지역적 핸들을 갖는다. 파괴하는 함수는 대부분 Delete 로 시작한다. 예) Bitmap, Brush, DC, Pen, Region …
Kernel 객체	파일, 메모리, 프로세스, IPC 등과 관련된 작업을 수행한다. 프로세스에 상대적 핸들을 갖는다. CloseHandle() 함수로 파괴한다.(실제 파괴가 아니고 참조개수가 감소한다.- 커널 오브젝트의 개념을 이해해야 한다.) 예) Process, Thread, Pipe, Mutex, Event, Timer …

User객체와 GDI 객체는 프로세스가 소유한다. 따라서 프로세스가 종료되면 해당 프로세스에서 생성된 모든 객체들은 자동으로 소멸된다. 단, User 객체는 전역적 핸들값으로 다른 프로세스에서도 접근이 가능하다는 성질을 갖고 있으며, GDI 객체는 지역적 핸들값으로 해당 프로세스 내부에서만 사용하기 위한 목적으로 생성하고 소멸시킨다.

반면, Kernel 객체는 Kernel에 의해 소유된다. 따라서 프로세스가 종료되더라도 해당 프로세스에서 생성된 모든 Kernel 객체들은 종료되지 않는다. Kernel에 의해 관리되기 때문에 생성한 부모와는 소멸 주기가 동일하지 않게 된다.

즉, 생성한 프로세스는 해당 객체를 사용할 수 있는 핸들값을 가지고 있을 뿐이고, 그 핸들값으로 관련 API 함수를 통해 접근할 수 있을 뿐이다. 다음 항목에서 자세히 살펴보자.

2. 3 Kernel Object 와 Handle Table

모든 프로세스는 커널 오브젝트의 정보를 담는 핸들 테이블을 갖고 있다.
만약, 프로세스가 커널 오브젝트를 생성한다면 아래의 흐름에 따라서 핸들값을 소유하게 된다.

- A라는 프로세스가 CreateXXX 함수를 호출하여 B라는 커널 객체를 생성을 요구했다고 가정하자.
- B 오브젝트가 Kernel 메모리 영역에 생성된다.
  오브젝트 생성시 Kernel 객체의 종류에 따라 다양한 정보를 담게 되지만, 아래의 내용은 모든 객체들의 공통된 정보이다.

Name	오브젝트별 구분할 수 있는 Key값 예) Process : ProcessID 동기화 객체 : 생성시 명명한 문자열
Usage Count	해당 커널 객체를 접근할 수 있도록 생성된 핸들값의 개수 모든 프로세스들의 핸들테이블에 저장되어 있는 해당 객체의 핸들값 의 총 합
SECURITY_ATTRIBYTE	보안 속성 값 모든 커널 오브젝트는 생성시 반드시 해당 정보를 파라미터로 받게 된다.
Signal	상태값 객체에 따라 다르지만 모든 객체들은 Signal 과 nonSignal값을 갖는다. 예) 프로세스는 살아 있을 경우 nonSignal 죽으면 Signal 된다.

- A 프로세스의 핸들 테이블에는 B 오브젝트를 접근할 수 있는 핸들의 정보가 저장된다. 따라서 B의 커널 객체의 정보 중 Usage Count 정보가 +1 증가하게 된다.
  (단, Process 와 Thread는 +2 증가 한다.(생성한 프로세스와 자신의 프로세스의 핸들 테이블에 모두 등록된다.)
- A라는 프로세스가 B라는 커널 객체를 생성할 때 사용했던 CreateXXX 함수가 리턴되며 이 때 B객체의 핸들값을 획득하게 된다.

이 후부터 A 프로세스는 B객체의 핸들값으로 해당 객체를 제어할 수 있다. 만약 더 이상 B객체를 제어하지 않으려면 해당 핸들값을 CloseHandle()하면 된다. 이 때 아래와 같은 흐름을 갖는다.

- A라는 프로세스가 CloseHandle(B)을 호출하게 되면 자신의 핸들테이블에 해당 핸들 값이 존재하는지 확인한다.
- 만약, 존재한다면 B의 커널오브젝트의 정보 중 Usage Count 값을 -1 감소시킨다. 이 때 Usage Count 값이 0이 되면 해당 커널 객체는 소멸되게 된다.
- 그 후 자신의 핸들테이블에서 해당 객체의 핸들값을 제거한다.

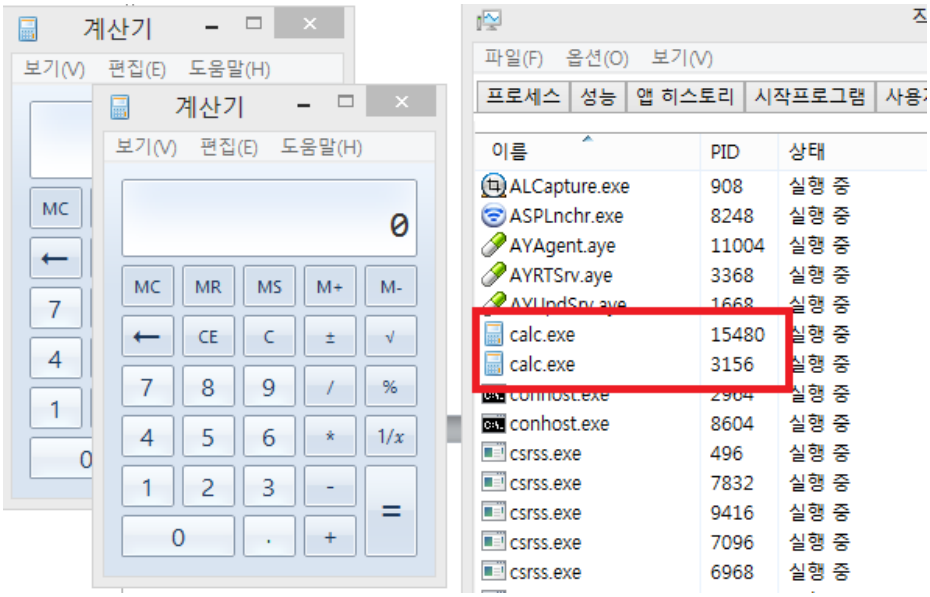
이 후부터 A 프로세스는 B객체를 더 이상 제어할 수 없게 된다.

결국 상대적 핸들이라는 개념은 프로세스가 다른 커널 객체를 제어하길 원한다면 반드시 자신의 핸들테이블에 제어하고자 하는 커널 객체의 핸들 정보를 가지고 있어야만 제어가 가능하다는 의미이다.

3. Process

3. 1 Process

프로세스(Process)란 실행중인 프로그램을 말한다. 예를 들어 calc.exe 라는 실행 파일이 있을 때 이 파일이 실행되어 메모리에 적재되면 계산기 프로세스가 된다. 만약 사용자가 두 개의 계산기 프로그램을 실행시켰다면 이 둘은 같은 프로그램이지만 각각 다른 프로세스로 인식된다. 구분되는 키값은 프로세스의 ID이다.



좌측에 보이는 이미지 처럼 2개의 계산기를 실행시키면 우측의 작업관리자에서 서로 다른 PID(ProcessID) 를 가진 두 개의 계산기 프로세스를 확인할 수 있다.

운영체제는 실행된 프로그램을 프로세스 단위로 관리한다. 프로세스는 각각 4GB(운영체제마다 달라질 수 있다.)의 주소 공간과 파일, 메모리, 스레드 등의 객체를 소유하며 프로세스가 종료될 때 프로세스가 소유한 모든 자원은 운영체제에 의해 파괴된다. 모든 것은 프로세스에 의해 소유된다.

프로세스는 실행중인 프로그램이지만 실제로 작업을 하는 주체는 아니다. 작업은 프로세스 내의 스레드(Thread)가 담당한다. 프로세스는 단지 메모리상에 존재하기만 할 뿐이며 실행과 동시에 스레드를 하나 만들고 스레드를 호출함으로써 스레드에게 모든 작업을 맡긴다. 정리하자면 프로세스는 스레드를 담는 껍데기이며 실제 일을 하는 것은 스레드이다.

프로세스는 최소 한 개 이상의 스레드를 가진다. 프로세스와 동시에 자동으로 만들어 지는 스레드를 주 스레드(Primary Thread)라 하며 이외에 필요에 의해 여러 개의 스레드를 더 만들어 사용할 수 있다.

3. 2 Process 생성

한 프로그램에서 다른 프로그램을 실행하고자 할 때는 Win32 API가 제공하는 프로세스 생성 함수를 사용한다. **프로세스를 생성하는 가장 쉽고도 간단한 함수는 WinExec** 함수이다.//유니코드로바꾸기도하는 것

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

보다시피 원형도 아주 간단하다. lpCmdLine 인수로 실행하고자 하는 프로그램의 이름을 전달하되 완전 경로를 줄 수도 있다. 실행 파일명만 주었을 때는 다음 순서대로 실행 파일의 위치를 검색하여 발견된 실행 파일을 실행한다.

1) 프로그램이 실행된 디렉토리
2) 현재 디렉토리
3) 시스템 디렉토리
4) 윈도우 디렉토리
5) PATH 환경변수가 지정하는 디렉토리들

간단한 사용 예를 보자.

```
#include <windows.h>

void main(){

        // Process 생성 함수

        WinExec("calc.exe", SW_SHOW);

        ShellExecute(0, TEXT("open"), TEXT("calc.exe"), 0, 0, SW_SHOW);

}//프로그램을 실행하거나 원하는파일을 탐색기로 열 때
```

3.2.1 CreateProcess 인자

```
BOOL CreateProcess(     LPCTSTR lpApplicationName, 실행할 exe 파일의 이름
                        LPTSTR lpCommandLine, 생성될 자식 프로세스에 전달할
Command line 을 지정한다 파일의 이름을 주어야 한다
                        LPSECURITY_ATTRIBUTES lpProcessAttributes,
                        LPSECURITY_ATTRIBUTES lpThreadAttributes,
                        BOOL bInheritHandles,
                        DWORD dwCreationFlags,
                        LPVOID lpEnvironment,
                        LPCTSTR lpCurrentDirectory,
                        LPSTARTUPINFO lpStartupInfo,
          LPPROCESS_INFORMATION lpProcessInformation );
```

- lpApplicationName

실행할 exe 파일의 이름, 전체경로를 주지 않을 경우 프로세스의 현재 드라이버와 현재 디렉토리를 사용한다. 열려진 path 검색을 하지 않는다.
이 파라미터로 NULL을 줄 수 있다. 하지만 이 경우 반드시 lpCommandLine 인자에 실행할 파일의 이름을 주어야 한다.

- lpCommandLine

생성될 자식 프로세스에 전달할 Command line을 지정한다. 첫 번째 인자가 NULL일 경우 사용 가능하다. 만약, 전체 경로를 지정하지 않을 경우는 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 & lpThreadAttributes

생성될 자식 프로세스의 프로세스 객체의 보안 속성과 생성될 자식 프로세스의 스레드 객체의 보안 속성이다.

- bInheritHandles

Object Table의 핸들을 자식 프로세스에게 상속할 것인가를 지정한다. TRUE이면 Object H Table의 핸들 중 상속 가능한 핸들만 자식 프로세스에게 상속된다.
(모든 프로세스는 커널Object의 정보를 담는 핸들테이블을 갖고있다.)

- dwCreationFlags

프로세스의 우선 순위 클래스와 *생성 flag*를 지정한다. 아래와 같은 상수 값을 지정할 수 있다. 일반적으로 대부분의 프로세스는 NORMAL+PRIORITY_CLASS 로 지정한다.

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

생성할 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, 표준 입추력 핸들, 윈도우의 크기와 위치등을 지정할 수 있다.

- lpProcessInformation

PROCESS_INFORMATION 객체의 주소, 생성된 프로세스와 스레드 ID와 핸들을 얻는다.

3.2.2 STARTUPINFO 구조체

CreateProcess 함수는 프로세스 생성에 관한 세밀한 제어를 할 수 있는 고급 함수이다. WinExec는 단순히 프로세스를 생성하는 정도지만 CreateProcess 함수는 프로세스 생성과 동시에 프로세스의 여러 가지 속성을 지정할 수 있다. 우선 아홉 번째 인수인 lpStartupInfo 에 대해 알아보자.

```
typedef struct _STARTUPINFO {
    DWORD cb;
    LPTSTR lpReserved;
    LPTSTR lpDesktop;
    LPTSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    WORD wShowWindow;
    WORD cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
}STARTUPINFO, *LPSTARTUPINFO;
```

- cb

구조체 크기를 전달한다.

- lpReserved
예약됨, CreateProcess() 를 호출하기전에 반드시 0으로 설정하고 호출해야 한다. *//왜?*

- 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 Application을 실행할 경우 NULL로 전달해야 한다.

- dwX, dwY
Console Application을 실행할 경우 Console 창의 위치를 지정한다.
GUI Application을 실행할 경우 CreateWindow() 함수의 CW_USEDEFAULT 값으로 전달된다. dwFlags 인자로 STARTF_USEPOSION 를 지정할 때만 적용된다.

- dwXSize, dwYSize
Console Application을 실행할 경우 Console 창의 크기를 지정한다.
GUI Application을 실행할 경우 CreateWindow() 함수의 CW_USEDEFAULT값으로 전달된다. dwFlags 인자로 STARTF_USESIZE 를 지정할 때만 적용된다.

- dwXCountChars, dwYCountChars
Console Application을 실행할 때 새로운 Console 창의 스크린 버퍼의 너비와 높이를 지정한다.
dwFlags 인자로 STARTF_USECOUNTCHARS 를 지정할 때만 적용된다.

- dwFillAttribute
dwFlags, 의 인자로 STARTF_USEFILLATTRIBUTE가 지정할 때만 적용된다.
Console Application을 실행할 때 새롭게 생성되는 Console 창의 글자 색과 배경색을 지정한다. 다음과 같은 상수 값이 지정될 수 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_GREEN | BACKGROUND_BLUE

- wShowWindow
dwFlags 의 인자로 STARTF_USESHOWWINDOW 가 지정될 때만 적용된다.
GUI Application 에서 첫 번째로 ShowWindow() 를 호출할 때 2번째 인자로 지정된다.

- cbReserved2, lpReserved2
C Runtime 에 의해 예약되어 있다. 반드시 0을 지정한다.

- hStdInput, hStdOutput, hStdError
새롭게 생성되는 프로세스의 표준 입력, 출력, 에러 핸들을 지정한다.
dwFlags 의 인자로 STARTF_USESTDHANDLES 가 지정될 때만 적용된다.

- dwFlags
STARTUPINFO 구조체중에서 어떤 맴버를 적용할 지를 지정한다.
아래와 같은 값들이 OR 비트 연산을 통해 지정될 수 있다.

value	Meaning
STARTF_FORCEONFEEDBACK	
STARTF_FORCEOFFFEEDBACK	
STARTF_RUNFULLSCREEN	Console Application을 Full Screen Mode 로 실행되게 한다.
STARTF_USECOUNTCHARS	dwXCountChars, dwYCountChars 멤버를 적용한다.
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.

```
#include <windows.h>

void main(){

        TCHAR name[] = TEXT("calc.exe");
```

```
        PROCESS_INFORMATION pi;
        STARTUPINFO                         si = { sizeof(si) };
        BOOL b = CreateProcess(0, name, 0, 0,
                                FALSE, NORMAL_PRIORITY_CLASS, 0, 0,
                                &si, &pi);
        if( b )   {
                WaitForInputIdle(pi.hProcess, INFINITE);
                CloseHandle(pi.hProcess);
                CloseHandle(pi.hThread);
        }
        HWND hwnd = FindWindow(0, TEXT("계산기"));
        if( hwnd != 0)
        MessageBox(NULL, TEXT("계산기를 생성했습니다."), TEXT("알림"), MB_OK);
}
```

[예제 3-2] CreateProcess

먼저 VS2012 버전에서는 CreateProcess 2번째 인자형에 문자열 포인터를 사용할 경우 실행 시 Run-time error 가 발생한다.
이유는 만약 calc.exe 실행시킬 경우 뒤에 추가로 명령행 인자를 전달할 수 있는데 이러한 것을 CreateProcess 함수 내부에서 파싱해서 처리할 경우 문자열 상수의 포인터를 가지고 파싱을 시도하면 메모리 엑세스 에러가 나기 때문이다.
따라서 반드시 2번째 인자는 항상 문자열 배열 형태로 넘겨야 한다.
잘못 전달한 예)

```
CreateProcess(0, TEXT("calc.exe"), ... );   // 실행시 에러

LPCTSTR cmd = TEXT("calc.exe");
CreateProcess(0, cmd, ... );                // 실행시 에러
```

위의 예제를 분석해 보자.
먼저 CreateProcess() 를 호출하여 계산기를 실행하였다. CreatProcess() 함수는 프로세스를 생성한 후 곧바로 리턴하므로 이 함수가 리턴된 직후에는 메모장이 아직도 초기화 중이며 메인 윈도우가 만들어져 있지 않다. 따라서 해당 함수 호출 즉시 FindWindow() 함수로 계산기 윈도우를 찾을 수 없다.

```
DWORD WaitForInputIdle(HANDLE hProcess, DWORD dwMilliseconds);
```

위 함수는 첫 번째 인자로 전달된 프로세스가 사용자 입력을 받을 수 있을 때까지, 즉 초기화가 완료될 때까지 또는 두 번째 인자의 시간이 경과할 때 까지 대기한다.

3. 3 Process 종료

프로세스가 자신을 종료하는 가장 간단하고도 일반적인 방법은 WM_CLOSE 메시지를 보내거나 DestroyWindow 로 메인 윈도우를 파괴하는 것인데 이 때 WM_DESTROY 에서 PostQuitMessage를 호출하여 메시지 루프를 종료한다. 이런 방법들에 의해 WinMain이 리턴되면 *C 런타임 코드는* 다음 함수를 호출하여 정리작업을 수행한다.

3.3.1 ExitProcess

void ExitProcess(UINT nExitCode);

이 함수가 호출되면 프로세스는 정리작업에 들어가 즉각 종료된다. 프로세스가 종료될 때는 다음 일련의 작업이 이루어진다.

- 프로세스와 연결된 모든 DLL을 종료시키기 위해 각 DLL의 DllMain 함수가 호출되며 DLL 들은 스스로 정리작업을 한다.
- 모든 열려진 핸들을 닫는다.
- 실행중인 모든 스레드를 종료한다.
- 프로세스 커널 객체와 스레드 객체는 신호상태가 되며 이 객체를 기다리는 다른 프로세스는 대기상태를 해제할 수 있다.
- 프로세스의 종료코드는 STILL_ACTIVE 에서 ExitProcess 가 지정하는 종료 값이 된다.

아래 코드는 ExitProcess() 함수 및 WM_CLOSE 사용 예이다.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
        switch( msg )       {
        case WM_LBUTTONDOWN:
                SendMessage(hwnd, WM_CLOSE, 0, 0);//비큐메시ㅣㅈ
```

```
                return 0;
        case WM_RBUTTONDOWN:
                ExitProcess(0);
                return 0;
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 3-3] ExitProcess

마우스 왼쪽버튼을 누를 때 비 큐 메시지를 이용하여 프로그램을 종료시키고, 마우스 오른쪽 버튼을 누를 때 ExitProcess 함수를 이용하여 종료한다.

ExitProcess 가 PostQuitMessage 와 다른 점은 곧바로 정리작업에 들어가 프로세스를 종료하기 때문에 ExitProcess 아래의 코드는 결코 실행되지 않는다는 점이다.

3.3.2 TerminateProcess

또한 아래의 함수도 프로세스를 종료한다.

BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);

이 함수는 ExitProcess 에 비해 종료 대상이 되는 프로세스의 핸들을 인수로 가지므로 자기 자신이 아닌 다른 프로세스를 강제로 종료시킬 수 있다. 물론 자신의 프로세스 핸들을 첫 번째 인자로 전달하면 자기 자신을 종료한다.

그러나 이 함수는 ExitProcess 보다 훨씬 더 위험하다. TerminateProcess 함수가 호출될 때 ExitProcess 와 동일한 정리작업이 수행되나 단 연결된 DLL에게 종료사실이 통지되지 않는다. 만약 DLL에서 대량의 메모리를 할당해 놓았거나 파일을 저장하지 않은 채로 가지고 있는 상황에서 이 함수로 강제로 프로세스를 종료해 버린다면 정보를 잃어버릴 수 있다. 따라서 어쩔 수 없이 강제로 종료시킬 경우에만 사용한다.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
	switch( msg )
	{
	case WM_LBUTTONDOWN:
		TerminateProcess(GetCurrentProcess(),0);
		return 0;
	case WM_RBUTTONDOWN:
		{
			TCHAR buf[20] = TEXT("calc.exe");
			STARTUPINFO si = {sizeof(STARTUPINFO),};
			PROCESS_INFORMATION pi;

	CreateProcess(NULL, buf, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
			Sleep(5000);
			TerminateProcess(pi.hProcess, 0);
			GetExitCodeProcess(pi.hProcess, &code);
			CloseHandle(pi.hProcess);
			CloseHandle(pi.hThread);
		}
		return 0;
	case WM_DESTROY:
		PostQuitMessage(0);
		return 0;
	}
	return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 3-4] TerminateProcess

TerminateProcess 함수는 비동기 함수이다. 즉, 이 함수가 리턴 할 때 해당 하는 프로세스가 바로 죽은 것은 아니다.
따라서 해당 함수 호출 후 종료 코드값을 획득하는 GetExitCodeProcess 함수를 호출 하면 실제 STIL_ACTIVE(Process 가 살아있을 때의 종료코드 값) 값을 획득할 수도 있다.

만약, 프로세스가 완전히 종료될 때 까지 대기하려면 아래와 같은 코드를 구성하면 된다.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
          static HANDLE hProcess;
          switch( msg )       {
          case WM_LBUTTONDOWN:
                    {
                    TCHAR buf[20] = TEXT("calc.exe");
                    STARTUPINFO si = {sizeof(STARTUPINFO),};
                    PROCESS_INFORMATION pi;

          CreateProcess(NULL, buf, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
                    hProcess = pi.hProcess;
                    }
                    return 0;
          case WM_RBUTTONDOWN:
                    {
                    TerminateProcess( hProcess, 0 );
                    DWORD ret = WaitForSingleObject( hProcess, 5000);
                    if ( ret == WAIT_TIMEOUT)
                    MessageBox(hwnd, TEXT("5초 경과"), TEXT("알림"), MB_OK);
                    else if ( ret == WAIT_OBJECT_0)
                    MessageBox(hwnd, TEXT("계산기종료"), TEXT("알림"), MB_OK);
                    }
                    return 0;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    return 0;
          }
          return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 3-5] WaitForSingleObject

3.3.3 WaitForSignleObject

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 사용하면 된다.

```
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 INFINITE 를 전달하면 시그널 될 때 까지 무한 대기한다.

```
WaitForSingleObject( hProcess, 0);
  ➔ hProcess 의 상태의 시그널 상태를 조사한다.(대기하지 않는다.)

WaitForSingleObject( hProcess, 5000);
  ➔ hProcess 가 시그널 되거나, 5 초가 지나면 리턴 된다.

WaitForSingleObject( hProcess, INFINITE);
  ➔ hProcess가 시그널 될 때 까지 무한히 대기한다.
```

WaitForSingleObject 는 다음 중 1개의 값을 리턴한다.

리턴 값	의미
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 Release 하지 않고 종료한 경우
WAIT_FAILED	함수 호출에 실패한 경우(주로 잘못된 핸들)

3.3.4 정리

프로세스는 다음의 4가지 방법으로 종료 할 수 있다.

- 주 스레드가 *엔트리 포인트 함수*에서 리턴한다.(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 TerminateProcess 함수 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 될 때 아래와 같은 일을 수행한다.

* 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.

3. 4 프로세스 ID

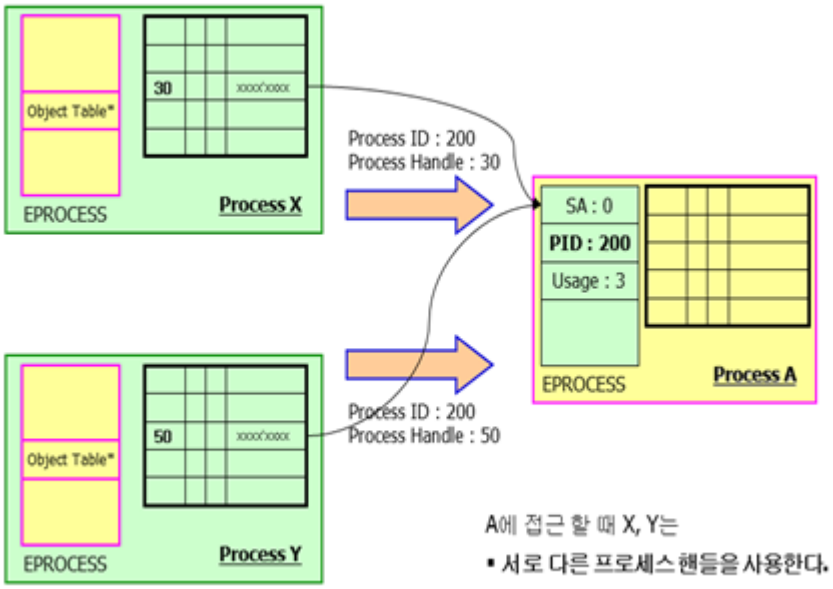
모든 프로세스는 자신만의 ID를 갖는다. 이 ID는 시스템내의 모든 프로세스에서 유일하다. 만약 메모장을 2번 실행하게 되면 서로 다른 프로세스 ID를 갖게 된다. //계산기

아래 그림을 살펴 보자. ProcessA의 정보를 좌측에 있는 2개의 Process에서 접근한 예이다. 이때 Process ID는 200으로 둘 다 동일하게 갖고 있는 것을 볼 수 있다. 하지만 Process Handle 값은 서로 다르다.

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 A()에 접근하기 위해 자신의 object Table에 있는 인덱스 값을 사용한다. 결국 동일한 A 커널 오브젝트를 접근할 때 프로세스 X, Y는 서로 다른 핸들 값을 사용한다//////////////.//

만약, 프로세스 X가 자신이 가지고 있던 ProcessA의 핸들값을 프로세스 Y에게 전달하여도 프로세스 Y는 절대로 Process A를 제어할 수 없다. 그 이유는 해당 핸들값이 자신의 Object Table에 존재하지 않기 때문이다.

즉, 프로세스는 커널 오브젝트를 제어하기 위해서는 자신의 Object Table에 제어하고자 하는 커널 오브젝트의 핸들이 등록되어 있어야만 가능하다.



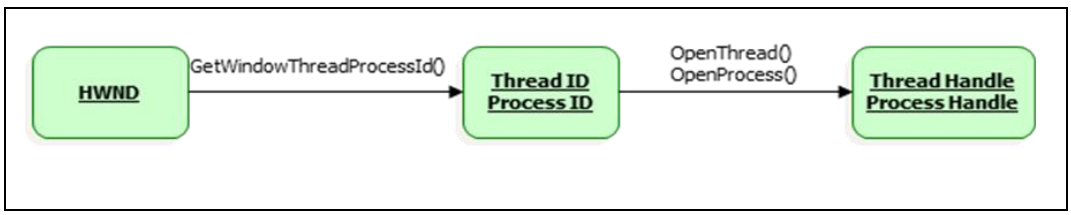
프로세스 ID는 프로세스를 구분하기 위한 용도일 뿐 해당 값으로는 프로세스를 제어할 수 없다. 그렇다면 프로세스 ID를 가지고 무었을 할 수 있을까?
프로세스 ID는 프로세스 핸들을 구하는 용도로 사용할 수 있다.

또한 윈도우 핸들을 알 때 아래 함수를 사용하면 해당 윈도우를 만든 프로세스의 ID를 얻을 수 있다.

DWORD GetWindowThreadProcessId(HWND hWnd, LPDWORD lpdwProcessId);

또한 아래 함수를 사용하면 프로세스 ID를 가지고 프로세스의 핸들을 얻을 수 있다.

HANDLE OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle,
DWORD dwProcessId);



아래 예제는 윈도우 핸들로 Process handle을 얻고 있다.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
	switch( msg )	{
	case WM_LBUTTONDOWN:
	if ( GetCapture() == hwnd )
	{
		ReleaseCapture();
		POINT pt;
		// 현재 마우스 포인터 얻기(스크린 좌표)
		GetCursorPos(&pt);
		// 윈도우 핸들 얻기..
		HWND h = WindowFromPoint( pt );
		// 해당하는 윈도우(h)를 만든 프로세스의 ID를 구한다.
		DWORD pid;
		DWORD tid = GetWindowThreadProcessId( h, &pid );
		// 프로세스 ID를 가지고 프로세스 핸들을 얻는다.(Table에 항목을만든다)
		HANDLE hProcess = OpenProcess( PROCESS_ALL_ACCESS, 0, pid );
		// 프로세스 핸들 사용
		TerminateProcess( hProcess, 0 );
		WaitForSingleObject( hProcess, INFINITE );
		CloseHandle( hProcess);
	}
	return 0;
	case WM_DESTROY:
		PostQuitMessage(0);
		return 0;
	}
	return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 3-6] OpenProcess

OpenProcess 함수는 Process ID를 인자로 받아 Process Handle 을 반환하는 함수이다. 이 함수는 CreateProcess() 로 생성된 커널오브젝트에 접근해서 해당 오브젝트의 핸들값을 획득하는 함수로써, 실제 해당 커널 오브젝트의 사용 카운트를 증가시키고 자신의 Object Table에 커널 정보를 등록한다.

3. 5 Handle 상속

프로세스는 자식 프로세스를 생성하면서 자신의 object table에 있는 핸들을 자식 프로세스 핸들 테이블에 전달 할 수 있다. 이를 Handle 상속이라 한다.

Handle 상속을 하기 위해서는 2가지 사항이 필요한다.

- 먼저 object table에 등록된 커널 오브젝트의 핸들이 상속 가능토록 등록되어 있어야 한다.
- 두번째로 자식 프로세스 생성시 핸들 상속 여부 인자에 true 값을 전달해야 한다.

아래 코드는 상속 가능한 커널 오브젝트 생성 예이다.

```
#include <windows.h>

void main(){

	// 상속 가능한 KO 만들기

	SECURITY_ATTRIBUTES sa;

	sa.nLength          = sizeof( sa );

	sa.bInheritHandle = TRUE;    // 상속가능하게

	sa.lpSecurityDescriptor = 0; // 실제 보안정보.

	HANDLE hEvent = CreateEvent(  &sa, 0, 0, TEXT("e"));


	// 상속가능하게 바꾸기..

	SetHandleInformation( hEvent, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);

	CloseHandle( hEvent );
}
```

[예제 3-7] 상속 가능한 KO 만들기

위 코드는 상속 가능한 커널 오브젝트를 만드는 2가지 방식을 보여준다.

첫 번째는 SECURITY_ATTRIBUTES 구조체를 사용하는 방식으로 해당 구조체의 bInheritHandle 맴버에 TRUE값을 설정하고, 커널 오브텍트 생성시 해당 구조체를 전달하면 된다.

두 번째는 만약 커널 오브젝트를 상속 가능하지 않도록 생성했을 경우 SetHandleInformation 함수를 위와 같이 사용하면 상속 가능토록 변경된다.

아래 예제는 자식프로세스에게 핸들을 상속하고 자식 프로세스는 상속된 핸들로 해당 커널 오브젝트를 접근한다.

```
#include <stdio.h>
#include <windows.h>
void main(){
	HANDLE hRead, hWrite;
	CreatePipe( &hRead, &hWrite, 0, 4096);
	// 읽기 위한 핸들을 상속 가능하게 한다.
	SetHandleInformation( hRead, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
	TCHAR cmd[256];
	wsprintf( cmd, TEXT("child.exe %d"), hRead); // 명령형 전달인자 사용
	PROCESS_INFORMATION pi;
	STARTUPINFO si = { sizeof(si)};
	BOOL b = CreateProcess( 0, cmd, 0, 0, TRUE, CREATE_NEW_CONSOLE,
		0,0,&si, &pi);
	if ( b )  {
		CloseHandle( pi.hProcess);
		CloseHandle( pi.hThread);
		CloseHandle( hRead );
	}
	//-----------------------------------------
	char buf[4096];
	while ( 1 )		{
		printf("전달할 메세지를 입력하세요 >> ");
```

```
                    gets( buf );
                    DWORD len;
                    WriteFile( hWrite, buf, strlen(buf)+1, &len, 0);
          }
}
```

[예제 3-8] 핸들 상속1

```
#include <stdio.h>
#include <windows.h>
// 이 실행파일의 이름을 child.exe 변경하세요..
void main(int argc, char** argv){
          if ( argc != 2 )    {
          printf("이프로그램은 직접 실행하면 안됩니다. 부모를 실행해 주세요₩n");
                    return;
          }
          // 부모가 보내준 pipe 핸들을 꺼낸다.
          HANDLE hPipe = (HANDLE)atoi(argv[1]);
          char buf[4096];
          while ( 1 )         {
                    memset( buf, 0, 4096 );
                    DWORD len;
                    BOOL b = ReadFile( hPipe, buf, 4096, &len, 0);
                    if ( b == FALSE ) break;
                    printf("%s₩n", buf );
          }
          CloseHandle( hPipe );
}
```

[예제 3-8] 핸들 상속2

핸들 상속2는 자식 프로세스이고, 핸들 상속1은 부모 프로세스이다.
핸들 상속2(부모)는 커널 오브젝트를 생성한다.(이름없는 파이프) , 생성 후 상속 가능토록 object table의 정보를 수정한다.

그 후 자식 핸들 상속1 프로세스를 생성하는데 이 때 상속된 핸들값을 명령형 인자로 전달한다.

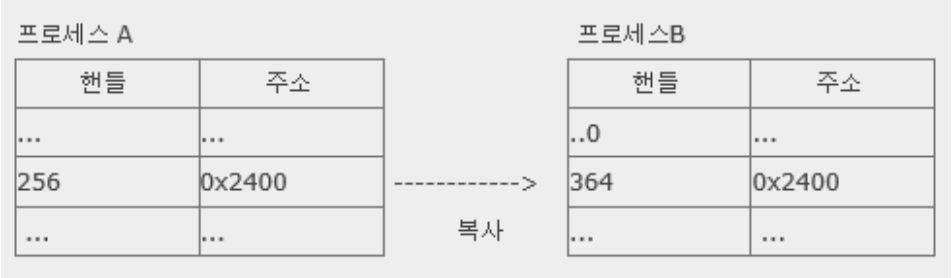
핸들 상속1(자식)은 명령형 인자로 전달된 핸들값을 가지고 이름 없는 파이프를 접근할 수 있다.

(이름 없는 파이프는 IPC 통신 관련 커널 객체로 뒷 부분에서 다룰 예정이다. 지금은 핸들 상속 개념 관점에서 해당 소스를 이해하도록 하자.)

3. 6 Handle 복사

이 전에 설명했듯이 프로세스마다 object table이 존재한다. 만약, 특정 프로세스가 파일의 핸들 값을 가지고 있다고 해도 파일의 정보가 object table에 존재하지 않는다면 해당 파일에 read, write 할 수 없다.

따라서 A라는 프로세스가 가지고 있는 커널 오브젝트의 핸들값을 B라는 프로세스에게 전달해도 B라는 프로세스는 해당 핸들값으로 제어를 할 수 없다. 만약 B라는 프로세스도 커널 오브젝트를 제어하게 하고 싶다면 B 프로세스의 object table에 커널 오브젝트의 핸들 값을 복사하면 된다.



다음은 Handle 을 복사할 수 있는 API 함수이다.

```
BOOL WINAPI DuplicateHandle(
  _In_   HANDLE hSourceProcessHandle,        -----①
  _In_   HANDLE hSourceHandle,               -----②
  _In_   HANDLE hTargetProcessHandle,        -----③
  _Out_  LPHANDLE lpTargetHandle,            -----④
```

```
  _In_   DWORD dwDesiredAccess,                -----⑤
  _In_   BOOL bInheritHandle,                  -----⑥
  _In_   DWORD dwOptions                       -----⑦
);
```

DuplicateHandle() 을 이용하면 핸들 테이블에 있는 핸들 값을 얻을 수 있다.

- hSourceProcessHandle :
복제할 핸들을 가지고 있는 프로세스를 지정한다. PROCESS_DUP_HANDLE 핸들 엑세스 권한이 있어야 한다.

- hSourceHandle :
복제할 핸들의 종류를 지정한다.

- hTatgetProcessHandle :
복제된 핸들값을 소유할 프로세스를 지정한다. PROCESS_DUP_HANDLE 액세스 권한이 필요함.

- lpTargetHandle :
복제된 핸들값을 리턴 받는다.

- dwDesiredAccess :
복제된 핸들의 접근 권한을 지정한다. 하지만 dwOption의 인자로 DUPLICATE_SAME_ACCESS을 지정할 경우 이 값은 무시된다.

- bInheritHandle :
복제된 핸들의 상속 여부를 지정한다. TRUE면 자식에 상속되고 FALSE면 상속되지 않는다.

- dwOption :

값	의미
DUPLICATE_CLOSE_SOURCE	소스 핸들을 닫는다.
DUPLICATE_SAME_ACCESS	dwDesiredAccess 를 무시한다. 중복 핸들은 소스 핸들과 동일한 액세스 권한을 가진다.

예제를 살펴보자.

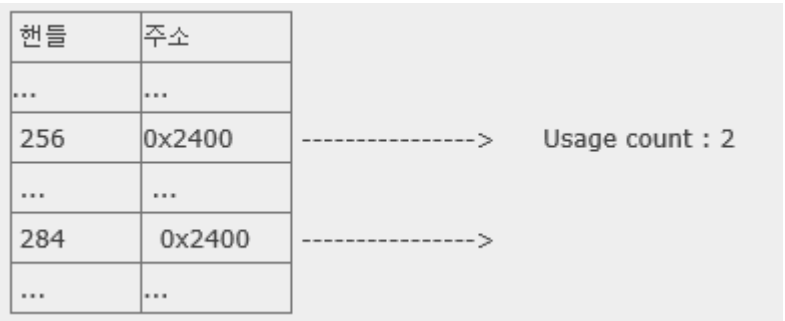
```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
          static HANDLE       hProcess1;
          static HANDLE       hProcess2;
          switch( msg )       {
          case WM_LBUTTONDOWN:
          {
          TCHAR name[] = TEXT("calc.exe");
          STARTUPINFO         si        = { sizeof(si) };
          PROCESS_INFORMATION pi;
          BOOL b = CreateProcess(NULL, name, NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS,
                              NULL, NULL, &si, &pi);

          // 가짜 복사
          hProcess1 = pi.hProcess;

          // 진짜 복사
          DuplicateHandle(  GetCurrentProcess(),  // 이 프로세스안의
                              pi.hProcess,          // 이 핸들을
                              GetCurrentProcess(),  // 이 프로세스에 복사하라.
                              &hProcess2,           // hProcess에 핸들값 저장
                              0, FALSE,             // 접근 권한, 상속
                              DUPLICATE_SAME_ACCESS);
          if( b )             {
                    CloseHandle(pi.hProcess);
                    CloseHandle(pi.hThread);
            }
        }
          return 0;
          case WM_RBUTTONDOWN:
```

```
                    TerminateProcess(hProcess2, 0);
                    return 0;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    return 0;
          }
          return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 3-9] 핸들 복사

핸들 복사는 진짜 복사와 가짜 복사로 구분할 수 있다. 진짜복사란 실제 커널 오브젝트의 사용카운트를 증가시키고 object table 에 해당 핸들이 복사되는 것이고, 가짜 복사는 단순히 핸들의 값을 복사하는 것이다. 따라서 가짜 복사는 원본 핸들값을 closeHandle() 하면 복사된 핸들 값을 사용할 수 없다. 하지만 진짜 복사된 핸들은 원본 핸들값을 closeHandle() 해도 복사된 핸들값을 사용할 수 있다.

CreateProcess의 pi.hProcess 를 가짜 복사하는 코드를 볼 수 있다.
Duplicatehandle 을 이용하여 자신의 프로스의 object table에 있는 계산기의 핸들값을복사 하는 것을 볼 수 있다.



다음 예제를 살 펴보자 2개의 프로그램으로 구성되는 데 첫 번째 예제의 코드에서 두 번째 예제의 코드로 파일의 핸들 값을 복사하고 있다.

```
#include <windows.h>

void main(){
```

```
        HANDLE hFile = CreateFile( TEXT("a.txt"), GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        0, // 보안
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

        printf( "생성된 화일 핸들(Table Index) : %xWn", hFile );
        HWND hwnd = FindWindow( 0, TEXT("B"));
        // hwnd을 만든 프로세스의 ID를 구한다.
        DWORD pid;
        DWORD tid = GetWindowThreadProcessId( hwnd, &pid );

        // 프로세스 ID를 가지고 PROCESS 핸들을 얻는다.
        HANDLE h;
        HANDLE hProcess = OpenProcess( PROCESS_ALL_ACCESS, 0, pid );

        // A의 Table의 내용을 B의 Table 에 복사 해준다.
        DuplicateHandle( GetCurrentProcess(), hFile, // source
                                        hProcess,   &h,    // target
                                        0, 0, DUPLICATE_SAME_ACCESS);
        printf("B에 복사한 핸들(Table index) : %xWn", h );
        SendMessage( hwnd, WM_USER+100, 0, (LPARAM) h );
        CloseHandle( hFile );
}
```

[예제 3-10] 프로세스간 핸들 복사1

```
#include <windows.h>
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
        switch( msg )       {
        case WM_USER + 100:
                {
                HANDLE hFile = (HANDLE)lParam;
```

```
                    char s[256] = "hello";
                    DWORD len;
                    BOOL b = WriteFile( hFile, s, 256, &len, 0);
                    if ( b == FALSE )
                    {
                    TCHAR buf[256];
                    wsprintf(buf, TEXT("전달된 핸들 : %x Wn실패 : %d"), hFile,
                                GetLastError() );
                    MessageBox( 0, buf, TEXT(""), MB_OK);
                    }
                    else
                    {
                              MessageBox( 0, TEXT("성공"), TEXT(""), MB_OK);
                              CloseHandle( hFile );
                    }
                    }
                    return 0;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    return 0;
          }
          return DefWindowProc( hwnd, msg, wParam, lParam);
}
```

[예제 3-10] 프로세스간 핸들 복사2

위 코드는 2개의 프로그램이 구동되는 소스로서 프로세스간 핸들 복사2 프로그램을 먼저 실행시키고, 프로세스 간 핸들 복사1 프로그램을 실행시키면 된다.

3. 7 Processs 열거

작업 관리자를 보면 현재 실행중인 프로세스 목록을 조사해 보여준다. 응용 프로그램 수준에서 이런 작업을 해야 할 경우는 그리 흔하지 않지만 원한다면 할 수도 있다. 실행중인 프로세스의 목록을 얻는 방법에는 Tool Help 라이브러리의 스냅샷을 사용하는 방법과 PSAPI.DLL에 의해 제공되는 EnumProcess 함수를 사용하는 방법이 있다.

사용 예는 하단의 Sample을 확인해 보기 바란다.

```
#include <windows.h>
#include <TLHelp32.h>
#include <stdio.h>
void main(){
          HANDLE    hSnap = CreateToolhelp32Snapshot(
                    TH32CS_SNAPPROCESS, 0);       // process ID
          if( hSnap == 0 )
                    return ;
          PROCESSENTRY32      ppe;
          BOOL      b = Process32First(hSnap, &ppe);
          while ( b )
          {
                    printf("%04d %04d %sWn", ppe.th32ProcessID,
                                        ppe.th32ParentProcessID,
                                        ppe.szExeFile);
                    b = Process32Next( hSnap, &ppe );
          }
          CloseHandle(hSnap);
}
```

[예제 3-11] Process 열거1

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include "psapi.h"
void PrintProcessNameAndID( DWORD processID){
```

```
        TCHAR szProcessName[ MAX_PATH ] = TEXT("unknown");
        // 프로세스의 핸들 얻기
        HANDLE hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
           PROCESS_VM_READ,   FALSE, processID);
        // process 이름 가져오기
        if( NULL != hProcess)         {
                HMODULE hMod;
                DWORD     cbNeeded;
                if( EnumProcessModules( hProcess, &hMod, sizeof( hMod), &cbNeeded))
                {
        GetModuleBaseName( hProcess, hMod, szProcessName, sizeof( szProcessName));
                }
                else
                        return;
        }
        else return;
        //print
        _tprintf(TEXT("%s ( PROCESS ID : %u )₩n"), szProcessName, processID);
        CloseHandle(hProcess);
}
void main()
{
        // process list 가져오기(id값)
        DWORD aProcess[1024], cbNeeded, cProcesses;
        unsigned int i;
        if( !EnumProcesses(aProcess, sizeof( aProcess), &cbNeeded) )
                // 배열 수, 리턴되는 바이트 수
                // 배열에 id값들이 들어간다.
                return;
        // 얼마나 많은 프로세스들이 리턴되었나 계산
        cProcesses = cbNeeded / sizeof( DWORD);
```

```
        // process 이름 출력
        for( i = 0; i < cbNeeded; i++)
                PrintProcessNameAndID( aProcess[i] );
}
```

[예제 3-12] Process 열거2

```
#include <windows.h>
#include <TLHelp32.h>
#include <iostream>
using namespace std;
void main(){
        // 계산기 ProcessID얻기
        HWND hCalc = FindWindow(0, TEXT("계산기"));
        DWORD pid;
        GetWindowThreadProcessId(hCalc, &pid);

        HANDLE hSnap = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, pid);
        MODULEENTRY32       mme;
        BOOL b = Module32First( hSnap, &mme);
        while( b )
        {
                cout << (void*)mme.modBaseAddr << " : "
                        << mme.szModule << " PATH : " << mme.szExePath << endl;
                b = Module32Next( hSnap, &mme);
        }
        CloseHandle(hSnap);
}
```

[예제 3-13] Process 내 모듈 열거

4. Process 간 데이터 공유 및 전달

4. 1 메모리 맵 파일(Memory Mapped File)

윈도우즈는 물리적인 메모리가 부족할 경우 하드 디스크의 페이징 파일을 메모리 대신 사용한다. 마치 페이징 파일이 물리적인 메모리의 일부인 것처럼 프로세스의 주소 공간에 맵하여 사용하며 필요한 경우 RAM 으로 읽어오므로 응용 프로그램 입장에서 볼 때 페이징 파일은 속도가 좀 느릴 뿐 RAM과 전혀 다를 것이 없다.

운영체제가 하드 디스크의 페이징 파일을 RAM 대용으로 사용하는 것이 가능하다면 일반 파일도 RAM 대용으로 사용하여 주소 공간에 맵핑할 수 있을 것이다.

메모리 맵 파일은 이런 이론에 기반하여 하드 디스크에 존재하는 파일의 내용을 프로세스의 주소 공간에 연결하는 기법이다. 이를 이용하면 서로 다른 프로세스간 메모리를 공유할 수 있다.

4.1.1 파일 매핑

파일 매핑을 하려면 다음과 같은 순서를 밟는다.

(1) CreateFile API 로 파일을 연다.
이 때 세로운 파일을 만들 지 않고 페이징 파일을 사용할 수 도 있다.
페이징 파일을 사용하면 해당 과정이 생략된다.

(2) CreateFileMapping API로 그 파일에 대한 파일 매핑 객체를 생성한다.

(3) MapViewOfFile API로 파일 매핑 객체의 뷰를 생성한다.

CreateFileMapping API의 프로토타입은 아래와 같다.

HANDLE CreateFileMaping {	반환값 :객체 핸들
HANDLE hFile,	파일 핸들
LPSECURITY_ATTRIBUTES lpAttrs,	보안 속성
DWORD flProtect,	접근 허가
DWORD dwMaxSizeHigh,	매핑할 크기의 상위 32비트
DWORD dwMaxSizeLow,	매핑할 크기의 하위 32비트
LPCTSTR lpName);	객체명

첫 번째 인자에는 매핑하려는 파일의 핸들을 지정한다. 이는 CreateFile API 로 파일을 열었을 때의 반환값 핸들을 그대로 넘기면 된다.
INVALID_HANDLE_VALUE(- 1) 을 전달하면 페이징 파일을 사용하게 된다.

두 번째 인자는 보안 속성을 지정하는 것인데, 주로 NULL 을 사용한다.

세 번째는 매핑된 메모리 영역의 접근 권한을 보여준다. PAGE_READONLY, PAGE_READ WRITE 등의 상수 외 영역을 예약만하고 commit 하지 않을 것을 지시하는 SEC_RESERVE 상수를 논리 OR 로 조합해서 지정할 수 있다.

네 번째와 다섯 번째 인자에는 매핑하려는 범위의 최대 크기를 상위 32비트, 하위 32비트로 나누어 지정한다. 양쪽 모두 0을 지정했을 경우에는 파일의 현재 크기를 지정한 것과 같다. 만약 파일 크기보다 큰 크기를 지정하면 자동으로 파일 크기가 확장된다. 반대로 작은 크기로 지정했을 경우에는 앞에서부터 그 범위까지만 매핑 대상이 된다.

마지막 인자에는 객체의 이름을 지정한다. NULL을 지정하면 이름없는 객체가 된다. 따라서 프로세스 간 파일 공유를 할 수 없고, 대용량 메모리 사용시 사용하는 기법이다.

MapViewOfFile API의 프로토타입은 아래와 같다.

LPVOID MapViewOfFile(	반환값 : 뷰 메모리 영역의 포인터
HANDLE hFileMapObj,	파일 매핑 객체 핸들
DWORD dwDesiredAccess,	필요한 접근 권한
DWORD dwFileOfsHigh,	뷰를 실행할 범위의 시작 오프셋 상위 32비트
DWORD dwFileOfsLow,	뷰를 실행할 범위의 시작 오프셋 하위 32비트
SIZE_T dwBytesToMap);	뷰를 실행할 범위의 크기

이전 함수인 파일 매핑 객체는 정보만 관리할 뿐이며 실제 매핑을 해서 파일을 읽고 쓰려면 MapViewOfFile API 로 '뷰'를 생성해야 한다. 뷰는 실제로 파일의 내용이 매핑되는 가상 주소 영역이다.

첫 번째 인자에는 파일 매핑 객체의 핸들을 지정한다.

두 번째 인자에 접근 권한을 지정한다. 접근 권한은 파일 매핑 객체에서 허가된 것만 가능하다.

세 번째와 네 번째 인자에는 매핑하려는 파일 영역의 선두부터의 오프셋을 상위 32비트, 하위 32비트로 나누어 지정한다.

마지막 인자에 그 영역의 크기를 지정한다. 여기서 지정한 크기 만큼의 연속 영역이 가상 주소 공간에 예약되어 지정된 범위의 파일 데이터가 매핑된다.

MapViewOffile API는 가상 메모리 영역을 확보한 뒤 선두 번지의 포인터를 반환하며 이것으로 파일 매핑을 완료한다. 이 영역의 가상 메모리 데이터를 읽으면 대응되는 파일의 데이터를 읽을 수 있다. 반대로 데이터를 써넣으면 파일 데이터가 갱신된다.

단 파일 매핑을 실시한 시점에서 뷰의 데이터가 모두 물리 메모리에 로드되는 것은 아니다. Windows에서는 프로그램이 논리 페이지에 실제로 접근한 시점에서 처음으로 파일 데이터를 로드하도록 되어 있다. 따라서 접근하지 않는 부분의 데이터는 파일에서 읽을 필요가 없다.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <windows.h>
void main(){
	// 1. 화일 생성
	HANDLE hFile = CreateFile( TEXT("a.txt"), GENERIC_READ | GENERIC_WRITE,
	FILE_SHARE_READ | FILE_SHARE_WRITE,0,
	CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

	// 2. 화일 매핑 KO 생성
	HANDLE hMap = CreateFileMapping( hFile,  0, // 매핑할 화일, KO 보안
			PAGE_READWRITE, // 접근 권한
			0, 100,           // 매핑 객체 크기
			TEXT("map"));    // 매핑 객체 이름

	// 3. 매핑 객체를 사용해서 가상 주소와 파일 연결
	char* p = (char*)MapViewOfFileEx( hMap, FILE_MAP_WRITE,
			0, 0,  // file offset

		0,                  // 크기.(0 매핑 객체 크기)
		(void*)0x10000000); // 원하는 주소.
	if ( p == 0 )
		printf("error");
	else{
		printf("매핑된 주소 : %p₩n", p);
		strcpy( p, "hello");
		p[50] = 'a';
		p[100] = 'b';
	}
	UnmapViewOfFile( p );
	CloseHandle( hMap );
```

```
        CloseHandle( hFile );
}
```

[예제 4-1] 파일 맵핑

4.1.2 파일 매핑을 이용한 공유 메모리

파일 매핑 객체를 이해했다면 공유 메모리와 어떤 관계가 있는지를 살펴보자. 사실 이름 붙은 파일 매핑 객체를 사용하면 간단하게 공유 메모리를 만들 수 있다.

파일 매핑 객체의 핸들을 얻어 올 때는 이름을 지정해서 OpenFileMapping API를 호출하면 된다. 생성하는 프로세스가 정해지지 않은 경우에는 우선 CreateFileMaping API를 호출해 보는 방법도 있다. CreateFileMapping 은 지정한 이름의 객체가 이미 존재할 경우에는 그 핸들을 돌려 주기 때문이다.

OpenFileMapping API 프로토타입은 아래와 같다.

HANDLE OpenFileMapping(	반환값 : 객체 핸들
DWORD dwDesiredAccess,	필요한 접근 권한
BOOL bInherithandle,	자식 프로세스에 상속할 것인가?
LPCTSTR lpName);	객체명

파일 매핑 객체 핸들을 위의 함수를 통해 얻었다면 MapViewOfFile API로 뷰를 생성한다. 파일의 동일한 부분을 담은 뷰를 여러 프로세스에서 생성하면 이들은 동일한 데이터를 참조한다. 이는 프로세스 하나에서 여러 개의 뷰를 만들었을 때와 같다.

물리 메모리에 로드되어 있을 때는 각각의 뷰 페이지가 동일한 물리 페이지를 참조한다. 따라서 프로세스 하나가 데이터를 기록하면 동일한 데이터가 다른 프로세스의 뷰에도 나타난다. 이것이 바로 공유 메모리의 조건이다.

아래 소스는 2개의 프로젝트로 되어 있다. 첫 번째 프로그램은 공유 메모리를 생성하고 데이터를 기록하는 역할을 하고 두 번째 프로그램은 기록된 데이터를 읽는 역할을 한다. 첫 번째 프로그램에서 기록했다는 사실을 알리기 위해 EVENT 동기화 객체를 사용하였다. 해당 객체는 추 후 학습할 예정이다.

```
typedef struct _LINE{
        POINTS ptFrom;
```

```
        POINTS ptTo;
} LINE;
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
        static HANDLE hEvent, hMap;
        static LINE*  pData;
        static POINTS ptFrom;
        switch( msg )
        {
        case WM_LBUTTONDOWN: ptFrom = MAKEPOINTS(lParam);return 0;
        case WM_MOUSEMOVE:
                if ( wParam & MK_LBUTTON )
                {
                        POINTS pt = MAKEPOINTS(lParam);
                        HDC hdc = GetDC( hwnd );
                        MoveToEx( hdc, ptFrom.x, ptFrom.y, 0);
                        LineTo  ( hdc, pt.x,     pt.y );
                        ReleaseDC( hwnd, hdc );

                        // MMF 에 넣는다.
                        pData->ptFrom = ptFrom;
                        pData->ptTo   = pt;
                        SetEvent( hEvent );
                        ptFrom = pt;
                }
                return 0;
        case WM_CREATE:
                hEvent = CreateEvent( 0, 0, 0, TEXT("DRAW_SIGNAL"));
                hMap = CreateFileMapping( (HANDLE)-1, // Paging 화일을 사용해서 매핑
                        0, PAGE_READWRITE, 0, sizeof(LINE), TEXT("map"));
                pData = (LINE*)MapViewOfFile( hMap, FILE_MAP_WRITE, 0, 0,0);
                if ( pData == 0 )
```

```
                                MessageBox( 0, TEXT("Fail"), TEXT(""), MB_OK);
                        return 0;
                case WM_DESTROY:
                        UnmapViewOfFile( pData );
                        CloseHandle( hMap );
                        CloseHandle( hEvent );
                        PostQuitMessage(0);
                        return 0;
                }
                return DefWindowProc( hwnd, msg, wParam, lParam);
}
```

[예제 4-2] 프로세스간 데이터 공유1

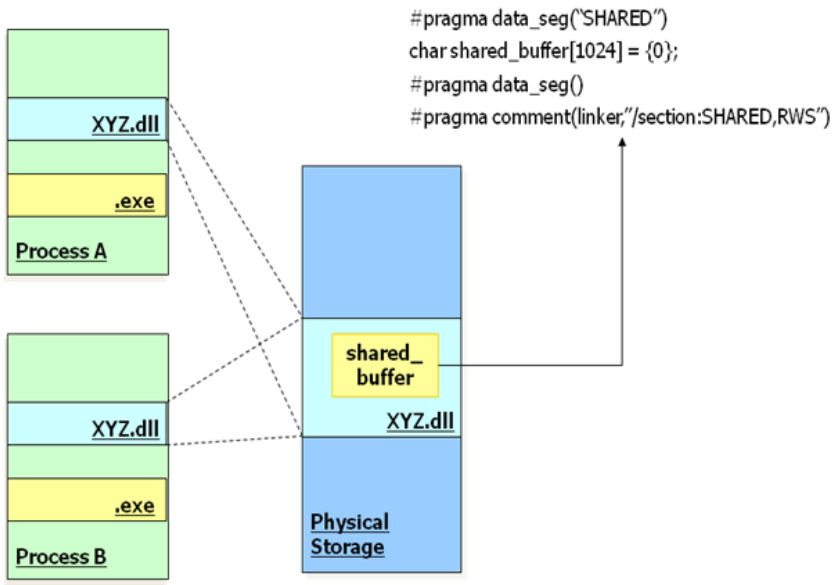
```
typedef struct _LINE
{
        POINTS ptFrom;
        POINTS ptTo;
} LINE;

DWORD WINAPI foo( void* p ){
        HWND hwnd = (HWND)p;
        HANDLE hEvent = OpenEvent( EVENT_ALL_ACCESS, FALSE, TEXT("DRAW_SIGNAL"));
        HANDLE hMap   = OpenFileMapping( FILE_MAP_ALL_ACCESS, FALSE, TEXT("map"));
        if ( hMap == 0 )
        {
        MessageBox( 0, TEXT("1번 프로그램을 먼저 실행하세요"), TEXT(""), MB_OK);
                        return 0;
        }
        LINE* pData = (LINE*)MapViewOfFile( hMap, FILE_MAP_READ, 0, 0, 0);
        while ( 1 )
        {
```

```
                    // Event를 대기한다.
                    WaitForSingleObject( hEvent, INFINITE );
                    // 이제 Line의 정보가 pData에 있다.
                    HDC hdc = GetDC( hwnd );
                    MoveToEx( hdc, pData->ptFrom.x, pData->ptFrom.y, 0);
                    LineTo  ( hdc, pData->ptTo.x,   pData->ptTo.y);
                    ReleaseDC( hwnd, hdc );
          }
          UnmapViewOfFile( pData );
          CloseHandle( hMap );
          CloseHandle( hEvent );
          return 0;
}
```

[예제 4-2] 프로세스간 데이터 공유2

4. 2 공유 섹션을 이용한 메모리 공유

프로세스간 메모리 공유 방법 중에 공유 섹션을 이용하여 메모리를 공유하는 방법이 있다.



PE 포맷(Windows실행 파일의 구조) 은 구조적으로 섹션이라는 영역이 존재한다. 보통 실행코드는 .text 섹션에, 데이터(초기화된 변수들)는 .data 섹션으로 지정되며 그 외 필요에 따라서 다른 섹션들도 존재할 수 있다.

만약, 자신이 작성한 실행 코드 혹은 데이터들을 별도의 섹션으로 만들어서 분리를 해야 할 경우가 생긴다면 어떻게 해야 할까? 결론은 섹션을 생성할 수 있는 기능을 지원해준다 .

그리고 해당 섹션 생성시 공유 여부를 설정할 수 있다.

아래 예제를 살펴보자.

```
#include <stdio.h>
#include <windows.h>
#include <conio.h>
// 초기화된 전역 변수 .data section 에 놓인다 - 기본적으로 COPY ON WRITE
int x = 0;
#pragma data_seg("AAA")       // exe(PE) 에 새로운 data section AAA를 만든다
int y = 0;
#pragma data_seg()
```

```
// AAA 섹션의 속성을 지정한다. (Read, Write, Share)
#pragma comment(linker, "/section:AAA,RWS")
void main()
{
        ++x; ++y;

        printf("x = %dWn", x);
        printf("y = %dWn", y);
        getch();
}
```

[첫 번째 실행시]

X = 1

y = 1

[두 번째 실행시]

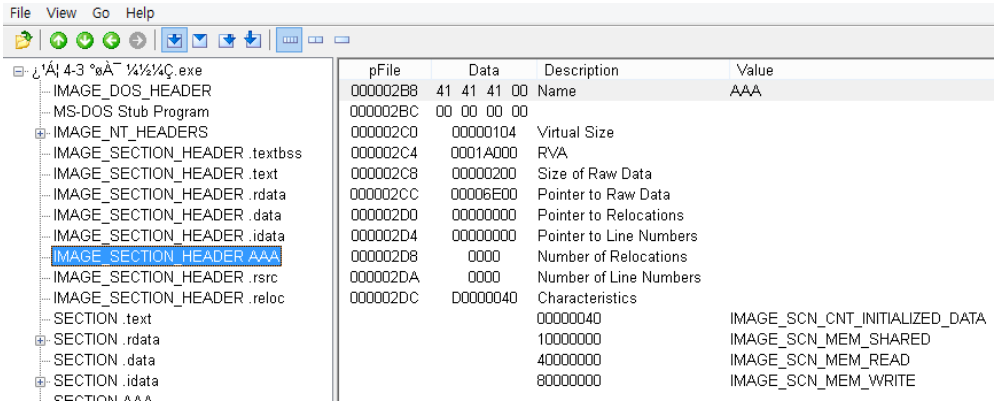
X = 1

y = 2

[예제 4-3] 공유 섹션

위 프로그램을 죽이지 말고 2번 실행하면 .AAA라는 섹션영역에 저장된 변수가 공유되는 결과를 확인할 수 있다.

주석을 읽어 보면 어떠한 프로그램인지 이해할 수 있을 것이다. 아래 그림은 PEView 유틸을 통해 생성된 .AAA 섹션이 생성된 결과이다. 우측 창에 보면 .AAA 섹션이 IMAGE_SCN_MEM_SHARED 플래그 값으로 설정된 것을 볼 수 있다.



위의 소스는 동일한 프로그램의 서로 다른 프로세스간 데이터 공유를 하는 예제인데, 만약 DLL을 사용하여 공유 섹션을 구성하면 서로 다른 프로그램의 프로세스 사이의 데이터 공유도 가능하다.

4. 3 WM_COPYDATA(IPC)

IPC는 Inter Process Communication 의 약어로써 서로 다른 프로세스간 데이터 통신을 하는 기법을 의미한다. IPC 관련 기법은 여러 가지가 있는데 여기서는 WM_COPYDATA에 대해 알아 보도록 하자.

Windows 메시지로 다른 프로세스에 포인터를 넘기는 것은 의미가 없다. 그러나 예외가 있다. 예를 들어, WM_GETTEXT 메시지는 WPARAM 타입, LPARAM 타입의 각 전달인자에 버퍼 크기와 주소를 넘겨서 SendMessage API로 송신하면 윈도우의 제목표시줄 문자열이 버퍼에 복사되어 돌아온다. 이 메시지는 송신지 윈도우가 다른 프로세스라도 작동한다.

Windows에서는 프로세스 간에 임의의 데이터를 주고 받을 수 있는 특별한 메시지를 제공해 주는데 그것이 바로 WM_COPYDATA 메시지이다.

WM_COPYDATA 메시지를 이용하려면 COPYDATASTRUCT 타입 구조체를 준비해서 송신하려는 데이터를 저장한 버퍼의 크기와 포인터를 cbData, lpData 맴버에 설정한다.
그리고 나서 wParam에 송신측 윈도우 핸들, lParam에 이 구조체의 포인터를 담아서 SendMessage API로 송신하면 Windows에서 자동으로 프로세스 간에 버퍼의 내용을 복사해 준다.

수신측 윈도우의 프로시저가 이 메시지를 처리할 경우에는 Windows에서 할당하여 데이터를 복사한 버퍼의 포인터가 lpData에 설정되어 있다. dwData에 데이터 구조를 식별하기 위한 ID를 넣어두면 다양한 데이터 형식에서 데이터를 주고 받을 수도 있다.

COPYDATASTRUCT 구조체 정의는 아래와 같다.

typdef struct tagCOPYDATASTRUCT{	
DWORD dwData,	임의의 데이터, 사용은 자유
DWORD cbData;	데이터 영역 크기

	(lpData로 전달되는 데이터 크기)
PVOID lpData;	송신할 데이터 영역 주소
}COPYDATASTRUCT, *PCOPYDATASTRUCT;	

WM_COPYDATA 메시지는 간편하게 이용할 수 있는 반면 주의가 필요하다.

첫 번째는 송신하려는 데이터 안에 포인터를 포함해서는 안 된다. Windows는 버퍼의 데이터 구조를 인식하지 못하기 때문에 필요에 따라서 주소를 변환하거나 그 포인터가 가리키는 메모리의 내용까지 복사해 주지는 않기 때문이다.

두 번째로 주의할 점은 버퍼의 프로세스 간 복사는 송신측에서 수신측에게로만 이루어진다. 수신측에서 버퍼의 내용을 변경해도 그 내용은 메시지의 송신측에는 복사되지 않는다. 반대 방향으로 데이터를 넘기고 싶다면 수신측에서 별도로 WM_COPYDATA 메시지를 보내야 한다.

세 번째로 복사된 버퍼는 수신측 윈도우 프로시저의 호출에서 복귀하는 시점에 자동으로 해제된다. 따라서 lpData에 설정된 포인터값을 보존해 두어도 의미가 없다. 다음에 버퍼의 내용이 필요하다면 데이터 그 자체를 다른 장소에 보존해 두지 않으면 안 된다.

이러한 제한을 이해했다면 프로세스 간 동기화에 신경쓸 필요가 없기 때문에 프로그램에서 간단히 사용할 수 있다.

아래 예제는 문자열을 전송/수신하는 간단한 예이다.
2번째 프로그램을 먼저 실행시켜서 테스트 하면 된다.

```
#include <windows.h>
#include <stdio.h>
void main()
{
	char buf[256] = {0};
	HWND hwnd = FindWindow( 0, TEXT("B"));
	if ( hwnd == 0 )
	{
		printf("B 윈도우를 먼저 실행해 주세요₩n");	return;
	}
```

```
        while ( 1 )
        {
                printf("B에게 전달한 메세지를 입력하세요 >> ");
                gets( buf); // 1줄입력 ?
                // 원격지로 Pointer를 전달하기 위한 메세지 - WM_COPYDATA
                COPYDATASTRUCT cds;
                cds.cbData = strlen(buf)+1;  // 전달한 data 크기
                cds.dwData = 1;                                    // 식별자
                cds.lpData = buf;                        // 전달할 Pointer
                SendMessage( hwnd, WM_COPYDATA, 0,  (LPARAM)&cds);
        }
}
```

[예제 4-4] WM_COPYDATA 문자열 전송1

```
#include <windows.h>
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
        static HWND hEdit;
        switch( msg )
        {
        case WM_CREATE:
                hEdit = CreateWindow( TEXT("edit"), TEXT(""),
                WS_CHILD | WS_VISIBLE | WS_BORDER | ES_MULTILINE,
                10,10,400,400, hwnd, (HMENU)1, 0, 0);
                return 0;
        case WM_COPYDATA:
                {
                        COPYDATASTRUCT* p = (COPYDATASTRUCT*)lParam;
                        if ( p->dwData == 1 ) // 식별자 조사.
                        {
                        // Edit Box에 추가 한다.
                        SendMessage( hEdit, EM_REPLACESEL, 0, (LPARAM)(p->lpData));
```

```
                                SendMessage( hEdit, EM_REPLACESEL, 0, (LPARAM)"WrWn");
                                }
                        }
                        return 0;
                case WM_DESTROY:
                        PostQuitMessage(0);
                        return 0;
                }
                return DefWindowProc( hwnd, msg, wParam, lParam);
}
```

[예제 4-4] WM_COPYDATA 문자열 전송2

아래 예제는 사용자 정의 구조체를 전송하는 예이다.
두 번째 프로그램을 먼저 실행시킨다.

```
typedef struct tagDATA
{
        TCHAR str1[20];
        TCHAR str2[20];
        int   num;
}DATA;

BOOL CALLBACK DlgProc( HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
        switch (msg)        {

        case WM_COMMAND:
                switch( LOWORD( wParam ) )
                {
                case IDOK:
                {
                        HWND hwnd = FindWindow( 0, TEXT("B"));
                        DATA data;
```

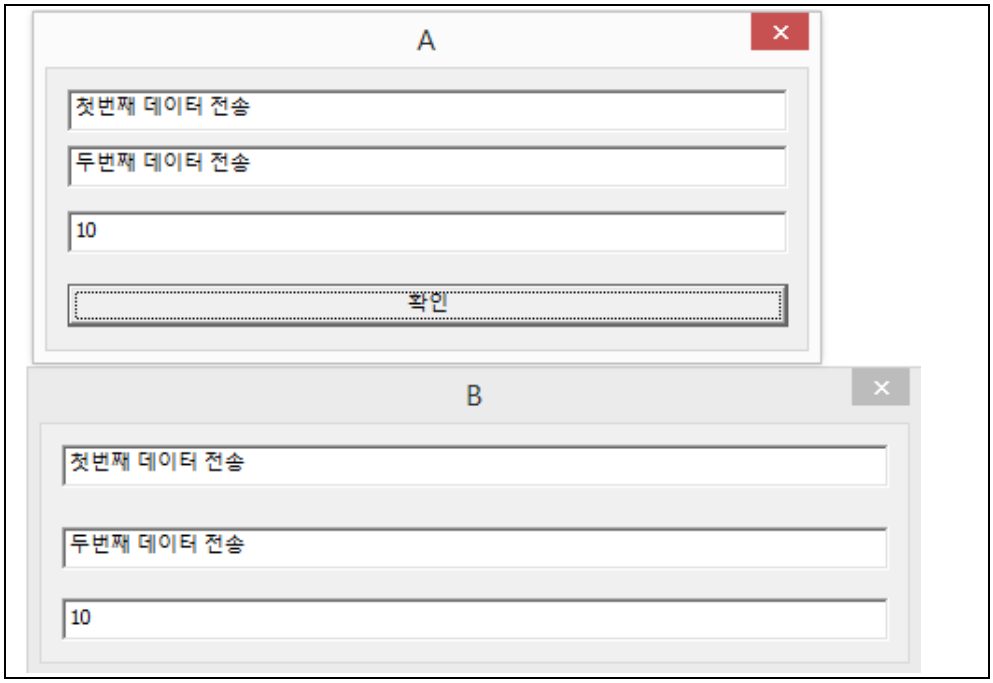
```
                    GetDlgItemText(hDlg, IDC_EDIT1, data.str1, sizeof(data.str1));
                    GetDlgItemText(hDlg, IDC_EDIT2, data.str2, sizeof(data.str2));
                              data.num = GetDlgItemInt(hDlg, IDC_EDIT3, FALSE, FALSE);
                              // 원격지로 Pointer를 전달하기 위한 메세지 - WM_COPYDATA
                              COPYDATASTRUCT cds;
                              cds.cbData = sizeof(DATA);  // 전달한 data 크기
                              cds.dwData = 1;
          // 식별자
                              cds.lpData = &data;                    // 전달할 Pointer
                              SendMessage( hwnd, WM_COPYDATA, 0,  (LPARAM)&cds);
                    }
                    return 0;
                    case IDCANCEL: EndDialog(hDlg, IDCANCEL);         return 0;
                    }
                    return TRUE;
          case WM_DESTROY:
                    return 0;
          }
          return FALSE;
}
```

[예제 4-5] WM_COPYDATA 사용자 정의 구조체 사용1

```
typedef struct tagDATA
{
          TCHAR str1[20];
          TCHAR str2[20];
          int   num;
}DATA;

BOOL CALLBACK DlgProc( HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
```

```
{
        switch (msg)        {
        case WM_COPYDATA:
                {
                COPYDATASTRUCT *ps = (COPYDATASTRUCT*)lParam;
                DATA *pData = (DATA*)ps->lpData;
                SetDlgItemText(hDlg, IDC_EDIT1, pData->str1);
                SetDlgItemText(hDlg, IDC_EDIT2, pData->str2);
                SetDlgItemInt(hDlg, IDC_EDIT3, pData->num, FALSE);
                }
                return 0;
        case WM_COMMAND:
                switch( LOWORD( wParam ) )
                {
                case IDCANCEL: EndDialog(hDlg, IDCANCEL);        return 0;
                }
                return TRUE;
        case WM_DESTROY:
                return 0;
        }
        return FALSE;
}
```

[실행 화면]



[예제 4-5] WM_COPYDATA 사용자 정의 구조체 사용2

4. 4 파이프(IPC)

파이프는 가상적인 데이터 통로다. 수도관과 같은 관을 상상하면 좋을 것이다. 한 프로세스와 다른 프로세스를 파이프로 연결해서 그 안에 데이터를 흘러보내는 형식으로 정보를 교환한다. 진짜 수도관과 달리 파이프는 중간에 분기하지 않는다. 데이터의 출입구는 각각의 프로세스에 하나뿐이다. 한쪽 입구에서 흘러보낸 데이터는 반드시 다른 쪽 입구에서 꺼낼 수 있다.

파이프의 역할은 데이터의 통신로를 제공할 뿐이므로 어떻게 데이털르 흘려보낼지 제어하는 것은 프로그램의 역할이다. 한쪽에서 보낸 데이터에 따라 상대가 어떤 데이터를 반환할 것인가 하는 것은 쌍방간에 미리 합의해 두어야 한다. 이런 규약을 프로토콜 이라 한다.

Windows 파이프는 크게 '이름있는 파이프(named pipe)' 와 '이름이 없는 파이프(unnamed pipe)'로 나뉜다. 이름의 유무에 따라 기능면에서 큰 차이가 있다. 이름있는 파이프는 본격적인 프로세스 간 통신 전용으로 기능이 풍부한 대신 이용하기가 약간 어렵다. 한편 이름없는 파이프는 기능이 제한된 대신에 구조가 단순하고 다루기가 쉽다. 이용하는 API

나 호출 순서가 다르므로 따로 설명하기로 하겠다.

4.4.1 이름 없는 파이프

이름 없는 파이프의 경우는 CreatePipe API를 호출하면 단방향 파이프가 하나 생성된다. 핸들이 두 개 반환되는 이유는 파이프가 두 개 있기 때문이 아니라 파이프의 입구와 출구에 대해 각각 핸들을 할당할 수 있기 때문이다. 어느 쪽 방향으로 데이터를 보낼지에 따라 두 핸들의 한쪽을 상대측 프로세스에 넘겨서 통신한다. 쌍방향으로 통신하고 싶을 경우에는 파이프를 두 개 만들어야 한다.

다음은 CreatePipe API 함수이다.

BOOL CreatePipe(	리턴값 : BOOL
PHANDLE hReadPipe,	읽기 핸들
PHANDLE hWritePipe,	쓰기 핸들
LPSECURITY_ATTRIBUTES lpPipeAttributes,	보안 속성
DWORD nSize);	파이프 버퍼 크기

이 함수는 파이프를 생성하고 파이프의 양쪽 끝인 읽기 핸들과 쓰기 핸들을 첫 번째 인수와 두 번째 인수로 전달된 참조인수로 전달된다. 두 개의 HANDLE 형 변수를 선언하고 변수의 포인터를 이 함수로 전달해 주면 된다.

세 번째 인수는 파이프의 보안 속성을 지정하는데 이름없는 파이프는 주로 상속을 통해 자식 프로세스로 전달되므로 보안 속성의 bInHeritHandle 맴버를 TRUE로 설정해 주어야 한다.

네 번째 인수 nSize는 파이프의 버퍼 크기이되 0으로 지정하면 디폴트 크기로 버퍼를 만들어 준다.

이 함수로 파이프를 생성한 후 파이프의 한쪽 끝을 통신하고자 하는 프로세스에게 전달해 주어야 한다. 이 때 IPC 나 부모가 자식에게 핸들을 상속시켜 주는 방식을 사용하게 된다.

만약 파이프를 만든 프로세스가 다른 프로세스에게 데이터를 보내고자 할 때는 읽기 핸들을 전달해 주고 자신은 쓰기 핸들에 데이터를 기록하면 된다. 반대의 경우, 즉 파이프를 만든 프로세스가 데이터를 받아야 할 때는 쓰기 핸들을 전달해 주고 자신은 읽기 핸들로 데이터를 읽는다.

다음 예제를 살펴보자. 총 2개의 예제로 되어 있다.

```
#include <stdio.h>
#include <windows.h>
void main(){
          HANDLE hRead, hWrite;
          CreatePipe( &hRead, &hWrite, 0, 4096);
          // 읽기 위한 핸들을 상속 가능하게 한다.
          SetHandleInformation( hRead, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
          TCHAR cmd[256];
          wsprintf( cmd, TEXT("child.exe %d"), hRead); // 명령형 전달인자 사용
          PROCESS_INFORMATION pi;
          STARTUPINFO si = { sizeof(si)};
          BOOL b = CreateProcess( 0, cmd, 0, 0, TRUE, CREATE_NEW_CONSOLE,
                    0,0,&si, &pi);
          if ( b )  {
                    CloseHandle( pi.hProcess);
                    CloseHandle( pi.hThread);
                    CloseHandle( hRead );
          }
          //-----------------------------------------
          char buf[4096];
          while ( 1 )         {
                    printf("전달할 메세지를 입력하세요 >> ");
                    gets( buf );
                    DWORD len;
                    WriteFile( hWrite, buf, strlen(buf)+1, &len, 0);
          }
}
```

[예제 4-6] 이름 없는 파이프1

```
#include <stdio.h>
#include <windows.h>
// 이 실행파일의 이름을 child.exe 변경하세요..
void main(int argc, char** argv){
        if ( argc != 2 )    {
        printf("이프로그램은 직접 실행하면 안됩니다. 부모를 실행해 주세요₩n");
                return;
        }
        // 부모가 보내준 pipe 핸들을 꺼낸다.
        HANDLE hPipe = (HANDLE)atoi(argv[1]);
        char buf[4096];
        while ( 1 )         {
                memset( buf, 0, 4096 );
                DWORD len;
                BOOL b = ReadFile( hPipe, buf, 4096, &len, 0);
                if ( b == FALSE ) break;
                printf("%s₩n", buf );
        }
        CloseHandle( hPipe );
}
```

[예제 4-6] 이름 없는 파이프2

위 예제를 이해 했다면 또 다른 파이프를 만들어서 서로 간 통신이 가능하도록 구현해 보는 것도 좋을 것이다.

4.4.2 이름 있는 파이프

이름이 있는 파이프는 클라이언트/서버 형태의 통신을 위해 사용한다. 파이프를 생성하는 프로세스가 서버가 되며, 클라이언트 프로세스는 그 파이프에 접속하는 방법으로 서버와 통신한다. 데이터 베이스 관리 시스템이 같은 시스템의 클라이언트 프로그램과 통신할 경우에 자주 이용하는 방법이다.

이름있는 파이프를 만들려면 CreateNamePipe API를 사용한다.

다음은 CreateNamedPipe API함수이다.

HANDLE CreateNamedPipe	리턴값 : 핸들
DWORD dwOpenMode,	읽기, 쓰기 모드
DWORD dwPipeMode,	전송타입, 데이터 수신타입, 블로킹 모드
DWORD nMaxInstance,	파이프 최대 개수
DWORD nOutBufferSize,	출력 버퍼 사이즈
DWORD nInBufferSize,	입력 버퍼 사이즈
DWORD nDefaultTimeOut,	
LPSECURITY_ATTRIBUTES lpPipeAttributes,	보안 속성

첫 번째 인자에 파이프명을 '₩₩.₩pipe₩파이프명' 형식으로 지정한다. 파이프를 생성할 때는 반드시 로컬 머신에서 생성하므로 서버명 부분에 로컬 머신을 나타내는 '.'을 사용한다.

두 번째 오픈모드는 파이프 내부의 데이터 방향을 지정한다.

PIPE_ACCESS_OUTBOUND	서버에서 출력하는 방향
PIPE_ACCESS_INBOUND	입력받는 방향
PIPE_ACCESS_DUPLEX	단방향 혹은 쌍방향

그러나 쌍뱡향이라고 해도 파이프 하나에서 흐르는 방향을 바꾸어 사용하는 것은 아니다. 쌍방향으로 지정 했을 경우에는 내부적으로 상향/하향 파이프가 두 세트 생성된다.

세 번째 파이프모드에는 파이프 내부에 흐르는 데이터를 바이트 단위로 따로 다룰 것인가(PIPE_TYPE_BYTE), 메시지 라는 묶음 데이터로 다룰 것인가(PIPE_TYPE_MESSAGE)를 지정한다. 후자의 방법은 기록한 데이터를 메시지로 만들고 읽을 때도 한 번의 조작으로 단위 범위로 읽을 수 있다. 데이터의 단락을 프로그램에서 판별할 필요가 없기 때문에 데이터 형식이 정해져 있는 경우에 편리하다.

네 번째 인자는 같은 이름의 파이프를 몇 개 생성할 수 있는지 지정한다. 클라이언트/서버형 통신에서는 여러 클라이언트로부터 동시에 접속 요구를 받아들이는 일이 있기 때문에 그것들을 병행해서 처리할 수 있도록 같은 이름의 파이프를 여러 개 만들 수 있게 되어 있다. 대개 이런 경우에 각각의 클라이언트 처리는 다른 스레드에서 한다. 그러나 스레드의 수가 많아지면 리소스가 부족하게 되고, 성능이 저하되므로 최대값을 설정하도록 되어 있다.

다섯번째와 여섯번째 인자는 각각 입출력에 사용하는 파이프의 버퍼 크기이다. 버퍼 크기는 수도관의 용량에 해당하는 것이라고 생각하면 좋을 것이다. 반대쪽 입구가 막혀 있

어도 그 용량까지는 관에 물을 넣을 수 있다. 마찬가지로 파이프도 반대쪽에서 데이터를 읽지 않아도 버퍼 크기까지는 데이터를 기록할 수 있다. 그러나 버퍼가 가득 찬 상태에서 계속 기록하려고 하면 빈 공간이 생길 때까지 대기하므로 주의해야 한다.

서버에 빈 파이프가 없는 경우 클라이언트는 나중에 설명할 WaitNamedPipe API로 인해 빈 곳이 생길 때까지 대기할 수 있다.

일곱번째 인자에는 대기 시 기본 타임아웃 시간을 밀리초 단위로 지정한다.

마지막 인자는 보안 지정이다. 이름있는 파이프는 네트워크의 다른 컴퓨터에서도 접속할 수 있기 때문에 부정한 접근을 허락하지 않도록 적절히 설정해야 한다.

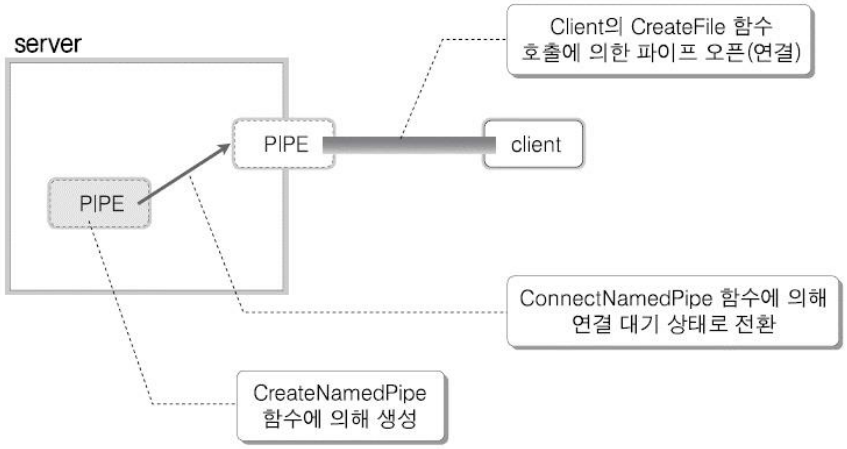
파이프를 생성했다면 서버측에서 파이프를 대기 상태로 두어야 한다. 그러기 위해서는 CreateNamedPipe API에서 반환된 파이프 핸들을 ConnectNamedPipe API로 호출하면 된다. 두 번째 전달이자는 오버랩된 IO를 사용하기 위한 것이지만, 처리가 복잡하므로 여기서는 생략한다. 사용하지 않을 경우에 는NULL을 지정한다.

BOOL ConnectNamedPipe	반환값 : 접속에 성공했는가?
HANDLE hNamedPipe,	이름있는 파이프 핸들
LPOVERLAPPED lpOverlapped	오버랩된 IO를 지정하기 위한 구조체

오버랩된 IO가 아닌 경우에 ConnectNamedPipe API는 클라이언트에서 오는 접속을 기다리는 대기 상태에 들어가며, 접속될 때까지 복귀하지 않는다. 즉 에러 없이 복귀하면 클라이언트와 접속 상태에 있다는 것이다. 파이프 핸들은 파일 핸들과 똑같이 취급할 수 있으므로 ReadFile/WriteFile API로 파이프를 읽고 쓸 수 있다.

통신이 끝나면 DisconnectNamedPipe API를 호출해서 접속을 끊는다. 이 때 파이프 버퍼에 남아 있는(클라이언트가 아직 읽지 않은)데이터는 모두 파기된다. 이를 피하기 위해서는 접속 해제 전에 FlushFileBuffers API를 호출하면 된다. 그리고 파일 핸들처럼 CloseHandle API로 핸들을 닫는다.

아래 그림은 위에서 설명한 내용이다.



클라이언트에서 파이프에 접속한다.

클라이언트에서는 일단 서버의 파이프에 접속할 필요가 있다. 이때 사용하는 것은 파일을 열 때 사용하는 CreateFile API이다.

'₩₩'서버명₩pipe명' 을 지정하면 서버의 파이프에 접속할 수 있다. 접근모드는 파이프모드에 맞추어 지정하면 된다.

파이프 오픈 모드	CreateFile 접근 모드
PIPE_ACCESS_INBOUND	GENERIC_WRITE
PIPE_ACCESS_OUTBOUND	GENERIC_READ
PIPE_ACCESS_DUPLEX	GENERIC_READ | GENERIC_WRITE

[클라이언트측에서 접속할 때 사용하는 접근 모드]

서버의 파이프가 모두 사용중이라서 접속할 수 없으면 CreateFile은 INVALID_HANDLE_VALUE 에러를 반환한다. 이는 파이프 그 자체가 존재하지 않는 경우와 구별할 수 없기 때문에 GetLastError API로 에러 코드를 조사한다. ERROR_PIPE_BUSY 라면 모두 사용중이다.

이런 경우에는 WaitNamedPipe API로 빈 곳이 생길 때까지 대기할 수 있다. 인자에는 파이프명과 최대 대기 시간을 지정한다. 대기 시간에는 무한(NMPWAIT_WAIT_FOREVER) 외에 파이프 생성시 설정한 기본값을 사용하도록 지정하는 NMPWAIT_USE_DEFAULT_WAIT 도 사용할 수 있다. 작업 하나를 처리하는 데 걸리는 시간이 대부분 정해져 있다면 서버

에서 기본값을 적절히 설정해서 클라이언트의 대기 시간을 최적화 할 수 있다.

주의할 것은 파이프에 빈 곳이 생겼을 경우에도 이 API는 단순히 복귀한다는 것이다. 접속하려면 다시 CreateFile API 를 호출해야 한다. 그래서 CreateFile API를 호출했을 때는 간발의 차이로 파이프를 다른 프로세스에게 뺏길 가능성도 있다.

접속이 완료되면 서버와 마찬가지로 CreateFile API가 반환하는 파이프 핸들에 대해 ReadFile/WriteFile API를 사용해서 데이터를 읽고 쓸 수 있다. 통신이 종료되면 CloseHandle API를 호출해서 파이프를 닫는다.

```
#include <iostream>
using namespace std;
#include <windows.h>
void main()
{
	// named pipe 만들기
	HANDLE hPipe = CreateNamedPipe( TEXT("₩₩₩₩.₩₩pipe₩₩TimeServer"), // pipe이름
		PIPE_ACCESS_OUTBOUND, // 출력 전용.

	PIPE_TYPE_BYTE,
	1, // 동일 이름의 파이프를 만들수 있는 최대 갯수.
	1024,1024, // 입출력 버퍼 크기
	1000, // WaitNamedPipe함수로 대기할수 있는 시간
	0); // K0 보안
	if ( hPipe == INVALID_HANDLE_VALUE )
	{
		cout << "Pipe를 만들수가 없습니다." << endl;
		return ;
	}
	while ( 1 )
	{
		// 클라이 언트의 접속을 대기한다.
		BOOL b = ConnectNamedPipe( hPipe, 0);
```

```
                    if ( b == FALSE && GetLastError() == ERROR_PIPE_CONNECTED )
                              b = TRUE;
                    if ( b == TRUE )
                    {
                              // pipe를 통해서 현재 시간을 알려준다.
                              SYSTEMTIME st;
                              GetSystemTime( &st );
                              DWORD len;
                              WriteFile( hPipe, &st, sizeof(st), &len, 0);
                              FlushFileBuffers( hPipe );
                              // 접속을 끊는다.
                              DisconnectNamedPipe( hPipe );
                              cout << "Work... Done" << endl;
                    }
          }
}
```

[예제 4-7] 이름 있는 파이프1

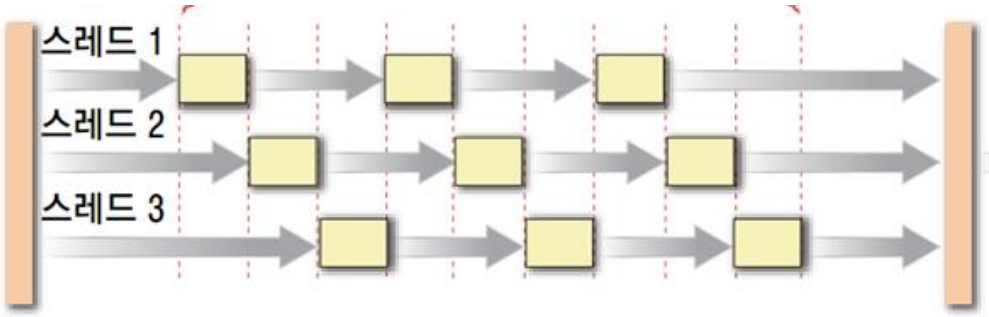
```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void main()
{
          HANDLE hPipe = CreateFile( TEXT("WWWW.WWpipeWWTimeServer"), // UNC
                    GENERIC_READ,
                    FILE_SHARE_WRITE,
                    0,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    0);
          if ( hPipe == INVALID_HANDLE_VALUE )
```

```
        {
                _tprintf(TEXT("Pipe 서버에 연결할수 없습니다₩n"));
                        return;
        }
        DWORD len;
        SYSTEMTIME st;
        ReadFile( hPipe, &st, sizeof(st), &len, 0);
        // UTC 시간을 ->지역 시간으로
        ::SystemTimeToTzSpecificLocalTime( 0, &st, &st );
        SetLocalTime( &st );
        printf("%d시 %d분₩n", st.wHour, st.wMinute);
        TCHAR date[256];
        TCHAR time[256];
        GetDateFormat( LOCALE_USER_DEFAULT, 0, &st,  0, date, 256);
        GetTimeFormat( LOCALE_USER_DEFAULT, 0, &st,  0, time, 256);
        _tprintf(TEXT("%s₩n"), date);
        _tprintf(TEXT("%s₩n"), time);
        CloseHandle( hPipe );
}
```

[예제 4-7] 이름 있는 파이프2

5. Thread

한 프로그램에서 여러 가지 작업을 동시에 수행해야 할 경우가 있다. 보통 CPU의 갯수에 따라 동시에 작업을 수행하는 것이 불가능하지만 동시에 수행되는 것처럼 보이게 만들 수 있는 여러 가지 방법들이 있다.



아래 예제를 살펴 보자.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
          static int Blue = 0;
          switch( msg )
          {
          case WM_CREATE:
                    SetTimer(hwnd, 1, 10, NULL);
                    break;
          case WM_TIMER:
                    Blue +=5;
                    InvalidateRect(hwnd, NULL, FALSE);
                    break;
          case WM_PAINT:
                    {
                              PAINTSTRUCT ps;
                              HDC hdc = BeginPaint(hwnd, &ps);
                              HBRUSH hbrush = CreateSolidBrush(RGB(0,0,Blue));
```

```
                    HBRUSH hOld = (HBRUSH)SelectObject(hdc, hbrush);
                    Rectangle(hdc, 100,100,400,400);
                    DeleteObject(SelectObject(hdc, hOld));
                    EndPaint(hwnd, &ps);
                }
                return 0;
        case WM_LBUTTONDOWN:
                {
                    HDC hdc = GetDC(hwnd);
                    POINTS pt = MAKEPOINTS(lParam);
                    Rectangle(hdc, pt.x, pt.y, pt.x+50, pt.y+50);
                    ReleaseDC(hwnd, hdc);
                }
                return 0;
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 5-1] Timer를 이용한 동시 작업

Timer를 이용하면 동시에 여러 작업을 수행할 수 있다. WM_SETTIMER는 0.1 초 간격으로 호출되면서 도형의 색상을 변경하고 있고, LButton을 클릭할 경우 해당 위치에 사각형을 출력한다.

동시 작업이 필요한 프로그램들은 고전적으로 이런 방법을 많이 사용했었다. 하지만 이 방법도 문제점을 가지고 있다. 우선 WM_TIMER 메시지는 최대 1초에 18.2 회밖에 발생하지 않으므로 좀 더 고속 처리가 필요할 때는 사용할 수 없다. 또한 타이머 메시지에서 화면을 그리는 동안은 다른 메시지를 곧바로 처리할 수 없으므로 반응성이 좋지 않다.

5. 1 CreateThread

멀티 스레드는 동시에 여러 작업을 수행할 수 있는 방법이다.

먼저 스레드는 프로세스 내에 존재하는 실제 경로, 즉 일련의 실행코드이다. 프로세스는 단지 존재하기만 한는 껍데기일 뿐이며 실제 작업은 스레드가 담당한다.

프로세스의 최초 스레드, 즉 메인스레드는 프로세스 생성 시에 자동으로 생성된다. 따라서 그대로라면 그 프로세스는 싱글스레드 프로세스가 된다. 프로세스를 멀티스레드로 하려면 메인스레드 안에서 새로운 스레드를 생성하면 된다. 이런 식으로 생성된 새로운 스레드 안에서 또 다른 스레드를 생성할 수도 있다.

스레드 생성에는 CreateThread API를 사용한다. 생성에 성공하면 이 API는 반환값으로 새로운 스레드 핸들을 돌려준다.

HANDLE CreateThread(	반환값 : 스레드 핸들
LPSECURITY_ATTRIBYTES lpThreadAttributes,	스레드 속성
SIZE_T dwStackSize,	스택 초기 크기
LPTHREAD_START_ROUNTINE lpStartAddress,	스레드 프로시저
LPVOID lpParameter,	스레드 전달인자
DWORD dwCreationFlags,	스레드 생성 플래그
LPDWORD lpThreadId	스레드 ID를 받을 변수의 주소
);	

첫 번째 인자는 스레드의 권한과 같은 보안 정보를 설정하는 부분이다. 특별한 이유가 없는 한 NULL을 넘겨도 상관없다.

두 번째 인자에는 생성하려는 스레드의 초기 스택 크기를 지정한다. 프로세스 내의 각 스레드에서는 주소 공간 자체는 공유하지만, 스택 영역은 각기 다른것을 사용하기 때문이다. 이 인자 역시 0을 넘기면 기본 크기를 할당해 준다.

세 번째 인자는 스레드 프로시저를 지정한다. 반드시 다음과 같은 프로토타입을 사용하는 함수의 주소를 지정해야 한다.

DWORD WINAPI threadFunc(LPVOID lpParameter);

이 함수는 스레드에서 실행하는 처리를 담당하므로 스레드 프로시저라고 한다. 스레드가

생성되면 프로시저의 앞부분부터 실행이 시작된다. 그리고 보통의 함수처럼 프로시저의 마지막에 도달하거나 도중의 return 문 혹은 ExitThread API 의 호출이 있을 때까지 코드에 작성된 순서대로 실행된다. 그리고 실행이 종료되는 동시에 스레드는 소멸한다.

스레드 프로시저는 반드시 스레드마다 다른 것을 준비하지 않아도 된다. 비슷한 처리를 하는 스레드를 많이 생성할 경우에는 동일한 스레드 프로시저를 사용해서 CreateThread API 를 여러 번 호출하면 된다.

스레드 프로시저의 인자인 lpParameter에는 CreateThread 의 네번째 인자에서 지정한 값을 그대로 전달 받는다. 이 인자를 이용해서 특정 데이터를 넘겨주면 스레드마다 다르게 처리할 수 있다.

주고 받는 데이터가 많아서 하나로 넘길 수 없는 경우도 있다. 그럴 때는 구조체에 데이터를 저장해서 그 포인터를 넘기면 된다. 프로세스 내의 스레드는 메모리 공간을 공유하기 때문에 포인터로 서로의 데이터를 참조할 수 있다.

그러나 포인터가 가리키는 데이터의 유효성을 판단할 때는 주의해야 한다. CreateThread API는 스레드 생성에 성공하면 스레드 프로시저의 종료를 기다리지 않고 즉시 호출로부터 복귀한다. 즉 호출한 곳은 스레드 프로시저의 처리 상태에 관계없이 계속해서 프로그램을 실행한다. 만약 호출한 곳의 함수가 종료되어 변수가 유효 영역을 벗어나거나 동적으로 할당된 메모리를 해제하였다면 다른 스레드에서 참조하려고 해도 이미 데이터는 무효가 되어 있을 것이다. 이런 일을 막기 위해서는 전역변수, 정적변수를 사용하거나 동적으로 할당한 메모리 포인터를 넘겨서 호출된 스레드에서 해제하는 방법을 사용할 수 있다.

CreateThread()의 5번째 인자로 CREATE_SUSPENDED를 지정하면 스레드가 실행을 중지한 상태로 만들어 진다.(Suspend Count가 1 인 상태)

아래코드를 살펴보자. 마우스 왼쪽버튼을 클릭하면 무한 루프를 돌면서 해당 함수가 종료하지 못하기 때문에 프로그램은 더 이상의 이벤트를 처리하지 못한다.

```
void fun1( LPVOID temp)
{
        HDC       hdc;
        BYTE Blue=0;
        HBRUSH hBrush, hOldBrush;
```

```
		HWND h = (HWND)temp;
		hdc = GetDC(h);
		while(1)
		{
			Blue++;
			Sleep(1);
			hBrush		= CreateSolidBrush(RGB(0, 0, Blue));
			hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
			Rectangle(hdc, 100, 100, 300, 400);

			SelectObject(hdc, hOldBrush);
			DeleteObject(hBrush);
		}
		ReleaseDC(h, hdc);
}
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
		static HANDLE hThread;
		static DWORD  ThreadID;
		switch( msg )
		{
		case WM_LBUTTONDOWN:
			{
				fun1(hwnd);
			}
			return 0;
		case WM_RBUTTONDOWN:
			{
				HDC hdc = GetDC(hwnd);
			Ellipse(hdc, LOWORD(lParam), HIWORD(lParam), LOWORD(lParam)+20,
				HIWORD(lParam)+20);
```

```
                    ReleaseDC(hwnd, hdc);
                }
                return 0;
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 5-2] Thread를 사용하지 않는 예

위의 코드를 스레드를 생성하는 코드로 변경한 예이다. 실행해 보면 이전 소스와는 달리 마우스 왼쪽 버튼을 클릭해도 다른 이벤트를 처리할 수 있음을 볼 수 있다.

```
DWORD WINAPI ThreadFunc1( LPVOID temp)
{
        HDC     hdc;
        BYTE Blue=0;
        HBRUSH hBrush, hOldBrush;

        HWND h = (HWND)temp;

        hdc = GetDC(h);
        while(1)
        {
                Blue++;
                Sleep(1);
                hBrush          = CreateSolidBrush(RGB(0, 0, Blue));
                hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
                Rectangle(hdc, 100, 100, 300, 400);

                SelectObject(hdc, hOldBrush);
```

```
                    DeleteObject(hBrush);
          }
          ReleaseDC(h, hdc);
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
          static HANDLE hThread;
          static DWORD  ThreadID;
          switch( msg )
          {
          case WM_LBUTTONDOWN:
                    {

          hThread = CreateThread(NULL, 0, ThreadFunc1, hwnd, 0, &ThreadID); // 2
                              CloseHandle(hThread);                    // use count : 1
                    }
                    return 0;
          case WM_RBUTTONDOWN:
                    {
                              HDC hdc = GetDC(hwnd);
                    Ellipse(hdc, LOWORD(lParam), HIWORD(lParam), LOWORD(lParam)+20,
                                                            HIWORD(lParam)+20);
                              ReleaseDC(hwnd, hdc);
                    }
                    return 0;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    return 0;
          }
          return DefWindowProc(hwnd, msg, wParam, lParam);
```

```
}
```

[예제 5-3] Thread를 사용한 예

5. 2 스레드 제어

아래의 함수를 사용하면 스레드를 잠시 동작을 중지시킬 수도 있고 다시 실행시킬 수도 있다.

DWORD SuspendThread(HWND hThread);
DWORD ResumeThread(HWND hThread);

SuspendThread는 스레드의 동작을 중지시키고 ResumeThread는 중지된 스레드를 다시 동작하도록 재개한다.

스레드는 내부적으로 중지 카운트를 유지하는데 이 카운트는 SuspendThread 함수가 호출되면 증가하고 ResumeThread 함수가 호출되면 감소하며 카운트가 0이 되면 스레드는 재개된다. 그래서 SuspendThread를 두 번 호출했다면 ResumeThread도 같이 두 번 호출해 주어야 스레드가 재개된다.

다음 예제는 스레드를 만들고 실행 중에 스레드를 중지/재개 할 수 있는 예제이다.

```
#include <windows.h>
#include <commctrl.h>
DWORD WINAPI foo( void* p){
        HWND hPrg = (HWND)p;

        for ( int i = 0; i < 1000; ++i )
        {
                SendMessage( hPrg, PBM_SETPOS, i, 0); // 프로그래스 전진
                for ( int k = 0; k < 5000000; ++k ); // 0 6개 - some work.!!
        }
        return 0;
}
```

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
          static HWND hPrg;
          static HANDLE hThread;
          switch( msg )
          {
          case WM_CREATE:
                    hPrg = CreateWindow( PROGRESS_CLASS, TEXT(""),

           WS_CHILD | WS_VISIBLE | WS_BORDER | PBS_SMOOTH,
           10, 10, 500,30, hwnd, (HMENU)1, 0, 0);
                    //범위:0 ~1000 초기위치:0으로 초기화.
                    SendMessage( hPrg, PBM_SETRANGE32, 0, 1000);
                    SendMessage( hPrg, PBM_SETPOS,     0, 0);
                    return 0;
          case WM_LBUTTONDOWN:
                    {
          // 새로운 스레드를 만들어서 작업을 시키고 주스레드는 최대한 빨리
                              // 메세지 루프로 돌아 가서 다음 메세지를 처리한다.
                              DWORD tid;
                    hThread = CreateThread( 0, 0, // TKO 보안, Stack 크기
                              foo, (void*)hPrg, // 스레드로 실행할 함수,인자
                              CREATE_SUSPENDED,// 생성하지만 실행은 하지 않는다.
                              &tid);      // 생성된 스레드 ID를 담을 변수
                    //CloseHandle( hThread ); // TKO의 참조개수를 초기에 2이다.
                    // 스레드 종료와 함께 즉시 파괴되도록 1 줄인다.
                    }
                    return 0;
          case WM_RBUTTONDOWN:
                    {
                              static BOOL bRun = FALSE;
```

```
                        bRun = !bRun; // Toggle
                        if ( bRun )
                                ResumeThread( hThread );      // 스레드 재개
                        else
                                SuspendThread( hThread );  // 스레드 일시 중지
                }
                return 0;

        case WM_DESTROY:
                CloseHandle( hThread);
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc( hwnd, msg, wParam, lParam);
}
```

[예제 5-4] Thread 제어

5. 3 스레드 종료

일반적으로 스레드는 일정한 백그라운드 작업을 맡아 처리하고 작업이 완료되면 종료되는 것이 보통이다.

예를 들어 시간이 오래 걸리는 인쇄 작업이나 정렬, 다운로드 작업 등에 스레드가 사용된다. 작업이 종료되면 스레드의 시작함수가 종료되며 이렇게 되면 스레드도 더 이상 필요가 없으므로 파괴된다.

작업 스레드가 백그라운드 작업을 할 때 주 스레드는 작업 스레드를 만들기만 하고 종료 상태에는 별로 관심을 두지 않는 것이 보통이다. 특별한 경우를 제외하고 두 스레드는 서로 독립저으로 실행될 뿐이다. 그러나 주 스레드는 적어도 작업 스레드가 종료되었는지의 여부는 주기적으로 조사해 봐야 하는데 이때는 다음 함수가 사용된다.

BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);

첫 번째 인자로 관심의 대상인 스레드의 핸들을 넘겨주면 두 번째 인자로 이 스레드의 종료 코드를 조사해 준다.

만약 스레드가 실행중이라면 STIL_ACTIVE 가 리턴되며 종료되었다면 스레드 시작 함수가 리턴한 값이나 ExitThread 함수의 인수가 리턴된다. 이 함수로 주 스레드는 작업 스레드의 종료 상태를 조사하여 다음 행동을 결정할 수 있다.

백그라운드 작업을 하는 작업 스레드는 정해진 작업을 순서대로 처리한 후 자연 종료되지만 때로는 작업 중간에 스레드를 종료해야 할 경우도 있다. 예를 들어 다운로드를 받는 스레드를 만들었는데 중간에 사용자가 다운로드를 취소했다면 더 이상 이 스레드는 존재할 필요가 없다. 스레드를 강제 종료할 때는 다음 두 함수가 사용된다.

void ExitThread(DWORD dwExitCode);

void TerminateThread(HANDLE hThread, DWORD dwExitCode);

ExitThread 는 스레드가 스스로를 종료할 때 사용하는 인수로 종료 코드를 넘겨준다. 조료 코드는 주 스레드에서 GetExitCodeThread 함수로 조사할 수 있다. 스레드가 ExitThread를 호출하면 자신의 스택을 해제하고 연결된 DLL을 모두 분리한 후 스스로 파괴된다.

TerminateThread는 스레드 핸들을 인수로 전달받아 해당 스레드를 강제로 종료한다. 즉 다른 스레드를, 예를 들어 주 스레드가 작업 스레드를 강제로 종료하고자 할 때 이 함수가 사용된다.

이 함수는 스레드와 연결된 DLL에게 어떠한 통보도 하지 않으므로 DLL 들이 제대로 종료 처리를 하지 못할 수도 있으며 할당된 자원들이 제대로 해제되지 않을 수도 있다.

이 함수 외에는 다른 방법이 없을 때 등의 위급한 상황에서만 사용되어야 하며 스레드가 어떤 작업을 하고 있는지, 종료 후 어떤 일이 벌어질지를 정확히 알고 있을 때만 사용해야 한다.

스레드를 중간에 종료할 때는 전역변수나 기타 다른 방법을 통해 스레드가 종료 사실을 알 수 있도록 해 주어 ExitThread로 스스로 종료하도록 하는 것이 가장 좋다.

아래는 간단한 스레드 강제 종료 코드이다.

```
#include <windows.h>
#include <commctrl.h>
DWORD WINAPI foo( void* p)
{
	HWND hPrg = (HWND)p;
	for ( int i = 0; i < 1000; ++i )
	{
		SendMessage( hPrg, PBM_SETPOS, i, 0); // 프로그래스 전진
		for ( int k = 0; k < 5000000; ++k ); // 0 6개 - some work.!!
	}
	return 0;
}
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
	static HWND hPrg;
	static HANDLE hThread;
	switch( msg )
	{
	case WM_CREATE:
		hPrg = CreateWindow( PROGRESS_CLASS, TEXT(""),
	 WS_CHILD | WS_VISIBLE | WS_BORDER | PBS_SMOOTH,
	 10, 10, 500,30, hwnd, (HMENU)1, 0, 0);
		//범위:0 ~1000 초기위치:0으로 초기화.
		SendMessage( hPrg, PBM_SETRANGE32, 0, 1000);
		SendMessage( hPrg, PBM_SETPOS,     0, 0);
		return 0;
	case WM_LBUTTONDOWN:
		{
			DWORD tid;
			hThread = CreateThread( 0, 0, // TKO 보안, Stack 크기
```

```
                        foo, (void*)hPrg, // 스레드로 실행할 함수,인자
                                0,
                        &tid);
                }
                return 0;
        case WM_RBUTTONDOWN:
                {
                        DWORD code;
                        GetExitCodeThread(hThread, &code);
                        if( code == STILL_ACTIVE)
                        {
                                TerminateThread(hThread, 100);
                                CloseHandle(hThread);
                        }
                }
                return 0;
        case WM_DESTROY:
                CloseHandle( hThread);
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc( hwnd, msg, wParam, lParam);
}
```

[예제 5-5] Thread 강제 종료

5. 4 _beginThread & TLS

C/C++ 언어로 멀티스레드 프로그램을 만드는 경우에는 CreateThread API로 스레드를 생성하는 경우는 드물다. C/C++로 프로그램을 만들 때는 표준 C라이브러리를 이용하는데, 이 경우에는 CreateThread API를 사용해서는 안 된다고 알려져 있기 때문이다. 멀티스레드에 대응되는 라이브러리에는 스레드 생성 함수도 준비되어 있을 것이므로 이를 사

용해야 한다.

라이브러리 함수를 이용할 때 CreateThread를 이용해서는 안 되는 이유는 다음과 같다.

- 프로세스 내의 스레드는 메모리 공간을 공유하기 때문에 정적변수를 사용할 때 문제가 생길 수 있다.
  예를 들어 표준 C라이브러리 함수의 strtok는 처음 호출했을 때 받은 포인터를 정적변수에 보존하고, 이후에 호출에서는 보존한 값을 참조하게 되어 있다.
  그러나 이 정적변수의 값은 다른 포인터를 지정해서 strtok 함수를 호출하면 덮어쓰기가 된다. 싱글스레드라면 프로그램에서 주의하면 끝나지만, 멀티스레드 환경일 경우에는 언제든지 다른 스레드가 strtok 함수를 호출할지 예측할 수 없다. 자신도 모르는 사이에 포인터가 바뀔 가능성이 있다.

이런 사태를 막기 위해 Windows에서는 스레드마다 다른 메모리 영역을 정적으로 할당하는 구조를 제공한다. 이를 스레드 로컬 스토리지(TLS)라고 한다.

5.4.1 _beginThread

C 런타임 라이브러리 함수는 싱글스레드용과 멀티스레드용으로 다른 라이브러리 파일을 제공한다. 단, TLS를 사용하려면 스레드를 생성할 때 초기화 등의 준비가 필요하다. 갑자기 CreateThread API 를 사용해 스레드를 생성한다면 이런 처리를 할 수 없다. 그래서 스레드 생성용으로 전용 함수를 준비해서 그 안에서 초기화 처리를 한 뒤 스레드를 생성한다.

그 함수가 바로 _beginThread 이다.

unsigned long _beginThreadex(	반환값 : 스레드 핸들
void * __security_atrr,	스레드 속성
unsigned __stksize,	스택의 초기 크기
unsigned (__stdcall *_f)(void*),	스레드 프로시저
void * __arg,	스레드 전달 인자
unsigned __create_flags,	스레드 생성 플래그
unsigned * __thread_id);	스레드 id를 받을 변수의 주소

다음 예제는 _beginThread 사용 예이다.

```
// 컴파일러에서 정의 안했다면 정의
#ifndef _MT
```

```
#define _MT         // MSDN이나 다른 소스에서는 반드시 정의하라고 되어 있는 심볼...
          // 결국 _MT심볼은 컴파일러 옵션에서 /MT 를 지정하는 것과 동일한 효과 ..
#endif
#include <iostream>
#include <process.h>  // _beginthreadex() 를 사용하기 위해..
#include <windows.h>
using namespace std;
unsigned int __stdcall foo(void *p)  // 결국 DWORD WINAPI foo() 이다 ~!!
{
          cout << "foo" << endl;
          Sleep(1000);
          cout << "foo finish" << endl;
          return 0;
}
void main()
{
          unsigned long h = _beginthreadex(0, 0, foo, 0, 0, 0);
          // h가 결국 핸들이다.
          WaitForSingleObject((HANDLE)h, INFINITE);
          CloseHandle((HANDLE)h);
}
```

[예제 5-5] _beginThread

5.4.2 TLS(Thread Local Storage)

TLS(Thread Local Storage)는 스레드 별로 고유한 저장공간을 가질 수 있는 방법이다. 각각의 스레드는 고유한 스택을 갖기 때문에 스택 변수(지역 변수)는 스레드 별로 고유하다.

예를 들어서 각각의 스레드가 같은 함수를 실행한다고 해도 그 함수에서 정의된 지역변수는 실제로 서로 다른 메모리 공간에 위치한다는 의미이다. 그러나 정적 변수나 전역 변

수의 경우에는 프로세스 내의 모든 스레드에 의해서 공유된다.

TLS는 정적, 전역 변수를 각각의 스레드에게 독립적으로 만들어 주고 싶을 때 사용하는 것이다. 다시 말해서, 분명히 같은 문장(context)을 실행하고 있지만 실제로는 스레드 별로 다른 주소공간을 상대로 작업하는 것이다.

```
#include <windows.h>
#include <stdio.h>
// goo 는 static 지역 변수을 사용해서 원하는 기능을 수행하였다.
// 싱글스레드 환경에서는 아무 문제 없다.   멀티 스레드 라면. ?
void goo( char*  name)
{
          // TLS 공간에 변수를 생성한다.
          __declspec(thread) static int c = 0;
//        static int c = 0;
          ++c;
          printf("%s : %d₩n", name, c ); // 함수가 몇번 호출되었는지 알고 싶다.
}
DWORD WINAPI foo( void* p )
{
          char* name = (char*)p;
          goo( name );
          goo( name );
          goo( name );
          return 0;
}
void main()
{
          HANDLE h1 = CreateThread( 0, 0, foo, "A", 0, 0);
          HANDLE h2 = CreateThread( 0, 0, foo, "₩tB", 0, 0);

          HANDLE h[2] = { h1, h2};
```

```
        WaitForMultipleObjects( 2, h, TRUE, INFINITE );
        CloseHandle( h1 );
        CloseHandle( h2 );
}
```

```
[출력 결과]
A : 1
A : 2
A : 3
        B : 1
        B : 2
        B : 3
```

[예제 5-6] TLS

5. 5 스레드 생명 주기

스레드는 생성되어 소멸될 때까지 여러 형태의 생명주기를 가진다.

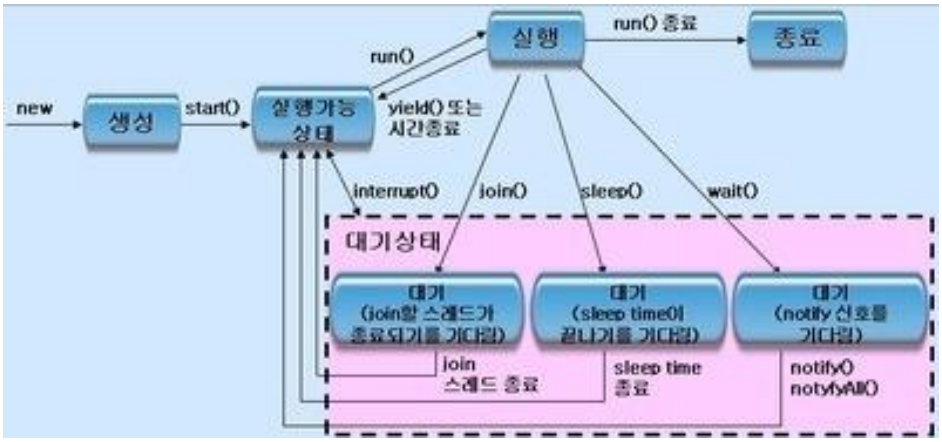
스레드가 생성된 후의 상태는 크게 alived와 dead의 두 가지로 나누어진다. dead 상태는 스레드가 자신의 run() 메소드를 완전히 수행하여 더 수행할 코드가 남아 있지 않거나 stop() 메소드에 의하여 종료되는 경우이다. 나머지 모든 상태는 alive 상태인데 이 상태는 실행 가능 상태, 실행 상태, 대기 상태로 나눌 수 있다.

실행 상태
- 스레드가 CPU를 차지하여 코드를 수행하는 단계

실행 가능 상태
-Runnable pool이라고 하는 특정 장소에 실행 상태로 들어가기 위하여 스레드들이 모여있는 모습

대기 상태
- sleep pool, wait pool, join pool, I/O blocking pool 등

대기 상태에 있는 모든 스레드는 특정 조건을 만족하면 실행 가능 상태로 바뀌며 스케줄러를 통하여 실행 상태로 갈 수 있다.



<스레드의 라이프 사이클>

실행 가능 상태는 스레드가 CPU를 차지하여 실행 상태로 갈 수 있는 상황이다. 실행 가능한 스레드들이 모여서 대기하는 장소를 Runnable Pool이라고 한다. 이곳에는 수많은 스레드들이 모여 서로 경쟁적으로 CPU를 차지하고 실행 상태로 들어갈 기회를 엿보고 있다. 실행 상태로 들어갈 수있는 스레드는 1개이므로 나머지 스레드는 계속 Runnable Pool에서 대기해야 한다.

실행 상태로 들어갈 수 있는 스레드를 선택하는 역할을 수행하는 것이 바로 스케줄러이다. 스케줄러는 JVM 안에서 수행되는 특별한 스레드라고 생각할 수 있다.

스레드의 라이프 사이클을 보면 실행 상태에서 대기 상태로 나오는 여러 가지 상황이 설정되어 있는데 각 상황별로 적용되는 메소드가 다르다.

sleep() 메소드
sleep() 메소드는 현재 실행중인 스레드를 Sleep Pool로 쫓아낸다. sleep() 메소드는 밀리세컨드 단위로 표현된 인자를 가질 수 있는데, 이 시간동안 Sleep Pool에서 대기한다. 시간이 모두 경과되거나 사용자가 인터럽트를 호출하는 경우에 Sleep Pool에서 빠져나올 수 있으며 빠져나온 스레드는 실행 가능 상태가 되어 Runnable Pool로 들어간다. 대부분 sleep() 메소드는 다른 스레드에게 실행 기회를 양보하기 위한 목적으로 쓰인다.

5. 6 스레드 우선 순위

멀티 스레드란 복수 개의 스레드가 동시에 실행되는 시스템이다. 그렇다면 과연 모든 스레드가 동시에 실행되는 것이 가능할까? CPU가 하나라면 멀티 스레드는 동시에 실행되는 것처럼 흉내내는 방법으로 구현된다.

운영체제는 CPU 의 실행시간을 아주 잘게(대략 0.02초) 쪼개어 스레드를 조금씩 조금씩 순서대로 실행함으로써 동시에 실행되는 것처럼 보이게 만든다. 이런 방식을 라운드 로빈(Round Robin) 방식이라 한다.

시스템을 좀 더 효율적으로 운영하기 위해서는 스레드간 우선순위를 정해 긴급하고 사용자에 가까운 스레드에게 좀 더 많은 시간을 주는 것이 합리적이다. 예를 들어 스크린 세이버는 결코 긴급한 프로세스가 아니므로 우선 순위가 낮아도 상관없지만 작업 관리자는 언제든지 사용자가 호출하면 곧바로 실행되어야 하므로 우선 순위가 높아야 한다. 또한 같은 우선 순위를 가진 스레드끼리가도 백그라운드 스레드보다는 포그라운드로 실행되는, 즉 현재 액티브되어 있는 스레드가 더 높은 우선 순위를 가져야 한다.

스레드의 우선 순위는 우선 순위 클래스, 우선 순위 레벨 두 가지 조합으로 결정된다. 우선 순위 클래스는 스레드를 소유한 프로세스의 우선 순위이며 CreateProcess 함수로 프로세스를 생성할 때 dwCreationFlags 로 지정한 값이다. 디폴트는 NORMAL_PRIORITY_CLASS 로 보통 우선 순위를 가지므로 프로세스를 생성할 때 특별히 지정하지 않으면 이 우선 순위가 적용되며 대부분의 응용 프로그램이 그렇다. 우선 순위 레벨은 프로세스 내에서 스레드의 우선 순위를 지정하며 일단 스레드를 생성한 후 아래와 같은 함수로 설정하거나 읽을 수 있다. 자세한 사항은 MSDN을 참고하도록 하자.

Thread Priority(스레드 우선순위)

```
BOOL SetPriorityClass( HANDLE hProcess, DWORD dwPriorityClass );
DWORD GetPriorityClass( HANDLE hProcess );
BOOL SetThreadPriority( HANDLE hThread, int nPriority );
int GetThreadPriority( HANDLE hThread );
```

지정할 수 있는 우선 순위 레벨은 다음 일곱 가지 중 하나이며 디폴트는 보통 우선 순위인 THREAD_PRIORITY_NORMAL 이다.

THEREAD_PRIORITY_IDLE
THEREAD_PRIORITY_LOWEST
THEREAD_PRIORITY_BELOW_NORMAL
THEREAD_PRIORITY_NORMAL
THEREAD_PRIORITY_ABOVE_NORMAL
THEREAD_PRIORITY_HIGHEST
THEREAD_PRIORITY_TIME_CRITICAL

6. 스레드 간 동기화

동기화(Synchronizaion)란 멀티태스킹 환경에서 여러 개의 처리를 서로의 진행 상태에 맞추어 진행시키는 것을 말한다. 전혀 관계가 없는 사이라면 각각을 마음대로 진행해도 문제가 없지만, 관련이 있는 처리를 순서대로 실행할 때나 데이터 교환이 필요한 경우에는 어느 한쪽만 처리를 계속 진행시킬 수 없다. 이때 다른 태스크의 처리를 기다리거나 혹은 기다리게 하는 구조가 동기화다.

동기화의 또 다른 측면은 공유 리소스 접근 제어다. 여기서 말하는 리소스의 종류로는 윈도우, 파일, 메모리 외에도 디스플레이, 비디오 카드, 하드디스크라고 하는 물리 장치도 포함된다. 리소스 중에는 여러 개의 태스크에서 동시에 접근할 때 문제가 발생하는 경우가 있다. 예를 들어, 두 개의 태스크가 같은 파일을 동시에 읽고 쓰는 경우를 생각해 보자. 한쪽에서 읽어오는 동안 다른 쪽에서 그것을 바꿔 쓰면 데이터가 변하기 때문에 전체적으로 데이터의 무결성이 깨질 가능성이 있다.

또 구조체 맴버처럼 서로 관련된 여러 개의 데이터를 하나씩 바꾸는 경우에도 쓰는 도중에 다른 태스크에서 읽는다면 불완전한 데이터를 읽게 된다. 특정 태스크가 리소스에 접근하는 동안에는 다른 태스크가 접근할 수 없게 하는 구조가 필요하다. 이를 해결하는 방법이 바로 동기화이다.

6. 1 동기화가 필요한 이유

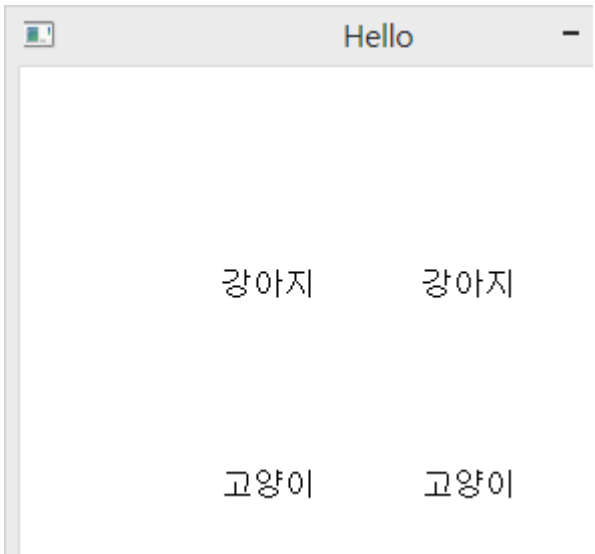
우선 아래 예를 살펴보자.

```
int x;


DWORD WINAPI ThreadFun1(LPVOID param)
{
          HDC hdc = GetDC((HWND)param);
          for(int i=0; i<100; i++)
          {
                    x = 100;
```

```
                    Sleep(1);
                    TextOut(hdc, x, 100, TEXT("강아지"), 3);
          }
          ReleaseDC((HWND)param, hdc);
          return 0;
}
DWORD WINAPI ThreadFun2(LPVOID param)
{
          HDC hdc = GetDC((HWND)param);
          for(int i=0; i<100; i++)
          {
                    x = 200;
                    Sleep(1);
                    TextOut(hdc, x, 200, TEXT("고양이"), 3);
          }
          ReleaseDC((HWND)param, hdc);
          return 0;
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
          switch( msg )
          {
          case WM_LBUTTONDOWN:
                    {
                    DWORD ThreadID;
          HANDLE hThread = CreateThread(NULL, 0, ThreadFun1, hwnd, 0, &ThreadID);
                    CloseHandle(hThread);
          hThread = CreateThread(NULL, 0, ThreadFun2, hwnd, 0, &ThreadID);
                              CloseHandle(hThread);
                    }
```

```
                return 0;
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[출력 결과]



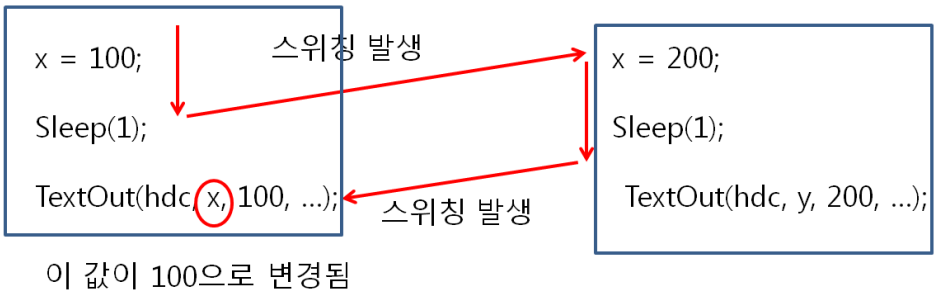
[예제 6-1] 동기화가 필요한 이유

전역 변수 x 는 공유자원이며 두 스레드가 이 변수 하나를 두고 경쟁을 하게 된다. WinProc 에서는 사용자가 마우스 왼쪽 버튼을 누를 때 두 개의 스레드를 생성하는 코드가 있다.

첫 번째 스레드는 100번 루프를 돌며 문자열을 반복 출력한다. 전역변수 x에 100을 대입하고 0.001초간 잠시 쉰 후 (100,100) 에 "강아지"라는 문자열을 출력한다.
두 번째 스레드는 100번 루프를 돌며 문자열을 반복 출력한다. 전역변수 x에 200을 대입하고 0.001초간 잠시 쉰 후 (200,200) 에 "고양이"라는 문자열을 출력한다.

의도한 바와 다른 출력 결과를 볼 수 있다. 이런 틀린 결과가 나오는 이유는 스레드의 실행 순서는 전혀 예측할 수 없으며 프로그래머가 제어할 수도 없기 때문이다. CPU는 시간

을 잘게 쪼개 스레드를 조금씩 나누어 실행할 뿐이며 시간이 경과되면 언제든지 다른 스레드로 스위칭해 버린다.



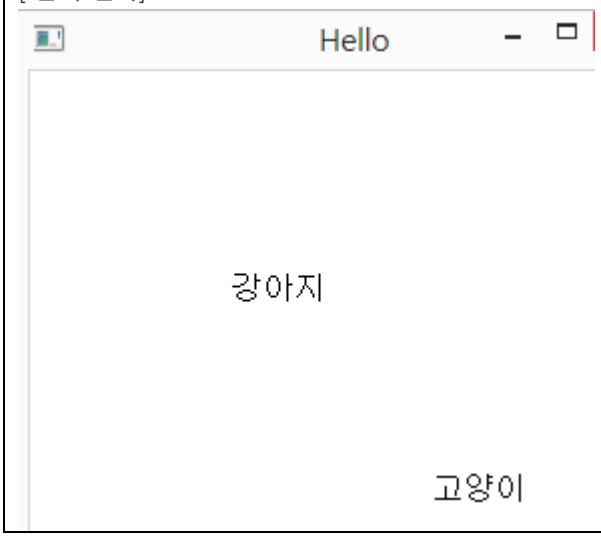
첫 번째 스레드가 x에 100을 대입한 후 문자열이 출력 될때까지 계속 CPU를 차지하면 아무 문제가 없다. 그러나 x에 100을 대입한 직후에 스위칭이 발생하여 두 번째 스레드로 제어가 넘어갔다고 가정하자. 그러면 두 번째 스레드는 x 에 200을 대입하고 문자열을 출력한다. 다시 제어가 첫 번째 스레드로 넘어왔을 때는 x가 변경되어 있기 때문에 (200,100)에 "강아지"를 출력하게 된다.

아래는 위의 문제를 해결한 예이다.

```
int x;
BOOL g_Wait = FALSE;

DWORD WINAPI ThreadFun1(LPVOID param)
{
	HDC hdc = GetDC((HWND)param);
	for(int i=0; i<100; i++)
	{
		while(g_Wait == TRUE);
		g_Wait = TRUE;
		x = 100;
		TextOut(hdc, x, 100, TEXT("강아지"), 3);
		g_Wait = FALSE;
		Sleep(1);
	}
	ReleaseDC((HWND)param, hdc);
```

```
        return 0;
}
DWORD WINAPI ThreadFun2(LPVOID param)
{
        HDC hdc = GetDC((HWND)param);
        for(int i=0; i<100; i++)
        {
                while(g_Wait == TRUE);
                g_Wait = TRUE;
                x = 200;
                TextOut(hdc, x, 200, TEXT("고양이"), 3);
                g_Wait = FALSE;
                Sleep(1);
        }
        ReleaseDC((HWND)param, hdc);
        return 0;
}
```

[출력 결과]



[예제 6-2] 전통적인 방식의 동기화 처리

문제 해결을 위해 또 다른 전역변수 g_Wait를 선언하였다. 해당 변수가 TRUE 인 동안은 전역변수 x를 액세스 하지 못하도록 코드를 작성하였다.

g_Wait의 초기값이 FALSE로 되어 있으므로 최초 실행시는 누구나 X를 엑세스 할 수 있다. 두 스레드 중 하나가 실행되면 g_Wait 를 TRUE로 바꾸어 버린다. 이 상황에서 스위칭이 발생되더라도 g_Wait가 TRUE이기 때문에 전역 변수 x에 접근하지 못하게 된다.

결국 전역변수를 통해 공유자원을 하나의 스레드만 접근할 수 있게 구성하였다. 목적은 무난히 달성되었지만 이 예제는 아주 비효율적이다. 왜냐하면 스레드가 무한루프를 실행하는 동안에도 계속 CPU 시간을 낭비하고 있기 때문이다. 어차피 대기만 할 것이라면 자기에게 주어진 CPU 시간을 더 바쁜 스레드에게 양보하는 것이 훨씬 더 효율적이다.

6. 2 크리티컬 섹션(동기화)

동기화 방법에는 여러 가지가 있는데 그 중에서 크리티컬 섹션이 가장 이해하기 쉽고 속도도 빠르다. 다만 동일한 프로세스 내에서만 사용해야 하는 제약이 있다.

크리티컬 섹션은 "임계 영역" 이라 번역할 수 있는데 정의를 내리자면 공유 자원의 독점을 보장하는 코드의 한 영역이라고 할 수 있다.

크리티컬 섹션은 다음 두 함수로 초기화 및 파괴를 한다.

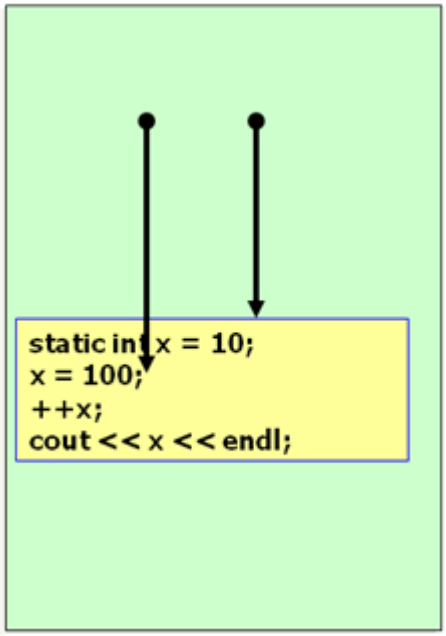
```
void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

둘 다 CRITICAL_SECTION 형의 포인터를 인수로 요구하므로 해당 타입의 변수를 선언하여 주소값을 넘겨주어야 한다. 단 이 변수는 복수 개의 스레드가 참조해야 하므로 반드시 전역 변수로 선언해야 한다.

전역변수로 선언한 후 InitialzeCriticalSection API로 초기화하면 이후부터 크리티컬 섹션을 사용할 수 있다. 물론 다 사용한 후에 DeleteCriticalSection 으로 파괴해 주어야 한다. 다음 두 함수가 실제로 크리티컬 섹션을 구성하는 함수이다.

```
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

보다시피 둘 다 CRITICAL_SECTION 형의 포인터를 인수로 취한다. 이 두 함수 사이의 코드가 바로 크리티컬 섹션, 즉 임계영역이 된다. 다음과 같이 보호될 코드를 두 함수로 감싸준다.




```
EnterCriticalSection(&crit);
  // 이 사이에서 공유 자원을 안전하
  //게 액세스한다.
  LeaveCriticalSection(&crit);
```

EnterCriticalSection API는 이 코드가 크리티컬 섹션을 소유하도록 해 주며 이후부터 다른 스레드는 같은 크리티컬 섹션에 들어올 수 없게 된다. 만약 EnterCritical Section이 호출될 때 이미 다른 스레드가 크리티컬 섹션에 들어와 있으면 이 함수는 크리티컬 섹션이 해제(LeaveCriticalSection)될 때까지 안전하게 대기하도록 한다.

아래 예제를 살펴보자.

```
#include <windows.h>
#include <iostream>
using namespace std;
void WorkFunc()     {          for( int i=0; i <10000000; i++);          }// 시간 지연 함수


// 공유 자원
```

```
int g_x   = 0;
int g_x1 = 0;
//-----------------------------------------------
CRITICAL_SECTION    g_cs;     // 전역
//-----------------------------------------------
DWORD WINAPI Func( PVOID p){
          EnterCriticalSection(&g_cs);
          for( int i=0; i< 20; i++)
          {
                    g_x = 200;
                    WorkFunc();
                    g_x++;
                    cout << "             Func : " << g_x << endl;
          }
          LeaveCriticalSection(&g_cs);
          return 0;
}
void main(){
          InitializeCriticalSection(&g_cs);       // 임계영역 변수 초기화.
          DWORD tid;
          HANDLE hThread = CreateThread(NULL, 0, Func, 0, NORMAL_PRIORITY_CLASS, &tid);
//        Sleep(1);
          EnterCriticalSection(&g_cs);
          for( int i=0; i< 20; i++)     {
                    g_x = 200;
                    WorkFunc();
                    g_x--;
                    cout << "             Main : " << g_x << endl;
          }
          LeaveCriticalSection(&g_cs);
          WaitForSingleObject(hThread, INFINITE);
```

```
        CloseHandle(hThread);
        DeleteCriticalSection(&g_cs); // 파괴
}
```

```
[출력 결과]
...
Main : 199
 Main : 199
 Func : 201
 Func : 201
...
```

[예제 6-3] 크리티컬 섹션

스레드 함수의 반복문이 크리티컬 섹션으로 구성되어 있기 때문에 둘 중 하나의 스레드의 출력이 완료되어야만 다른 스레드의 출력이 된다.

6. 3 동기화 객체

6.3.1 동기화 객체

동기화 객체란 말 그대로 동기화에 사용되는 객체이다. 프로세스, 스레드 처럼 커널 객체이며 프로세스 한정적인 핸들을 가진다. 반면 앞에서 살펴본 크리티컬 섹션은 단순히 데이터 형일 뿐이지 커널 객체는 아니다.

동기화 객체는 크리티컬 섹션보다 느리기는 하지만 훨씬 더 복잡한 동기화에 사용될 수 있다. 동기화 객체는 일정 시점에서 다음 두 가지 상태 중 한 가지 상태를 유지한다.

- 신호상태(Signal) : 스레드의 실행을 허가하는 상태이다. 신호상태의 동기화 객체를 가진 스레드는 계속 실행할 수 있다.

- 비신호상태(Nonsignal) : 스레드의 실행을 허가하지 않는 상태이며 신호상태가 될 때까지 스레드는 블록된다.

동기화 객체는 대기 함수와 함께 사용되는데 대기 함수(Wait Function)는 일정한 조건에

따라 스레드의 실행을 블록하거나 실행을 허가하는 함수이다. 여기서 일정한 조건이란 주로 동기화 객체의 신호 여부가 된다. 몇 가지 부류의 대기함수들이 있는데 다음 함수가 대표적이다.

DWORD WatiForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);

이 함수는 첫 번째 인자가 Signal 상태가 되기를 기다린다. dwMilliseconds 인수는 타임아웃 시간을 1/1000초 단위로 지정하는 데 이 시간이 경과하면 설사 동기화 객체가 비신호 상태이더라도 즉시 리턴된다. 타임 아웃을 INFINITE 로 지정하면 첫 번째 인자가 신호상태가 될 때 까지 무한정 대기한다.

해당 함수의 리턴값은 아래와 같으며, 어떤 이유로 대기상태를 종료했는지를 알 수 있다.

값	설명
WAIT_OBJECT_0	hHandle 객체가 신호상태가 되었다.
WAIT_TIMEOUT	타임 아웃 시간이 경과하였다.
WAIT_ABANDONED	포기된 뮤텍스

WaitForSingleObject 함수는 리턴하기 전에 첫 번째 인자의 상태를 변경한다. 어떻게 변경하는 가는 동기화 객체에 따라 조금씩 다른데 일반적으로는 신호상태의 동기화 객체를 비신호 상태로 변경한다. 따라서 일단 한 스레드가 동기화 객체를 가지면 동기화 객체는 비신호상태가 되므로 다른 스레드가 이 객체를 중복하여 소유하지 못하게 된다.

동기화 객체 종류는 아래와 같으며 주요 동기화 방법을 일단 확인해 보고 넘어가자.

동기화 객체	방법
뮤텍스	뮤텍스 소유권 여부 뮤텍스 소유시 넌시그널 상태가 된다.
세마포어	카운트 카운트로 공유 자원 접근 개수를 제어한다.
이벤트	수동, 자동이벤트 기법이 있으며 알림의 역할을 통해 스레드 순서를 제어할 수도 있다.

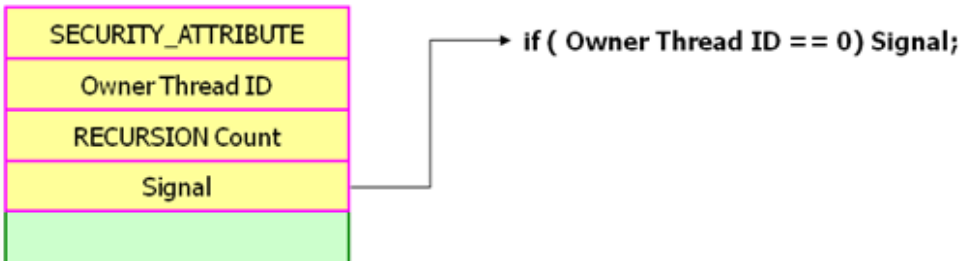
객체별 자세한 내용은 하나씩 살펴보도록 하자.

6.3.2 뮤텍스

뮤텍스는 크리티컬 섹션과 여러 가지 면에서 비슷하므로 크리티컬 섹션이 쓰이는 곳에

대신 사용될 수 있다. 그러나 이름을 가짐으로써 프로세스간에도 사용될 수 있다는 점에 있어 크리티컬 섹션보다는 더 우월한 존재라고 할 수 있다. 그만큼 속도는 느리다 .

뮤텍스는 오직 한 스레드에 의해서만 소유될 수 있으며 일단 어떤 스레드에게 소유되면 비신호상태가 된다. 반대로 어떤 스레드에도 소유되어 있지 않은 상태라면 신호상태가 된다.



뮤텍스를 사용하려면 다음 함수로 뮤텍스를 먼저 생성해야 한다. 이 함수는 뮤텍스를 생성한 후 그 핸들을 리턴해 준다.

HANDLE CreateMutex(LPSECURITY_ATTRIBUTE lpMutex, BOOL bInitialOwner,
LPCTSTR lPName);

첫 번째 인자는 대개의 경우 NULL을 준다.
두 번째 인자는 뮤텍스를 생성함과 동시에 소유할 것인지를 지정하는 데 이 값이 TRUE이면 이 스레드가 뮤텍스를 소유하며 뮤텍스가 비신호 상태로 생성됨으로 다른 스레드는 이 뮤텍스를 소유할 수 없다.

마지막 인수는 뮤텍스의 이름을 지정하는 문자열이다. 뮤텍스는 프로세스끼리의 동기화에도 사용되므로 이름을 가지는 데 이 이름은 프로세스간에 뮤텍스를 공유할 때 사용된다.

뮤텍스에 이름이 있을 경우 다른 프로세스가 일단 뮤텍스의 이름만 알면 다음 함수로 뮤텍스의 핸들을 얻을 수 있다. 만약 동일한 이름으로 CreateMutex를 한 번 더 호출하면 아래 함수처럼 동작하게 된다.

HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle,

LPCTSTR lpName);

만약 뮤텍스를 소유한 상태에서 뮤텍스를 반납하기를 원한다면 아래 함수를 호출하면 된다. 해당 함수가 호출되면 뮤텍스는 비신호 상태에서 신호상태로 변경된다.

BOOL ReleaseMutex(HANDLE hMutex);

생성한 뮤텍스를 파괴할 때는 모든 커널 객체와 마찬가지로 CloseHandle 함수를 사용하면 된다.

예제를 살펴 보자.

```
// 뮤텍스
#include <windows.h>
#include <iostream>
using namespace std;
void main(){
	// 뮤텍스 생성
	HANDLE	hMutex = CreateMutex(NULL,  // 보안속성
		FALSE,	// 생성시 뮤텍스 소유 여부
		TEXT("mutex"));		// 이름
	// 소유가 ture일때 -> Signal 을 nonsignal로 바꾼다.
	cout << "뮤택스를 기다리고 있다." << endl;
	DWORD d = WaitForSingleObject(hMutex, INFINITE);		// 대기
	if( d == WAIT_TIMEOUT)
		MessageBox(NULL, TEXT("WAIT_TIMEOUT"), TEXT(""), MB_OK);
	else if( d ==WAIT_OBJECT_0)
		MessageBox(NULL, TEXT("WAIT_OBJECT_0"), TEXT(""), MB_OK);
	else if( d == WAIT_ABANDONED_0)
		MessageBox(NULL, TEXT("WAIT_ABANDONED_0"), TEXT(""), MB_OK);
	cout << "뮤택스 획득" << endl;
	MessageBox(NULL, TEXT("뮤택스를 놓는다.") , TEXT(""), MB_OK);
	//ReleaseMutex(hMutex);
```

```
}
```

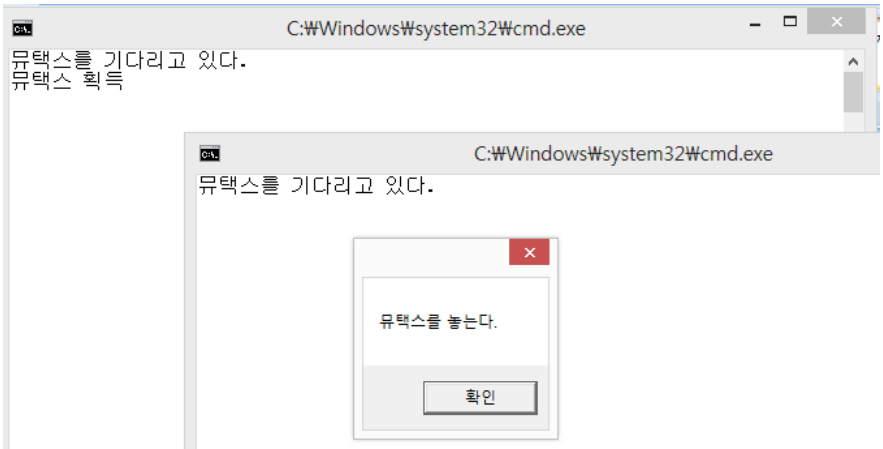
[예제 6-4] 뮤텍스

위 프로그램을 종료하지 않은 상태에서 2번 반복 실행 해보자.

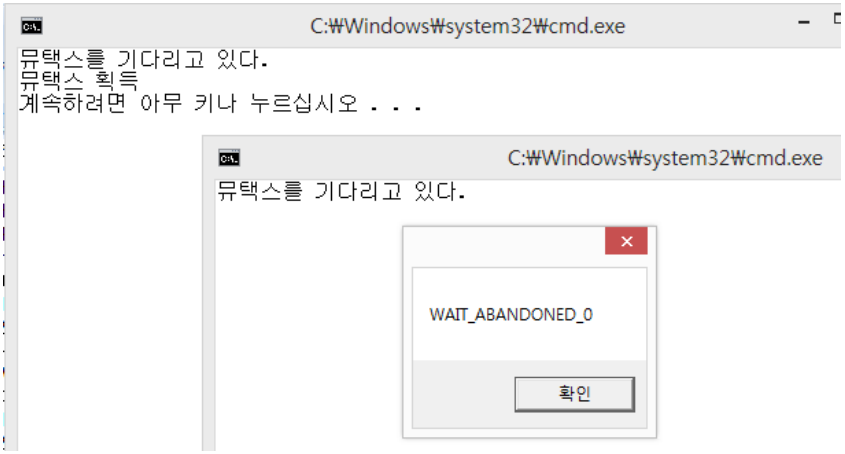
첫 번째 프로그램이 실행될 때는 뮤텍스라 시그널 상태이기 때문에 뮤텍스를 소유하게 된다. 하지만 두 번째 프로그램이 실행될 때는 뮤텍스가 넌 시그널 상태이기 때문에 뮤텍스를 소유하지 못하고 기다리게 된다.

만약 첫 번째 프로그램에서 메시지 박스를("뮤텍스를 놓는다.") 호출하면 ReleaseMutex API를 호출하기 때문에 두 번째 프로세스가 뮤텍스를 소유하게 된다.

아래 그림은 최초 2개의 프로그램을 호출했을 때의 상황이며,



아래 그림은 첫 번째 프로그램이 메시지 박스를 종료 한 후 상황이다.



메시지 박스에 보면 WAIT_ABANDONED_0 의 결과값이 나왔는데 이는 뮤텍스를 소유한 프로세스가 뮤텍스를 ReleaseMutex API를 호출하지 않은 상태에서 종료되면 해당 결과가 발생된다. 이를 포기된 뮤텍스라 한다.

만약 포기된 뮤텍스를 받았다는 것은 스레드 코드에 버그가 있다는 뜻이다. 관련 스레드가 보통 정상 종료되지 못했음을 나타낸다.

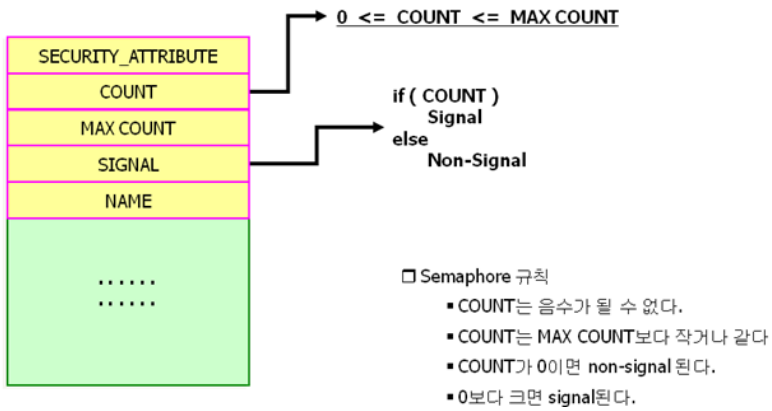
6.3.3 세마포어

세마포어는 뮤텍스와 유사한 동기화 객체이다. 물론 차이점이 있다. 뮤텍스는 하나의 공유 자원을 보호하기 위해 사용하지만 세마포어는 그렇지 않다. 세마포어는 제한된 일정 개수를 가지는 자원을 보호하고 관리한다.

여기서 자원이라 함은 추상적인 개념인데 어떠한 것이라도 가능하다. 하드웨어 일수도 있고 윈도우, 프로세스, 스레드와 같은 소프트웨어적인 것일 수도 있다.

세마포어는 사용가능한 자원의 개수를 카운트하는 동기화 객체이다. 유효 자원이 0이면 즉 하나도 사용할 수 없으면 세마포어는 비신호상태가 되며 1이상이면, 즉 하나라도 사용할 수 있으면 신호상태가 된다.

대기함수는 세마포어가 0이면 스레드를 블록시켜 사용가능한 자원이 생길 때까지, 즉 다른 스레드가 자원을 풀 때까지 대기하도록 하며 세마포어가 1이상이면 유효 자원의 개수를 1 감소시키고 곧바로 리턴한다.



세마포어 관련 함수는 아래와 같다.
세마 포어를 생성하고 생성된 세마포어의 핸들을 획득하는 함수이다.

HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTE lpSemaphoreAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);

HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);

CreateSemaphore 의 첫 번째 인자는 주로 NULL을 사용한다.
두 번째 인자는 초기 카운트 값을 지정한다.
세 번째 인자는 최대 카운트 값을 지정한다.
마지막 인자는 이름이며, 이 인자를 통해 프로세스간 동기화가 가능해 진다.

BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount);

자원의 사용이 끝난 스레드는 위의 함수를 호출하여 사용 종료를 알린다. 뮤텍스의 Release 함수와의 차이점은 뮤텍스에서는 뮤텍스를 소유한 스레드나 프로세스가 해당 함수를 호출할 때만 유효한 기능이 되지만, 세마포어는 누구라도 몇 번이라도 해당 함수를 호출할 수 있다. 따라서 조심해서 사용할 필요가 있다.

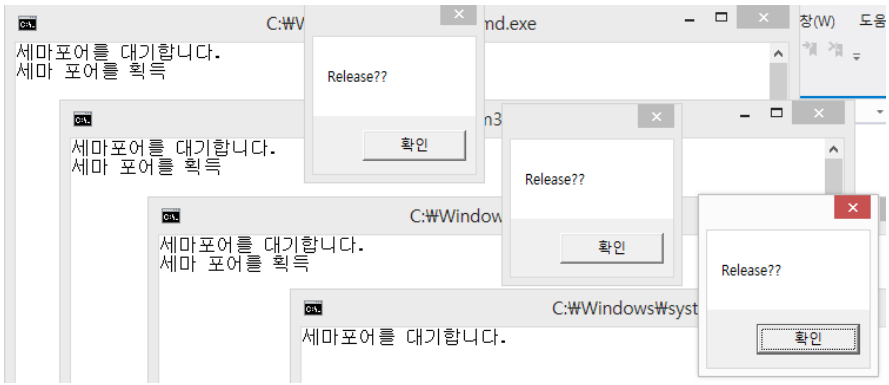
예제를 살펴 보자.

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

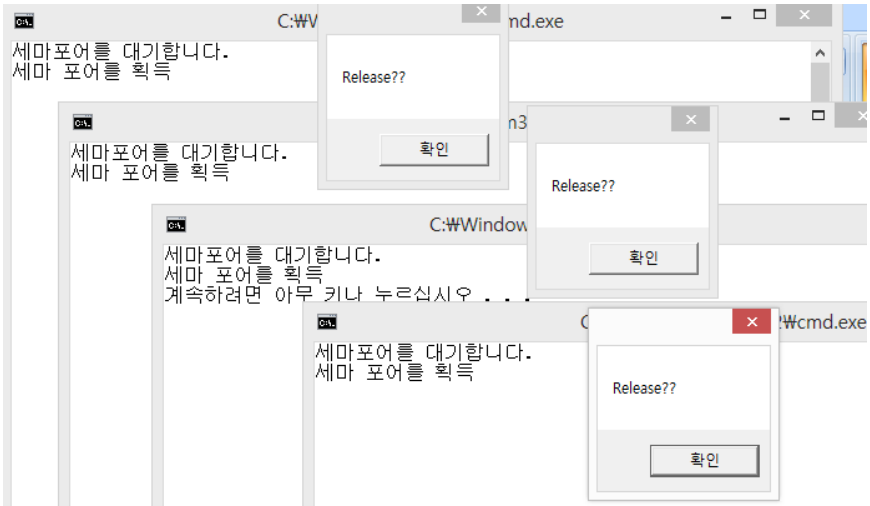
```
#include <stdio.h>

void main()

{

          HANDLE hSemaphore = CreateSemaphore( 0, // 보안

                    3,  // count 초기값

                    3,        // 최대 count

                    TEXT("s")); // 이름

          printf("세마포어를 대기합니다.₩n");

          WaitForSingleObject( hSemaphore, INFINITE ); //--count

          printf("세마 포어를 획득₩n");

          MessageBox(0, TEXT("Release??"),TEXT(""), MB_OK);

          LONG old;

          ReleaseSemaphore( hSemaphore, 1, &old ); // count -= 2nd parameter

          CloseHandle( hSemaphore );

}

// 메세지 Box 의 OK를 누르지 말고.. 4번 이상 실행해 보세요..
```

[예제 6-5] 세마포어

위 프로그램을 4번 이상 실행시키면 아래 그림의 결과를 볼 수 있다.
즉, 3개의 프로그램은 세마포어를 획득했지만, 4번째 실행한 프로그램은 접근하지 못함을 알 수 있다.

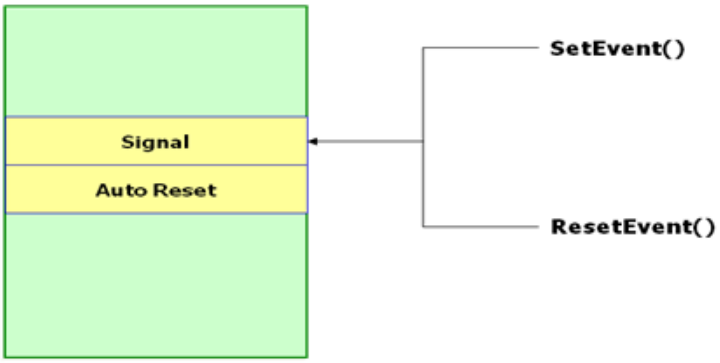


아래의 그림은 임의의 하나의 프로그램에서 메시지 박스를 종료한 후의 상황이다. 대기하고 있던 프로그램이 세마 포어를 획득함을 볼 수 있다.



6.3.4 이벤트

이벤트(EVENT)란 어떤 사건이 일어났음을 알려주는 동기화 객체이다. 크리티컬 섹션, 뮤텍스, 세마포어는 주로 공유 자원을 보호하기 위해 사용하는 데 비해 이벤트는 그보다는 스레드간의 작업 순서나 시기를 조정하기 위해 사용된다.



특정한 조건이 만족될 때까지 대기해야 하는 스레드가 있을 경우 이 스레드의 실행을 이벤트로 제어할 수 있다.

이벤트는 윈도우즈의 메시지와 유사하다. 만약 정렬이나 다운로드가 끝났을 때 이벤트를 보내주어 관련된 다른 작업을 하도록 지시할 수 있다. 이벤트를 기다리는 스레드는 이벤트가 신호상태가 될 때까지 대기하며 신호상태가 되면 대기를 풀고 작업을 시작한다. 이벤트는 리셋되는 방식에 따라 두 가지 종류가 있다.

자동 리셋 이벤트 : 대기 상태가 종료되면 자동으로 비신호상태가 된다.
수동 리셋 이벤트 : 스레드가 비신호상태로 만들어줄 때까지 신호상태를 유지한다.

다음 두 함수는 이벤트를 만들거나 오픈하는 함수이다.

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpEventAttributes,
                   BOOL bManualReset, BOOL bInitialState,
                   LPCTSTR lpName);

HANDLE OpenEvent(DWORD dwDesiredAccess, BOOL bInheriHandle,
                 LPCTSTR lpName);
```

bManualReset 은 이벤트가 수동 리셋 이벤트인지 자동 리셋 이벤트인지를 지정하는 데 TRUE 이면 수동 리셋 이벤트가 된다.

bInitialState 가 TRUE 이면 이벤트를 생성함과 동시에 신호상태로 만들어 이벤트를 기다리는 스레드가 곧바로 실행을 하도록 해 준다. 이벤트도 이름을 가지므로 프로세스간의 동기화에 사용될 수 있다.

이벤트가 뮤텍스나 크리티컬 섹션과 또 다른 점은 대기 함수를 사용하지 않고도 스레드에서 임의적으로 신호상태와 비신호상태를 설정할 수 있다는 점이다. 이 때는 다음 함수를 사용한다.

BOOL SetEvent(HANDLE hEvent):
BOOL ResetEvent(HANDLE hEvent);

수동 리셋 이벤트

대기 함수가 리턴될 때 신호상태를 그대로 유지하며 ResetEvent 함수로 일부러 비신호상태로 만들어 줄 때만 상태가 변경된다.

그래서 여러 개의 스레드가 하나의 이벤트를 기다리고 있더라도 한 번의 신호로 대기하고 있던 모든 스레드가 일제히 작업을 시작할 수 있다. 수동 리셋 이벤트를 다시 비신호상태로 만들어 주어야 하는 시점은 이 이벤트를 기다리는 모든 스레드가 대기 상태에서 풀려났을 때이다. 이런 일을 해 주는 함수가 다음 함수이다.

BOOL PulseEvent(HANDLE hEvent);

이 함수는 SetEvent를 호출하여 이벤트를 신호상태로 만든다. 그리고 대기하던 스레드가 대기 상태를 벗어나면 다시 이벤트를 비신호상태로 만든다.

예제를 살펴보자.

```
#include <iostream>
#include <windows.h>
using namespace std;
void main()
{
        HANDLE    hEvent = CreateEvent(NULL,              //  보안속성
                  TRUE,              // 수동리셋(TRUE), 자동리셋(FALSE)
                  FALSE,             // 초기 상태( NON SIGNAL )
```

```
                    TEXT("e"));                    // 공유할 이벤트 이름
          /*
          AUTO RESET : WaitForSingleObjecct를 통과하는 순간
                                        Signal => NonSignal로 변경시켜 줌.

          MANUAL RESET : WaitForSingleObject를 통화하더라도
                              Signal 상태를 계속 유지함..
          => 기다리던 스레드들을 다 깨워주는 역할..
          */
          cout << "Event를 기다린다." <<endl;
          WaitForSingleObject(hEvent, INFINITE);
          cout << "Event 획득 " << endl;

          cout << "Event를 기다린다." << endl;
          WaitForSingleObject(hEvent, INFINITE);
          cout << "Event 획득" << endl;

          CloseHandle(hEvent);
}
```

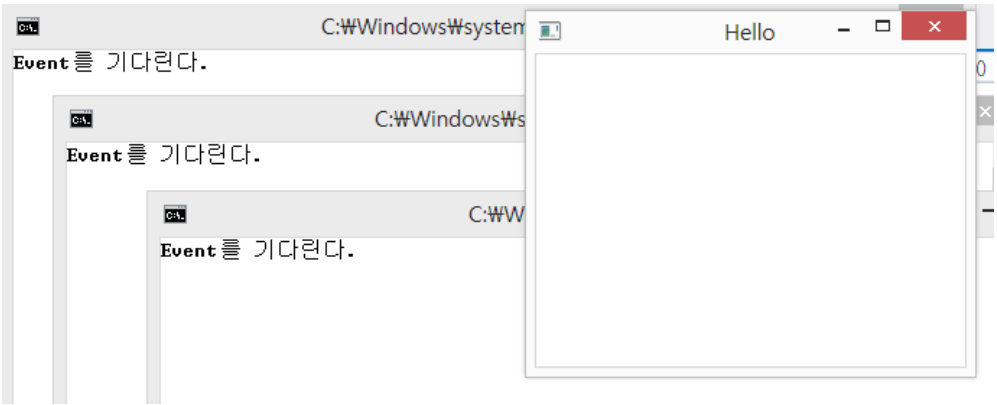
[예제 6-6] 수동 이벤트

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
          switch( msg )
          {
          case WM_LBUTTONDOWN:
                    {
                              HANDLE    h = CreateEvent(NULL, FALSE, TRUE, TEXT("e"));
                              SetEvent(h);                    // Sinal...
//                            PulseEvent(h);
          // Set + ReSet 동시에 날린 개념..
```

```
                    CloseHandle(h);
                }
                return 0;
        case WM_RBUTTONDOWN:
                return 0;
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

[예제 6-7] 이벤트 발생

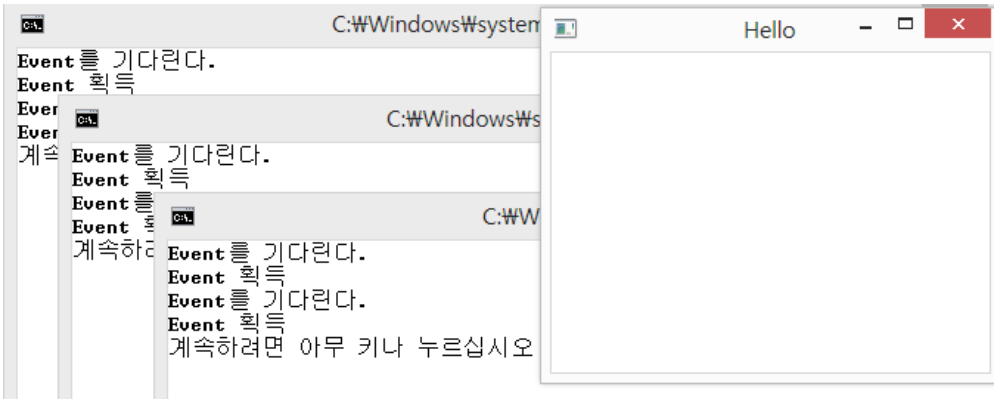
예제 6-6 을 4번 정도 실행시켜보자. 동일한 이벤트 객체를 모든 프로세스가 공유하게 되고, Event 가 넌시그널 상태이기 때문에 모든 프로세스가 대기하게 된다. 아래 그림은 그 결과를 보여준다.



예제 6-7 을 실행시키고 마우스 L 버튼을 클릭하게 되면, SetEvent API를 호출하여 해당 이벤트를 발생시킨다.

SetEvent API 는 이벤트 상태를 Signal 로 변경시켜 주는 역할을 하고 현재 이벤트 객체는 수동 이벤트 객체이기 때문에 대기하고 있던 모든 프로세스들은 대기상태에서 러닝 상태로 변경되게 된다.

아래 그림은 그 결과를 보여 준다.



자동 리셋 이벤트

자동 리셋이라는 말의 의미는 대기 상태를 풀 때 자동으로 리벤트를 비 신호상태로 만든다는 뜻이다. 하나의 스레드만을 위해 이벤트를 사용할 때는 자동 리셋 이벤트를 사용하는 것이 훨씬 편리하며 논리적으로도 별 문제가 되지 않는다.

자동 리셋 이벤트는 기존의 동기화 객체와는 다른 성질을 가지는 데 바로 유일하게 스레드들의 흐름을 제어할 수 있다.

아래 예제를 살펴보자.

```
#include <iostream>
#include <windows.h>
using namespace std;

HANDLE    hEvent1, hEvent2;
BOOL      bContinue = TRUE;
int               g_x,       sum;

DWORD WINAPI      ServerThread( LPVOID p)
```

```
{
	while( bContinue)
	{
		WaitForSingleObject(hEvent1, INFINITE);
		sum = 0;
		for( int i=0; i< g_x; i++)
			sum += i;

		SetEvent(hEvent2);
	}
	cout << "Server종료" << endl;
	return 0;
}

void main()
{
	hEvent1   = CreateEvent(0, 0, 0, TEXT("e1"));
	hEvent2 = CreateEvent(0, 0, 0, TEXT("e2"));
	HANDLE hThread = CreateThread(NULL, NULL, ServerThread, 0, 0, 0);
	while( 1)
	{
		cin >> g_x;
		if( g_x == -1	)	break;
		SetEvent(hEvent1);				// Signal 발생...
		// ... 다른 일 수행
		WaitForSingleObject(hEvent2, INFINITE);
		cout << "결과 : " << sum << endl;
	}
	// 먼저 ServerThread 종료
	bContinue = FALSE;
	SetEvent(hEvent1);
```

```
        WaitForSingleObject(hThread, INFINITE);
        CloseHandle(hThread);
}
```

```
[출력 결과]
50
결과 : 1225
```

[예제 6-8] 자동 이벤트

위의 소스는 메인 스레드와 2nd 스레드와의 흐름을 분석해 보아야 한다.

아래와 같은 흐름이 있다고 가정해 보자.

1) 연산에 필요한 정보를 입력받는다.
2) 연산을 수행한다.
3) 결과를 출력한다.

본 예제는 위의 역할을 2개의 스레드로 분리해서 구현한 예제이다.
먼저 메인 스레드의 역할은 사용자로부터 연산에 필요한 값을 입력받고, 연산된 결과를 출력한다. 2nd 스레드의 경우 메인 스레드에서 생성된 연산에 필요한 값을 가지고 연산을 수행한다.

상호간 스레드의 호출 흐름이 중요하기 때문에 이를 위해 Event 객체를 사용하였다.

6. 4 기타

6.4.1 대기 함수

아래 예제는 WaitForMultipleObjects 함수를 사용하는 예제이다. 기본 성질은 WaitForSingleObject 와 동일하며 몇 가지 차이점만 추가로 이해해 보도록 하자.

```
DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles,
        BOOL bWaitAll,DWORD dwMilliseconds);
```

첫 번째 인자는 대기 하는 핸들의 개수 이며, 최대 64개의 핸들값을 대기할 수 있다.

두 번째 인자는 핸들을 담고 있는 배열의 주소값이다.
세 번째 인자는 함수의 리턴 방식인데, 만약 TRUE 일 경우는 대기 하는 모든 핸들이 시그널 되어야만 리턴되며 FALSE 이면 대기 하는 핸들 중 하나라도 시그널되면 리턴된다.
마지막 인자는 대기 시간으로 기존 WaitForSingleObject 함수와 동일하다.

해당 함수의 리턴값은 아래와 같으며, 어떤 이유로 대기상태를 종료했는지를 알 수 있다.

값	설명
WAIT_OBJECT_0	대기중인 첫번째 hHandle 객체가 신호상태가 되었다.
WAIT_OBJECT_1	대기중인 두번째 hHandle 객체가 신호상태가 되었다.
…	…
WAIT_OBJECT_N	대기중인 N번째 hHandle 객체가 신호상태가 되었다.
WAIT_TIMEOUT	타임 아웃 시간이 경과하였다.
WAIT_ABANDONED	포기된 뮤텍스

```
#include <windows.h>
#include <stdio.h>
void Delay() { for(int i = 0; i< 5000000; ++i); }  // 시간 지연
BOOL bWait = TRUE;
CRITICAL_SECTION cs; // 임계영역 cs 안에는 1개의 스레드만 들어 올수 있다.
DWORD WINAPI foo( void* p )
{
	char* name = (char*)p;
	static int x = 0;

	for ( int i = 0; i < 20; ++i )
	{
		EnterCriticalSection( &cs ); // cs에 들어 간다.
		//-----------------------------------------
		x = 100; Delay();
		x = x+1; Delay();
		printf("%s : %dWn", name, x); Delay();
```

```
                    //-----------------------------------------
                    LeaveCriticalSection( &cs ); // cs에서 나온다.
          }
          printf("Finish...%sWn", name);
          return 0;
}
void main()
{
          InitializeCriticalSection( &cs ); // main 제일아래에
          HANDLE h1 = CreateThread( 0, 0, foo, "A", 0, 0);
          HANDLE h2 = CreateThread( 0, 0, foo, "WtB", 0, 0);
          HANDLE h[2] = { h1, h2 };
          // 64 개 까지의 K0 를 대기할수 있다.
          WaitForMultipleObjects( 2, h, TRUE, // 2개 모두 signal 될때 까지 대기
                              INFINITE );
          CloseHandle( h1 );
          CloseHandle( h2 );
          DeleteCriticalSection(&cs);
}
```

[예제 6-9] WaitMultipeObjets

6.4.2 원자 연산 함수

아래 코드를 살펴보자.

```
#include <windows.h>
long value = 0;
DWORD WINAPI ThreadFunc( void* p)
{
          int i = 0;
          for ( i = 0; i < 10000000; ++i )
```

```
                ++value;
        return 0;
}
void main()
{
        int i = 0;
        HANDLE hThread[5];
        for ( i = 0; i < 5; ++i )
                hThread[i] = CreateThread( 0, 0, ThreadFunc, 0, 0,0);
        WaitForMultipleObjects( 5, hThread, TRUE, INFINITE );
        for ( i = 0; i < 5; ++i )
                CloseHandle( hThread[i]);
        printf("value = %dWn", value);
}
```

```
[출력 결과]
Value = 17415445
```

[예제 6-10] 원자 연산 함수1

스레드 함수인 ThreadFunc 는 전역변수 value 를 천만번 증가(++value) 시킨다.
Main 함수에서는 5개의 스레드를 생성해서 ThreadFunc 함수를 수행한다.
주 스레드는 모든 스레드가 종료된 후 value의 값을 출력 하고 있다.
Value 값은 얼마가 될까 ?

왜 이런 결과가 나왔을까 ?

다음 아래 예제를 살펴보자.

```
#include <windows.h>
long value = 0;
DWORD WINAPI ThreadFunc( void* p)
{
        int i = 0;
```

```
        for ( i = 0; i < 10000000; ++i )
                InterlockedIncrement(&value);
                //++value;
        return 0;
}
void main()
{
        int i = 0;
        HANDLE hThread[5];
        for ( i = 0; i < 5; ++i )
                hThread[i] = CreateThread( 0, 0, ThreadFunc, 0, 0,0);
        WaitForMultipleObjects( 5, hThread, TRUE, INFINITE );
        for ( i = 0; i < 5; ++i )
                CloseHandle( hThread[i]);
        printf("value = %d₩n", value);
}
```

[출력 결과]

Value = 50000000

[예제 6-11] 원자 연산 함수2

6-11 예제는 예상한 결과값이 나온다.

즉 원자 함수란 공유 자원의 연산시 동기화를 함으로써 연산의 안전성을 확보한다.

아래는 다양한 원자 연산 함수들이다. 자세한 내용은 MSDN을 참조해보길 바란다.

LONG InterlockedIncrement(LONG volatile* Addend);
LONGLONG InterlockedIncrement64(LONGLONG volatile* Addend);
LONG InterlockedIncrementAcquire(LONG volatile* Addend);
LONG InterlockedIncrementRelease(LONG volatile* Addend);

LONG InterlockedDecrement(LONG volatile* Addend);
LONGLONG InterlockedDecrement64(LONGLONG volatile* Addend);

LONG InterlockedDecrementAcquire(LONG volatile* Addend);
LONG InterlockedDecrementRelease(LONG volatile* Addend);

LONG InterlockedExchange(LONG volatile* Target, LONG Value);
LONGLONG InterlockedExchange64(LONGLONG volatile* Target, LONGLONG Value);
LONG InterlockedExchangeAdd(LONG volatile* Addend, LONG Value);
LONGLONG InterlockedExchangeAdd64(LONGLONG volatile* Addend, LONGLONG Value);
PVOID InterlockedExchangePointer(PVOID volatile* Target, PVOID Value);

6.4.3 생산자 소비자

아래 코드를 제시하면서 이번장을 마무리 하고자 한다.
전체 구조를 분석해 보기 바란다.

```
#include <windows.h>
#include <iostream>
#include <queue> // STL의 Q
#include <time.h>
using namespace std;
queue<int> Q;                    // 2개의 스레드가 동시에 사용하는 공유 자원
HANDLE hMutex;      // Q에 접근을 동기화 하기 위해 Mutex사용(CRITICAL_SECTION 이
                                        // 더 좋긴 하지만.mutex예제를 위해)
HANDLE hSemaphore;  // Q의 갯수를 Count 하기 위해.
// 생산자
DWORD WINAPI Produce( void* )
{
        static int value = 0;
```

```
        while ( 1 )
        {
                // Q에 생산을 한다.
                ++value;
                // Q의 접근에 대한 독점권을 얻는다.
                WaitForSingleObject( hMutex, INFINITE);
                //-----------------------------------------
                Q.push( value );
                printf("Produce : %d₩n", value );
                LONG old;
                ReleaseSemaphore( hSemaphore, 1, &old); // 세마포어 갯수를 증가한다.
                //-----------------------------------------------
                ReleaseMutex( hMutex );

                Sleep( (rand() % 20 )* 100); // 0.1s ~ 2s간 대기.
        }
        return 0;
}

// 소비자
DWORD WINAPI Consume( void* p)
{
        while ( 1 )
        {
                WaitForSingleObject( hSemaphore, INFINITE ); // Q가 비어 있다면 대기.
                WaitForSingleObject( hMutex, INFINITE );
                //-----------------------------------------
                int n = Q.front();  // Q의 제일 앞요소 얻기(제거하지 않는다.)
                Q.pop();                        // 제거.
                printf("            Consume : %d₩n", n );
                //-----------------------------------------
```

```
                    ReleaseMutex( hMutex );
                    Sleep( (rand()%20)*100); // 0.1s ~ 2s간 대기
          }
          return 0;
}
void main()
{
          hMutex = CreateMutex( 0, FALSE, TEXT("Q_ACCESS_GUARD"));
          hSemaphore = CreateSemaphore( 0, 0, 1000, TEXT("Q_RESOURCE_COUNT")); //최대


                                                  // 1000개의 , 초기 0
          srand( time(0));
          HANDLE h[2];
          h[0] = CreateThread( 0, 0, Produce, 0, 0,0);
          h[1] = CreateThread( 0, 0, Consume, 0, 0,0);
          WaitForMultipleObjects( 2, h, TRUE, INFINITE );
          CloseHandle( h[0] );
          CloseHandle( h[1] );
          CloseHandle( hMutex );
          CloseHandle( hSemaphore );
}
```

[예제 6-12] 생산자 소비자

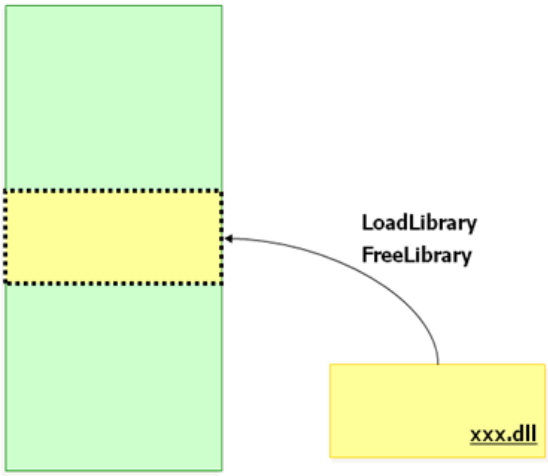
7. DLL

7. 1 DLL 이란?

DLL은 동적 링크 라이브러리(Dynamic Link Library)의 약자로 일반적으로 확장자가 ".DLL " 인 파일이다. '라이브러리'라는 말에서 알 수 있듯이 다른 프로그램에서 이용하는 함수들을 모아둔 것이다. 그러나 표준 C 라이브러리 같은 일반 라이브러리의 파일(.LIB)과는 구조나 사용법이 다소 다르다.

일반 라이브러리는 소스 코드를 컴파일한 결과로 생성되는 객체 파일(.OBJ)을 그대로 모아둔 것이다. 링커는 이 중에서 필요한 함수가 포함된 객체 파일을 꺼내서 실행 파일에 결합하는 것이다. 이를 '정적 링크'라고 한다.

이에 반해 DLL은 '동적 링크(dynamic link)' 에 이용한다. 이는 링크 시에 실행 파일에 결합되는 것이 아니라 프로그램 실행 시에 DLL도 함께 프로그램의 메모리 공간으로 읽어와 호출될 주소 등을 적절하게 바꾸는 것을 말한다. 일단 읽어온 DLL 함수는 프로그램 내부 함수처럼 호출할 수 있다(아래 그림 참조). DLL은 실행 파일과 다른 파일이므로 필요한 시점에 메모리로 읽어오고 불필요하면 메모리에서 없앨 수도 있다.



DLL은 Windows가 작동하는 데 필요한 매우 중요한 구조 중 하나다. 실제 Windows 시스템의 대부분이 DLL로 구성되어 있다고 해도 과언이 아니다. Windows에서 DLL 이 폭넓게 이용되는 이유는 다양한 장점이 있기 때문이다.

- 일단 DLL 코드는 여러 프로그램에서 공유할 수 있다.
정적 라이브러리는 라이브러리 코드가 각각의 실행 파일에 포함되어 있다. 즉, 같은 라이브러리를 사용하는 실행 파일이 두 개 있다면, 각 파일에는 똑같은 코드가 중복된 상태이다.
이에 반해 DLL 코드는 실행 파일에 포함되지 않는다. 그 때마다 독립된 DLL 파일을 로드하는 방법으로 공용하면 된다.

- DLL로 코드를 공유하면 디스크를 절약할 수 있다는 장점이 있다.
실행 파일에 라이브러리 코드가 없기 때문에 프로그램의 파일 크기가 작다. 예를 들어 VC++로 MFC 를 사용해서 Windows 애플리케이션을 생성하는 경우, MFC의 런타임 라이브러리를 정적으로 링크하면 실행 파일 하나가 수 MB를 훌쩍 넘는다. DLL 버전의 런타임 라이브러리를 사용한다면 일반적인 프로그램은 대개 수백 KB 정도면 된다.

- 또한 실행 파일과 독립되어 있기 때문에 DLL의 패치도 간단히 할 수 있다.
정적 링크에서는 실행 파일에 라이브러리 코드가 포함되므로 라이브러리 코드의 버그를 수정하려면 다시 새롭게 링크한 실행 파일을 배포해야 한다. DLL은 인터페이스가 호환되는 한 그 파일을 바꾸는 것만으로 끝난다.

7. 2 DllMain 함수에서 초기화 처리를 한다.

DLL 이 어떤 것인지 이해했다면 이제 만드는 방법을 알아보자. DLL은 단독으로 실행되는 것이 아니기 때문에 main 함수나 WinMain 함수가 필요 없다. 그 대신 DllMain 이라는 함수를 준비해야 한다. Windows 에서는 다음과 같은 상황에서 DLL 초기화나 종료 처리를 위해 DllMain을 호출한다.

1) 프로세스가 DLL을 로드할 때
2) 프로세스가 DLL을 언로드할 때
3) 스레드를 생성할 때(로드 중에만)
4) 스레드를 종료할 때(로드 중에만)

main 함수나 WinMain 함수와 달리 여러 번 호출될 가능성이 있다는 점에 주의하여야 한다.

아래 예제는 전형적인 DllMain 함수의 형태이다.
첫번째 인자에는 로드된 DLL 모듈(인스턴스) 핸들이 저장되었다. DLL을 포함한 리소스를 로드하고 싶을 때 사용할 수 있다. DLL 내부의 함수에서 참조할 것이 있다면 이 예제처럼 정적 변수로 선언하면 된다.

```
static char* buf = 0;
// DLL 핸들(Load 된 주소)
// DllMain이 호출된 이유(4가지 중 1개)
// DLL을 Load한 방법(명시적또는 암시적)
BOOL APIENTRY DllMain(HMODULE  hModule , DWORD ul_reason_for_call, LPVOID lpReserved )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
                    printf("DLL이 프로세스에 Load됩니다. 주소는 : %p₩n", hModule);
                    printf("Load방법은 %s₩n", lpReserved == 0 ? "명시적":"암시적");
                    buf = (char*)malloc( 1000 );
```

```
    case DLL_THREAD_ATTACH: free(buf); printf("DLL이 프로세스에서 해지 됩니다.₩n");break;
    case DLL_THREAD_DETACH:printf("새로운 스레드가 생성됩니다.₩n");   break;
    case DLL_PROCESS_DETACH:printf("스레드가 소멸됩니다.₩n");break;
        break;
    }
    return TRUE;// FALSE를 리턴하면 DLL이 Load되지 않는다.
}
```

[예제 7-1] DllMain

```
#include <stdio.h>
#include <windows.h>
#include <conio.h>
DWORD WINAPI foo( void* p)
{
          printf("foo₩n");Sleep(5000);
          return 0;                                        // DllMain() 호출-
>스레드 종료
}
void main()
{
          getch();
          TCHAR temp[] = TEXT("예제 7-1 DllMain.dll");
          HMODULE hDll = LoadLibrary(temp); // DllMain()호출.
          if ( hDll == 0)
          {
                    printf("DLL을 복사해 오세요₩n");
                    return;
          }
          getch();
          HANDLE h = CreateThread( 0, 0, foo, 0, 0,0); // DllMain()호출
          WaitForSingleObject( h, INFINITE);
```

```
        getch();
        FreeLibrary( hDll );          // DllMain()호출 -> 프로세스에서 해제.
}
```

[예제 7-2] DllMain 사용 예

DllMain 함수를 호출한 이유를 알려면 두 번째 인자를 이용한다.

DLL_PROCESS_ATTACH 와 DLL_PROCESS_DETACH 에서는 프로세스마다 필요한 초기화/종료 처리를 한다. 필요한 메모리를 확보하거나 초기 설정 파일을 읽어 들이는 동작이 이에 해당한다.

DLL_THREAD_ATTACH 와 DLL_THEREAD_DETACH 에서는 스레드마다 필요한 초기화/종료처리를 한다. 만약 TLS 를 사용할 경우에는 여기서 초기화 하면 된다.

특별한 처리가 필요하지 않을 경우에는 아무것도 하지 않고, 즉시 return 문으로 함수를 빠져나와도 상관없다. 단 DLL_PROCESS_ATTACH 의 경우에는 초기화 성공 여부를 반환 값으로 돌려줘야 한다. 여기서 FALSE 를 반환하면 에러가 발생했다고 생각하고 이용하는 쪽의 프로그램 실행이 중지되므로 주의하여야 한다.

세번째 인자는 예전에 '예약된 상태'였지만, 현재는 두번째 인자를 보충하는 정보가 설정된다. DLL이 프로세스 생성 시와 종료 시에 자동적으로 로드 / 언로드 되었을 때(로드 타임 동적 링크) DLL_PROCESS_ATTACH / DLL_PROCESS_DETACH 와 함께 DLL이 아닌 값이 설정된다. 이 정도의 차이 때문에 처리가 나뉘는 경우는 거의 없다.

DLL 을 만드는 데 최소로 필요한 것은 DllMain 함수뿐이다.

7. 3 함수를 익스포트 한다.

DLL 내의 함수를 외부에서 이용하려면 그 함수를 '익스포트' 해야 한다. '익스포트(export)'에는 '수출한다.' 라는 의미가 있지만, 여기서는 인터페이스를 외부에 공개하는 것을 말한다. 구체적으로는 외부 프로그램에서 호출할 함수명이나 메모리에 로드되었을 때의 주소 정보 등을 DLL 파일 안에 넣어둔다. Windows 에서는 이 정보를 기본으로 외부 프로그램에서 오는 호출을 DLL 내부 코드와 연결한다.

함수를 익스포트 하는 방법은 다음과 같다.

1) 모듈 정의 파일을 준비해 EXPORTS 문을 기술한다.

```
EXPORTS
  _GetPersonalInfo
```

2) 소스 코드 내의 함수 정의 __declspec 키워드를 부가한다.

```
extern "C"
_declspec(dllexport)
BOOL GetPersonalInfo( HWND hwndParent, PERSONALINFO * pInfo);
```

여기서는 2)번 방법을 기준으로 코드를 작성하고 설명하고자 한다.

__declspec 키워드를 사용하는 방법에서는 소스 코드 안에서 익스포트하려는 함수 정의에 _declspec(dllexport) 를 추가하면 된다. 컴파일러가 이 키워드를 찾아내면 그 함수를 익스포트하도록 링커에 지시한다.

참고로 __declspec 문에 사용할 수 있는 인수는 다음 네 가지 이다.

인수	설명
thread	TLS 데이터로 지정한다. 이 지정자가 붙은 변수는 해당 스레드에서만 사용할 수 있는 변수가 된다.
naked	접두, 접미를 생성시키지 않는다. 어셈블리 언어를 사용하여 직접 접두, 접미를 달고자 할 때 사용한다. 어셈블리 언어를 사용하여 가상 디바이스 드라이버를 작성 할 때 이 기억부류를 사용한다. 함수에만 적용되며 변수에는 적용되지 않는다.
dllimport	DLL에 있는 데이터, 오브젝트, 함수를 임포트한다. DLL에 있는 이렇게 생긴 함수를 앞으로 사용하겠다는 선언이다.
dllexport	DLL에 있는 데이터, 오브젝트, 함수를 익스포트한다. DLL을 사용하는 클라이언트에게 DLL의 정보를 명시적으로 제공하는 역할을 한다. Dllexport 로 함수를 선언하면 def 파일의 Exports 란에 이 함수를 명시하지 않아도 되며 __export 키워드를 대치한다.

extern "C"

C++ 언어는 작성된 모든 함수에 대한 정보를 밖으로 공개하여 별도의 선언없이 외부에서 함수를 사용할 수 있도록 해준다. 그런데 외부로 공개되는 함수에 대한 정보가 C언어와 C++ 언어의 경우가 다르다.

C++ 언어는 오버로딩 함수라는 문법이 존재한다. 이는 동일한 함수의 이름이지만 매개변수가 다르면 다른 함수로 취급되는 문법이다. C++ 이 공개하는 함수의 정보를 mangled name 이라고 하며 이런 식으로 공개된 함수는 C++ 이외의 다른 언어에서는 사용이 불가능하다.

따라서 extern "C" 지정자는 mangled name을 만들지 않도록 지정함으로써 C형식으로 함수의 정보를 공개하도록 한다.

7. 4 임포트 라이브러리(.lib)

클라이언트에서의 임포트 선언은 함수의 원형이 어떠하다는 것을 밝힐 뿐 그 함수가 어느 DLL에 있는지를 알려주지는 않는다. 그렇다면 클라이언트는 자신이 임포트해야 할 함수가 어느 DLL에 있는지 어떻게 알 수 있을까? 윈도우즈의 system 디렉토리와 현재 디렉토리에 있는 모든 DLL을 열어 함수를 찾아보지는 않을 것이다.

임포트 라이브러리는 DLL에서 엑스포트되는 함수에 대한 정보는 물론 이 함수의 코드가 어떤 DLL에 정의되어 있는지에 대한 정보를 가지고 있다. 함수에 대한 정보만을 가질 뿐이지 함수의 코드는 가지고 있지 않다.

그래서 클라이언트 프로그램은 링크시에 임포트 라이브러리를 링크해야 하며 LIB 파일이 지정하는 DLL 파일을 실행시에 읽어와야 한다. 임포트 라이브러리는 DLL과 같은 이름을 사용하며 확장자 LIB를 가진다. DLL을 만들 때 DLL 과 함께 만들어 주므로 우리는 이 라이브러리를 잘 써먹기만 하면 된다. 참고로 클라이언트 프로그램에서 DLL 파일을 찾는 순서는 다음과 같다.

1) 클라이언트 프로그램이 포함된 디렉토리
2) 프로그램의 현재 디렉토리
3) 윈도우즈의 시스템 디렉토리
4) 윈도우즈 디렉토리
5) PATH 환경 변수가 지정하는 모든 디렉토리

만약 이 순서대로 DLL 을 찾아보고 원하는 DLL이 발견되지 않으면 클라이언트 프로그램은 에러 메시지를 출력하고 실행을 종료한다.

7. 5 DLL 만들기

```
//예제 7-3 MyDll.h
// 이 헤더 한개로 DLL을 만들때와 DLL을 사용할때 모두 사용하기 위해 조건부
// 컴파일 사용
```

```
#pragma once
#ifdef DLL_SOURCE
          #define DLLFUNC     __declspec(dllexport)
#else
          #define DLLFUNC __declspec(dllimport)
#endif
#include <windows.h>
EXTERN_C DLLFUNC void Add( int a, int b);
EXTERN_C DLLFUNC void Sub( int a, int b);
EXTERN_C DLLFUNC void GetResult( int *a);
void foo( int flag);
```

```
// 예제 7-3 MyDll.cpp : Defines the exported functions for the DLL application.
```

```
#include "pch.h"
#include "예제 7-3 MyDll.h"
static int g_result;
void Add( int a, int b)
{
          g_result = a + b;
          foo(1);
```

```
}
void Sub( int a, int b)
{
          g_result = a - b;
          foo(2);
}
void GetResult( int *a)
{
          *a = g_result;
}
void foo( int flag)
{
}
```

[예제 7-3] MyDll

* 소스를 찾기 편리하게 하기 위해 한글 문자를 DLL 명에 추가하였는데 실제 구현시는 영문으로만 DLL 이름을 명명하기 바란다. 한글이 들어가 있는 경우 DLL을 사용하는 프로그램에서 찾지 못할 경우가 발생한다.

위 예제는 .h 파일과 .cpp 파일로 구성되어 있다.

.h 파일에는 전처리 문과 함수의 선언부가 있다. 전처리 문은 DLL_SOURCE 키워드의 define 여부에 따라 DLLFUNC 키워드를 __declspec(dllexport) 혹은 __declspec(dllimport) 로 처리하게 된다. 이러한 부분을 추가한 이유는 .h 파일은 DLL 의 h 이기도 하지만 해당 DLL을 사용하는 프로그램도 공통으로 해당 h 파일을 사용 가능토록 하기 위해서이다.

cpp 파일을 보면 제일 상단에 #define DLLFUNC 전처리 문을 볼 수 있다. 따라서 DLL 안에서는 __declspec(dllexport) 키워드로 처리가 되게 되며 함수가 노출되게 될 것이다. 하지만 DLL을 사용하는 프로그램에서 #define DLLFUNC 전처리 문이 없으면 __declspec(dllimport) 키워드로 처리되게 될 것이다.

또한 위의 DLL 은 3개의 함수가 노출되어 있고 하나의 함수는 외부에서 접근할 수 없도록 구성하였다.

이 프로젝트를 컴파일 하면 Debug 디렉토리에 .dll 파일과 .lib 파일이 생성되어 있을 것이다. DLL은 단독으로 실행할 수 없으므로 실행 여부를 판단할 수 없다.

7. 6 DLL 사용하기

7.6.1 묵시적 연결

앞에서 만든 DLL을 사용하는 예제를 만들어 보도록 하자.
묵시적 연결(Implicit) 이란 함수가 어느 DLL에 있는지 밝히지 않고 그냥 사용한다.

프로젝트에 임포트 라이브러리를 포함해 주어야 하며 윈도우즈는 임포트 라이브러리의 정보를 참조하여 알아서 DLL을 로드하고 함수를 찾아준다. 클라이언트 프로그램이 로드될 때 DLL이 같이 로드되거나 이미 DLL이 로드되어 있으면 사용 카운트를 1 증가시킨다.

클라이언트 프로그램이 실행될 때 DLL이 로드되므로 실행시 연결(LoadTime Linking)이라고도 한다.

묵시적 연결을 하기 위해서는 다음과 같은 DLL 정보 파일이 필요하다.

1) 함수의 선언부를 알기 위한 .h 파일
2) DLL 정보를 위해 필요한 .lib 파일
3) 함수의 정의부를 가지고 있는 .dll 파일

우선 위의 3가지 파일을 DLL 예제에서 복사하여 아래 예제의 디렉토리에 추가한 후 실행시켜보도록 하자.

```
#include <windows.h> // user32.dll의 모든 함수 선언.
#include <stdio.h>

// DLL 사용하기
#include "예제 7-3 MyDll.h"
          // 관련 헤더 include
#pragma comment(lib, "예제 7-3 MyDll.lib")                    // 라이브러리 추가

void main()
```

```
{
        Add(10,20);                             // DLL 함수 사용
        int value;
        GetResult(&value);
        printf("%d₩n", value);
}
```

[예제 7-4] UseMyDll1

7.6.2 명시적 연결

DLL을 명시적으로 읽고 사용하는 데는 다음 세 가지 함수가 사용된다. 일단 코드를 작성하기 전에 개별 함수들에 대해 알아 보자.

HINSTANCE LoadLibrary(LPCTSTR lpLibFileName);

사용하고자 하는 DLL을 메모리로 읽어와 현재 프로세스의 메모리 영역에 DLL을 맵핑시켜 사용할 수 있도록 해 주되 DLL이 이미 메모리에 올라와 있는 상태라면 사용 카운트만 1 증가시킨다.

인수로는 읽고자 하는 DLL의 파일 이름을 명시하는 널 종료 문자열을 주며 보통 경로는 생략한다. 경로 생략시 DLL을 찾는 순서에 따라 경로를 뒤져보고 찾아낸다.

1) 현재 디렉토리
2) 윈도우즈 시스템 디렉토리
3) 윈도우즈 디렉토리
4) PATH 환경 변수가 지정하는 디렉토리

경로를 지정한 경우는 물론 지정한 경로에서 DLL을 찾되 이렇게 하면 지정한 경로에 DLL이 꼭 있어야 하므로 경로는 지정하지 않는 것이 좋다.

확장자를 생략할 경우 디폴트 확장자인 .DLL이 적용된다. DLL을 읽어오는데 성공하면 DLL의 모듈 핸들을 리턴하며 이 핸들은 GetProcAddress 함수에서 사용된다. 에러 발생시 NULL 을 리턴한다.

FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);

DLL 에서 엑스포트한 함수의 번지를 찾아 그 함수의 함수 포인터를 리턴해 준다.
첫 번째 인수 hModule 은 함수가 포함된 DLL의 모듈 핸들이며 LoadLibrary 함수가 리턴해 준 값이다.

두 번째 인수 lpProcName은 찾고자 하는 함수 이름을 지정하는 널 종료문자열이거나 함수의 서수값이다. 널 종료 문자열로 함수의 이름을 명시할 경우는 특히 대소문자 구분에 유의하여야 한다. C언어는 대소 문자를 철저하게 구분한다는 점을 명심하자.

이 함수가 리턴한 함수 포인터를 사용하여 DLL에 있는 함수를 호출한다. 함수를 찾을 수 없다거나 DLL의 핸들이 무효라든가 하는 에러 발생시는 NULL을 리턴한다.

BOOL FreeLibrary(HMODULE hLibModule);

DLL 의 사용 카운트를 1 감소시키며 사용 카운트가 0이 되었을 경우 메모리에서 DLL을 삭제한다. hLibModule 인수는 LoadLibrary 가 리턴한 DLL의 모듈 핸들이다. LoadLibrary 로 DLL을 읽어왔다면 프로세스를 종료하기 전에 반드시 이 함수를 호출해 주어 DLL을 해제해 주어야 한다. 그렇지 않으면 사용되지도 않는 DLL이 메모리에서 계속 남겨 있게 되므로 불필요하게 메모리를 낭비하게 된다.

에러 발생시는 FALSE 를 리턴한다.

앞 서 만든 DLL 을 명시적 방법으로 로딩하도록 하자.
명시적 연결을 하기 위해서는 DLL 파일만 있으면 된다.

1) 함수의 정의부를 가지고 있는 .dll 파일

```
#include <windows.h>
#include <conio.h>
#include <stdio.h>
// DLL의 명시적 연결은 헤더와 라이브러리 파일이 필요 없다.
// DLL 자체만 있으면 사용 가능 하다. - 단 함수의 signature를 미리 알고 있어야 한다.
typedef void(*FUNC)(int, int);
typedef void(*FUNC1)(int*);
```

```
void main()
{
	getch();
	HMODULE hDll = LoadLibrary(TEXT("예제 7-3 MyDll.dll"));
	if ( hDll == 0 )
	{
		printf("dll 을 찾을수가 없습니다.₩n");
		return;
	}
	printf("DLL이 Load된 주소 : %p₩n", hDll );
	//-----------------------------------------------
	FUNC Add = (FUNC)GetProcAddress( hDll, "Add");
	FUNC Sub = (FUNC)GetProcAddress(hDll, "Sub");
	FUNC1 GetResult = (FUNC1)GetProcAddress(hDll, "GetResult");

	if ( Add == 0 || Sub == 0 || GetResult == 0)
		printf("DLL 안에서 해당 함수를 찾을수가 없습니다.₩n");
	else
	{
		int  value;
		Add(10, 20);
		GetResult(&value);
		printf("%d₩n", value);
	}

	FreeLibrary( hDll ); // DLL 해지
}
```

[예제 7-5] UseMyDll2

7.7 클래스 엑스포트

DLL이 엑스포트하는 주된 대상은 함수이지만 함수 외에도 타입이나 변수, 클래스 따위를 엑스포트 할 수 있다. 대개의 경우 변수는 DLL이 직접적으로 엑스포트하지 않는 것이 보통인데 왜냐하면 변수를 클라이언트가 직접 사용할 경우 호환성에 문제가 많고 여로 모로 위험해질 수도 있기 때문이다.

간단한 예를 들자면 int 타입이 모든 언어에서 같은 크기를 가진다고 보장할 수 없으며 심지어 한 언어에서조차도 버전에 따라 크기가 달라진다. 게다가 언어별로 지원하는 타입의 종류가 다양해 호환성을 확보하기가 어렵다.

만약, 클라이언트에게 어떤 값을 꼭 엑스포트해야 한다면 변수를 읽고 쓰는 Get, Set 함수를 대신 엑스포트하는 것이 바람직하다. 이렇게 되면 미리 정해진 규칙대로만 변수를 엑세스할 수 있으므로 구조상 더 안전하고 DLL이 내부적인 타입을 숨길 수 있다. 객체 지향 개념으로 표현하자면 일종의 추상화인데 클라이언트는 변수의 구체적인 형태나 규칙에 대해 알 필요가 없도록 함으로써 DLL의 독립성과 범용성을 높일 수 있다.

C++ 클래스도 DLL에 의해 엑스포트 될 수 있는데 방법은 함수의 경우와 동일하다. 클래스는 일종의 타입이므로 타입 선언문에 __declspec(dllexport) 지시자를 붙이면 이 타입이 외부로 엑스포트되며 쓰는 쪽에서는 __declspec(dllimport) 지시자를 붙이면 된다. 클래스는 C++ 언어 수준에서 제공되는 것이므로 extern “C” 는 붙이지 말아야 한다. 다음 예제는 정수를 캡슐화하는 CInt 클래스를 DLL로 제공한다.

예제 7-6 CIntDll.h

```
#ifndef DLLEXPORT
#define CINTDLL __declspec(dllexport)
#else
#define CINTDLL __declspec(dllimport)
#endif
class CINTDLL CInt
{
private:
        int i;
public:
        CInt(int _i) : i(_i) {}
        void Inc();
```

```
        void Dec();
        int GetValue() const;
        void SetValue(int _i);
};
```

예제 7-6 CIntDll.cpp

```
#define DLLEXPORT
#include "예제 7-6 CIntDll.h"
void CInt::Inc()
{
        i++;
}
void CInt::Dec()
{
        i--;
}
int CInt::GetValue() const
{
        return i;
}
void CInt::SetValue(int _i)
{
        i = _i;
}
```

[예제 7-6] CIntDll

CppDll.h 를 include 하기 전에 클래스 선언문을 엑스포트하도록 DLLEXPORT 매크로를 반드시 정의해야 한다.

컴파일된 DLL과 임포트 라이브러리, 그리고 헤더 파일을 같이 배포하면 클라이언트에서 이 클래스를 사용할 수 있다.

간단한 테스트 예제를 만들어 보자.

```
#include <stdio.h>
#include "예제 7-6 CIntDll.h"
#pragma comment(lib, "예제 7-6 CIntDll.lib")
void main()
{
                    CInt *p = new CInt(10);
                    p->Inc();
                    printf("%d", p->GetValue());
                    p->SetValue(100);
                    p->Dec();
                    printf("%d", p->GetValue());
}
```

[예제 7-7] UseCInitDll

CppDll.h 를 include 하기만 하면 CInt 클래스가 임포트 선언되며 이 클래스 타입과 맴버 함수를 바로 사용할 수 있다.

이 예제에서 보다시피 클래스를 엑스포트 하는 것도 문법적으로 별 어려움은 없다. 그러나 이렇게 만들어진 DLL은 C++ 언어로만 사용할 수 있다는 제약이 생겨 바람직하지 못하다. 클래스라는 것이 C++ 언어의 고유한 기능이라 다른 언어에서는 이 클래스를 사용하기 어렵다. 물론 C++ 이외에도 객체지향을 지원하는 언어들이 많지만 각 언어마다 클래스를 표현하는 구문이나 컴파일된 결과들이 다를 수 있으므로 범용적인 사용은 힘들것이다.

7. 8 Win32 DLL 고찰

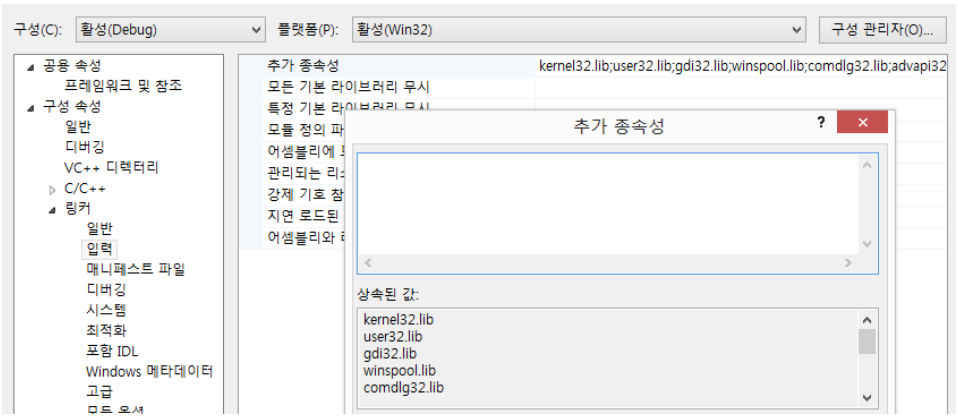
Win32 API 는 수많은 함수와 구조체 타입 들로 구성되어 있다. 그리고 중요한 3가지 DLL에 90% 이상의 함수들을 담아서 제공하고 있다.

우리는 지금까지 Win32 에서 제공하는 DLL 형태의 라이브러리를 암시적 방법으로 사용해 왔었다.

아래의 예제를 살펴보자.

Windows.h 파일은 Win32 API 의 h 파일들을 include 하고 있다. 따라서 우리는 Windows.h 파일만 include 하면 gdi, user, kernel dll 에 있는 모든 함수들의 선언부를 가져 올 수 있었다.

그렇다면 관련 lib 파일은 어떻게 프로젝트에 포함되었던 것일까?
아래의 그림은 프로젝트 속성 페이지 이다.
console 또는 Windows 프로젝트를 구성하게 되면 기본으로 아래 그림과 같이 필요한 lib 파일의 정보들이 들어가 있음을 볼 수 있다.



또한 아래의 예제를 살펴 보면, 실제 로딩된 dll 들의 정보를 얻어 올 수 있다.

```
#include <windows.h> // user32.dll의 모든 함수 선언.
#pragma comment(lib, "user32.lib")
#include <stdio.h>

void main()
{
        MessageBox(0,TEXT("A"),TEXT(""),MB_OK);

        HINSTANCE hModule = GetModuleHandle(TEXT("user32.dll"));
```

```
        printf("user32.dll : 0x%p₩n", hModule);


        hModule = GetModuleHandle(TEXT("gdi32.dll"));
        printf("gdi32.dll : 0x%p₩n", hModule);
        hModule = GetModuleHandle(TEXT("kernel32.dll"));
        printf("kernel32.dll : 0x%p₩n", hModule);
        hModule = GetModuleHandle(0);
        printf("exe Instnace : 0x%p₩n", hModule);
}
```

[예제 7-8] Win32API

8. 에러 핸들링(예외 처리)

8. 1 고전적인 에러 처리 방법

아래에 간단한 원시적인 에러가 있다.

```
If ( i == 3)        // I 가 대문자
foo()               // 함수 호출시 세미코론이 빠져있슴
arr[10] = {1,2};    // 배열의 잘못된 사용법
```

그런데 이런 에러는 컴파일러가 컴파일 중에 골라 낼 수 있기 때문에 별로 문제가 되지 않는다. 더 문제가 되는 것은 문법적으로는 분명히 맞지만 논리적으로 잘못된 코드인데 이런 코드를 버그라고 한다.

버그없는 프로그램을 짜기 위해서는 많은 노력과 오랜 시간이 필요하다. 하지만 버그를 완전히 소탕했다고 해도 프로그램은 항상 에러를 일으킬 소지를 가지고 있다. 어떤 경우가 그런지 보자.

- 존재하지 않는 파일을 열려고 했다.
- 포인터가 빈 메모리를 가리키고 있다.
- 리소스나 메모리가 부족하여 작업을 계속 진행할 수 없다.
- 하드 디스크에 물리적인 손상이 가서 사용할 수가 없다.

이 외에도 프로그램의 정상적인 실행을 방해하는 많은 요소들이 있을 수 있다. 이 장에서 논하는 에러란 바로 이런 종류의 에러를 말한다.

프로그램을 아무리 잘 짜도 외부적인 요인에 의해 발생되는 에러는 프로그래밍 중에는 계산에 포함시키기가 어렵다.

에러를 대처하는 전통적인 방식은 다음과 같이 코드를 작성하는 것이다.

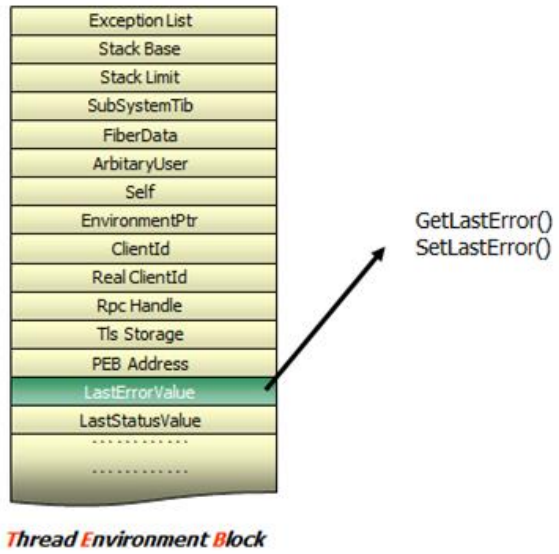
```
커널 핸들 = 커널 오브젝트 생성
if ( 커널 핸들 != 오류값 )
{
   // 정상 처리
}
else
{
    //오류 처리
}
```

이런 전통적인 에러 처리 방법은 오랫동안 사용되어 온 것이지만 아주 불편하고 비효율적이다. 때로는 조건문으로 에러를 적절히 처리할 수 없는 경우도 있다. 그래서 Win32 에서는 에러만을 전문적으로 처리할 수 있는 메커니즘을 운영체제 차원에서 제공하고 있다.

8. 2 GetLastError

에러가 발생할 소지가 있는 모든 API 함수들은 리턴값으로 에러 유무를 리턴해 준다. 예를 들어 CreateWindow 함수는 윈도우 생성에 성공했으면 새로 만든 윈도우 핸들을 리턴해 주지만 어떠한 문제로 인해 윈도우를 생성할 수 없으면 NULL 을 리턴하여 에러가 발생했음을 알려준다. CreateFile, GetDC 등의 함수들도 마찬가지다.

그러나 함수의 리턴값으로는 에러 발생 사실만 알려줄 뿐 구체적으로 어떤 에러가 발생했는지에 대한 자세한 정보는 알려주지 못한다. 발생한



에러의 종류는 다음 함수로 조사할 수 있다.

DWORD GetLastError(void);

스레드는 최후 에러 코드를 기억하고 있으며 API 함수들은 에러 발생시 어떤 종류의 에러가 발생했다는 것을 최후 에러 코드에 설정해 놓는다. 그러면 응용 프로그램에서는 GetLastError 함수로 이 최후 에러 코드를 읽어 발생한 에러의 종류를 조사할 수 있다. 즉 시스템에서 다양한 에러 코드를 미리 정의해 놓았다.

전통적인 에러처리와는 달리 에러 발생시 GetLastError API 를 호출하여 어떤 에러가 발생했는지 조사해 보고 발생한 에러 종류에 따라 적절한 조치를 취하면 된다.

8. 3 FormatMessage

파일이 읽기 전용이라 기록이 불가능하면 읽기 전용 속성을 해제한 후 다시 시도를 할 수 있고 디스크 용량이 부족하다면 휴지통을 비울 수도 있다. 만약 에러 발생 사실만 사용자에게 알려주려면 다음 함수로 에러 메시지 문자열을 구할 수 있다.

DWORD FormatMessage(DWORD dwFlags, LPCVOID lpSource,
DWORD dwMessageId, DWORD dwLanguageId, LPTSTR lpBuffer,

DWORD nSize, va_list *Arguments);

에러 메시지 문자열은 시스켐에 정의되어 있기도 하지만 응용 프로그램이 리소스에 테이블 형태로 정의할 수도 있고 이 함수로 즉시 조립할 수도 있다. 또한 이 함수는 메시지 버퍼를 내부에서 할당하기도 하고 서식 문자열도 지원하기 때문에 원형이 굉장히 복잡하고 사용방법도 어렵다.

간단한 사용예이다.

```
FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
                                    에러코드, 0, 버퍼, 버퍼길이, NULL);
```

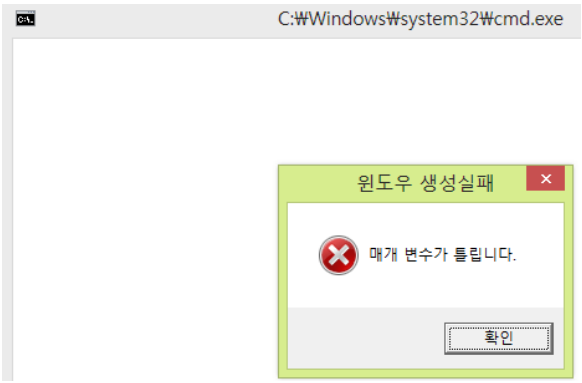
세 번째 인수에 GetLastError 로 조사한 에로 코드, 다섯 번째 인수에 에러 메시지를 돌려받을 버퍼, 여섯번째 인수에 버퍼 길이를 전달하면 시스템이 정의하는 에러 메시지 문자열을 구할 수 있다.

아래 예제를 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 );
}
void main(  )
{
    HWND hwnd = CreateWindowEx(0,0,0,0,0,0,0,0,0,0,0,0);
```

```
    if ( hwnd == 0)
        ReportError( TEXT("윈도우 생성실패") );
}
```

[예제 8-1] FormatMessage1



8. 4 C언어에서의 에러 처리

C런타임 라이브러리도 에러 코드값 및 에러 문자열을 획득하는 함수가 제공된다.

단, API에서 제공되는 GetLasetError API 와의 차이점이 있는데, GetLastError API 는 스레드 당 에로 코드 값이 존재하고 각각의 스레드에서 GetLastError API 를 호출하면 자신의 스레드에 등록되어 있는 에러 코드값을 획득한다.

C언어에서 제공되는 errno 라는 변수 값은 프로세스 당 1개의 에러 코드 값이다. 따라서 멀티 스레드 환경에서는 사용할 수 없다는 단점을 가지고 있다.

strerror() 함수는 에러 코드값을 인자로 전달하면 문자열로 에러 정보를 리턴해 준다.

perror() 함수는 통합된 형태로 내부적으로 에러 코드 값을 획득하고 획득한 에러 코드값에 해당되는 문자열을 출력해 준다.

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

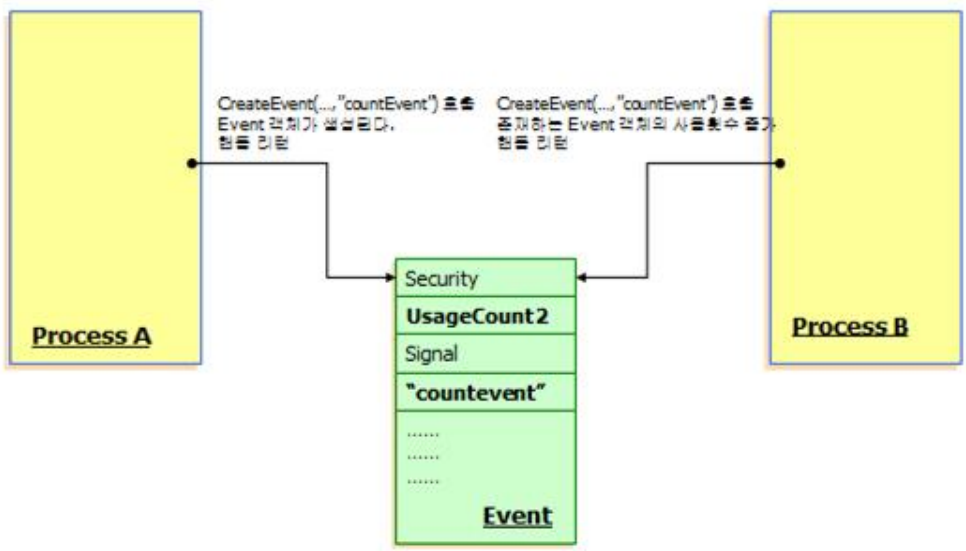
```
#include <conio.h>


// 전통적인 C 의 에러 처리 방법 -
// 단점 프로세스당 1개의 에러 코드.
void main()
{
	FILE* f = fopen("a.txt", "rt");
	if ( f == 0 )
	{
		printf("실패 ₩n");
		// 1. errno : C 언어가 미리 정의한 전역 변수.
		printf("errno : %d₩n", errno );
		// 2. errno-> 문자열로!
		char* s = strerror( errno );
		printf("%s₩n", s );
		// 1.2통합 fprintf(stderror, "Fail : %s₩n", strerror(errno));
		perror("Fail");
	}
}
```

```
[출력 결과]
실패
errno : 2
No such file or directory
Fail: No such file or directory
```

[예제 8-2] strerror

8. 5 활용 예제

함수호출이 실패 했을 때만 GetLastError()를 호출하는 것은 아니다. 때로는 함수 호출이 성공을 하고 원하는 결과를 얻었을때는 GetLastError()을 호출할 수 있다.



위 그림을 보자.

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 사용 횟수가 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.

만약, 오직 한 개의 인스턴스 만을 생성할 수 있는 프로그램을 만들어야 하는 문제가 주어졌다고 가정하자 하자. 실제 그러한 프로그램은 많이 있다. 예를 들어 MSN 메신저나 네이트온 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.

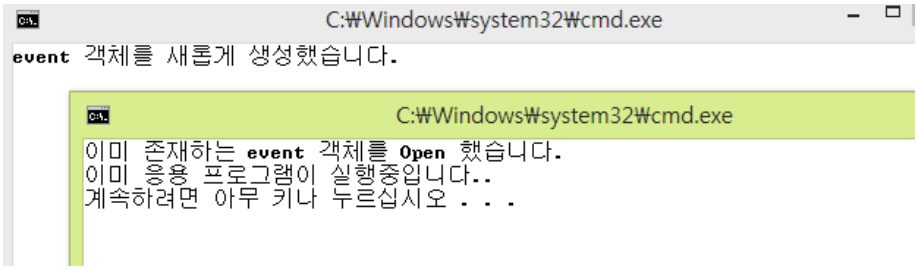
```c
#include <stdio.h>
#include <windows.h>
#include <errno.h>
#include <conio.h>

// 함수 호출이 성공했을때 성공의 이유를 알기 위해서
// GetLastError()를 호출할 때가 있다.
//  - 항상그런것은 아니다.
// 동시에 동일한 2개의 프로세스가 생성되는 것을 막는 방법.
void main()
{
	HANDLE hEvent = CreateEvent( 0, 0, 0,TEXT("e"));

	if ( hEvent != 0 ) // 성공..!!
	{
		DWORD e = GetLastError(); // 왜 성공했는지 이유를 알고 싶다.

		if ( e == ERROR_ALREADY_EXISTS )
		{
			printf("이미 존재하는 event 객체를 Open 했습니다.₩n");
			printf("이미 응용 프로그램이 실행중입니다..₩n");
			return;
		}
		else
		{
			printf("event 객체를 새롭게 생성했습니다.₩n");
		}
	}
	getch(); // #include <conio.h> 추가해 주세요.
}
```

[예제 8-3] 활용 예제

아래 그림은 위 코드를 두 번 실행한 결과이다.

첫 번째 실행화면을 보면 “event 객체를 새롭게 생성했습니다” 라는 출력 결과를 볼 수 있다.
만약 첫 번째 프로그램을 실행한 상태에서 두 번째 프로그램을 실행하면 동일한 이벤트 객체를 생성하려고 접근했기 때문에 그에 따른 에러 메시지를 획득할 수 있고, 해당 프로그램을 강제 종료 시킬 수 있다.

이러한 기법을 통해 하나의 프로그램을 통해 두 개 이상의 프로세스가 생성 되는 것을 금지 할 수 있을 것이다.



8. 6 구조화된 예외 처리

8.6.1 예외의 정의

예외(Exception) 란 정상적인 코드 흐름의 외부 코드를 실행하는 일종의 이벤트이다. 예외는 하드웨어에 의해 발생할 수도 있고 소프트웨어에 의해 발생할 수도 있다. 운영체제는 예외가 발생했을 때 예외를 처리하는 핸들러를 호출하여 프로그램이 예외에 대처할 수 있도록 한다. 예외는 여러 가지 요인에 의해 발생하는데 예외를 근본적으로 막을 수 있는 방법은 없다.

예외 처리는 운영체제가 직접 지원하는 매커니즘이다. 하지만 예외가 제대로 동작하도록 코드를 생성하는 책임은 컴파일러에게 있다. 비록 운영체제가 예외 처리를 위한 준비를 오나벽하게 하고 있더라도 응용 프로그램이 이러한 예외 처리 서비스를 받을 수 있도록 작성되어 있지 않다면 완벽한 예외 처리는 불가능하다.

응용 프로그램이 예외 처리를 하도록 코드를 생성하는 것은 물론 컴파일러의 책임이다.

컴파일러는 운영체제가 에외 처리에 사용하는 정보를 응용 프로그램에 삽입해 두어야 하며 예외 발생시 운영체제가 호출할 수 있는 콜백함수를 작성해야 한다. 또한 예외 발생시 응용 프로그램이 비정상적으로 종료되지 않고 예외에 능동적으로 대처하도록 해야 할 책임도 컴파일러에게 있다.

그래서 예외 처리 루틴은 사용하는 컴파일러마다 다를 수 밖에 없으며 Win32 API의 예외에 대한 명확한 규정은 없다. 일반적으로 사용되는 예외 처리 루틴은 두 가지 형식이 있다. 첫 번째는 C형이며 두 번째는 C++ 형이다. 이 중 C형을 구조화된 예외 처리(Structured Exception Handing) 라고 하며 간단히 줄여 SEH라고 한다. SHE 구문도 컴파일러마다 다른데 여기서는 비주얼 C++을 기준으로 설명한다. C++형은 언어 차원에서 제공되는 예외 처리 방식이며 C++ 국제 표준에서 정의되어 있다.

8.6.2 구조화된 예외 처리

예외 핸들러

예외 핸들러는 총 세가지 요소로 구성된다. 문법은 아래와 같다.

```
__ try {

    //1) 예외 발생 경계 지역

}
__except(/*2)예외 처리*/) {

   //3) 예외 처리를 위한 코드 지역

}
```

__try 블록내에서 예외가 발생하면 __except 에 있는 예외 필터 방식에 따라 예외가 처리된다.

아래는 예외 처리 부에 들어가는 상수 값이다.

값	상수	설명
EXCEPTION_EXCUTE_HANDLER	1	예외 핸들러로 제어를 넘기고 예외 핸들러 다음 코드를 실행한다.

EXCEPTION_CONTINUE_SEARCH	2	예외 핸들러를 계속해서 찾는다.
EXCEPTION_CONTINUE_EXECUTION	-1	예외가 발생한 지점을 다시 실행한다.

예외 필터 값의 의미와 사용 예를 정리해 보자.

EXCEPTION_EXECUTE_HANDLER
예외 발생시 __except 블록을 실행한 뒤 __except 블록 이후 부분을 실행한다.
이 때 __try 블록의 예외 발생 이 후 코드는 무시된다.

예외가 발생하면 쓰레드 흐름이 정지되고 최악의 경우 종료로 이러진다. 그러므로 예외 발생시 그 예외 부분을 건너뛰어서 계속 흐름을 이어가게 해주는 메커니즘이다.

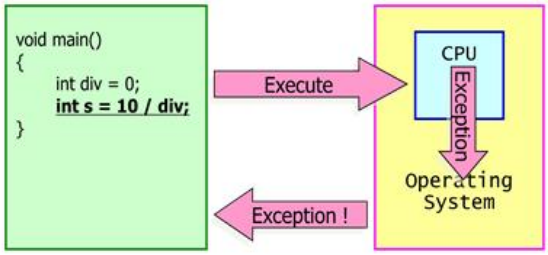
만약 __try 블록 내에 있는 어떤 함수를 호출하였을 경우 그 함수 내부에서 예외가 발생하면 예외로 처리될까?

당연히 예외 발생으로 인식하고 처리한다.
SHE 메커니즘은 함수 내에서 예외 발생시 __try __except 가 없을 경우 이 함수를 호출한 함수로 넘어간다. 그 함수에도 존재하지 않으면 계속 넘어간다.(스택 풀기)

EXCEPTION_CONTINUE_SEARCH
지금 실행 되고 있는 코드 이전의 try 블록이 위쪽 try 블록에 대응되는 except 문의 예외 필터를 호출한다.
이 값은 함수 호출 등에 의해 예외 처리가 중첩되어 있는 경우에만 사용할 수 있다.

예를 들어 보호 구역 내에 또 다른 보호 구역이 있는 경우 안쪽 보호 구역에서 이 값을 사용한다

EXCEPTION_CONTINUE_EXECUTION
예외가 발생한 문장을 다시 실행한다. 상식적으로 생각해 볼 때 예외가 발생한 지점으로 제어를 돌려 보내면 또 다시 예외가 발생할 것이다. 무한루프에 빠지지 않으려면 이동 이전에 조치를 취해서 예외 상황이 다시 발생되지 않도록 해야 할 것이다.




```
__try
{


}
__except( EXCEPTION_EXECUTE_HANDLER )
{

}
```

Hardware Exception : CPU에 의해 발생
Software Exception : OS와 Applicaton에 의해 발생

GetExceptionCode()
GetExceptionInformation()

EXCEPTION_EXECUTE_HANDLER(1)
EXCEPTION_CONTINUE_SEARCH(0)
EXCEPTION_CONTINUE_EXECUTION(-1)

위의 그림을 보면 main 함수에서 예외가 발생하면(0으로 나눈 경우) CPU에서는 예외를 감지하게 되며, Exception 에러를 발생시킨다.

만약 그림의 하단부처럼 오류 부분의 코드를 __try 문 안에서 작성했다면 Exception 에러 발생시 __except 문을 실행시키게 된다.

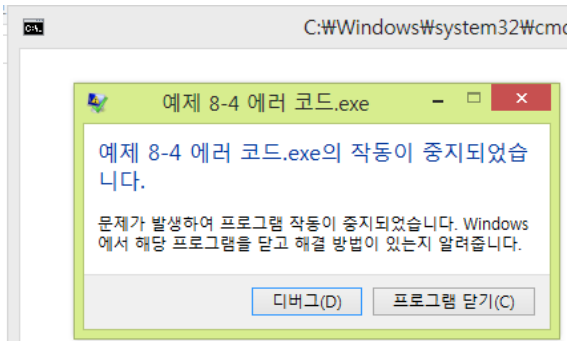
아래의 코드를 살펴보자.

```
#include <iostream>

using namespace std;

#include <stdlib.h>


int n = 0;

int main(void)

{

    int s = 10 / n;

     printf("결과 : %d₩n", s );

    return 0;

}
```

[예제 8-4] 에러 코드

위 코드는 0으로 나누는 에러를 가지고 있는 코드이며, 실제 해당 코드가 실행되면 아래와 같은 실행시 에러가 발생되면서 프로그램이 종료되게 된다.

보통의 경우 심각한 에러가 아닌 이상은 프로그램이 계속 실행 되기를 바랄 것이다. 예외 핸들러 기법을 이용하여 위의 문제를 해결 해 보고자 한다.



다음 예제는 위의 문제를 해결한다.

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

int n = 0;
DWORD ExceptionFilter( DWORD code  )
{
	if ( code == EXCEPTION_INT_DIVIDE_BY_ZERO ) // c0000094
	{
		return 1; // 처리할수 없는 예외 이므로 핸들러를 수행한다.
	}
}
void main()
{
	int s;
	__try
	{
		s = 10 / n;
```

```
                printf("결과 : %d₩n", s );
        }
        __except( ExceptionFilter( GetExceptionCode() ) )
        {
                printf("예외 발생 : %x₩n", GetExceptionCode() );
                ExitProcess( 0); // 예외가 발생한경우 대부분 프로세스를 종료 한다.
        }
        printf("프로그램 계속 실행₩n");
}
```

[예제 8-5] SHE 1

예외가 발생되면 ExceptionFilter API 함수가 호출되며 이 때 예외 코드가 인자로 전달된다. 전달된 인자가 미리 정의된 EXCEPTION_INT_DIVIDE_BY_ZERO 값과 동일하다면 return 1을 __except에 전달한다.

상수 1은 EXCEPTION_EXCUTE_HANDLER 의 의미를 가지며 따라서 예외 코드를 화면에 출력하며 프로세스를 종료하게 된다.

다음 두 번째 예제를 살펴보자.
프로그램 중 오류가 발생하면 EXCEPTION_CONTINUE_EXECUTION 플래그 값을 이용하여 오류를 처리하여 정상적으로 수행될 수 있는 흐름을 유도한다.

```
#include <windows.h>
#include <stdio.h>
int n = 0;
DWORD ExceptionFilter( DWORD code  ){
        if ( code == EXCEPTION_INT_DIVIDE_BY_ZERO ) // c0000094
        {
                printf("0으로 나누는 예외 발생. 새로운 값을 넣어 주세요 >> ");
                scanf( "%d", &n );
                return -; // 예외의 원인을 수정했으므로 예외난 곳을 다시 실행해 본다.
        }
        return 1; // 처리할수 없는 예외 이므로 핸들러를 수행한다.
}
```

```
void main(){
	int s;
	__try	{
		s = 10 / n;
		printf("결과 : %d₩n", s );
	}
	__except( ExceptionFilter( GetExceptionCode() ) )	{
		printf("예외 발생 : %x₩n", GetExceptionCode() );
		ExitProcess( 0); // 예외가 발생한경우 대부분 프로세스를 종료 한다.
	}
	printf("프로그램 계속 실행₩n");
}
```

```
[출력 결과]
0으로 나누는 예외 발생. 새로운 값을 넣어 주세요 >> 10
결과 : 1
프로그램 계속 실행
```

[예제 8-5] SHE 2

종료 핸들러

종료 핸들러(Structural Termination Handler)는 어떠한 상황에서도 특정 부분이 반드시 실행될 것을 보장하는 예외 처리 구조이다.

아래 예제를 참고하자.

```
#include <stdio.h>
#include <windows.h>
DWORD foo(){
	int ret = 0;
	__try	{
		return ret;
	}
	__finally {
		ret = 10;
```

```
        printf("try 블록을 벗어나기 전에 finally 구문은 반드시 실행됩니다₩n");
        }
        return 0;
}
void main(){
        printf("결과 : %d₩n", foo() ); // ?
}
```

[출력결과]

Try 블록을 벗어나기 전에 finally 구문은 반드시 실행됩니다.

결과 : 0

[예제 8-6] STH