12장. 쓰레드의 생성과 소멸

1. Windows에서의 쓰레드 생성과 소멸

쓰레드의 생성

HANDLE

CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

SIZE\_T dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

);

첫 번째 인자는 프로세스 생성 할 때도 본 것이다. 핸들의 상속 여부를 결정한다.

두 번째 인자는 쓰레드의 스택 크기를 지정하기 위한 매개변수이다. 0을 전달하면 디폴트 사이즈인 1M가 적용된다.

세 번째 인자는 쓰레드의 main 역할을 하는 함수를 지정하는 인자이다.

인자타입 LPTHREAD\_START\_ROUNTE 인데, 반환타입이 DWORD 이고 매개변수 타입은 LPVOID(void \*) 인 함수 포인터로 형변환 되어있다.

네 번째 인자는 쓰레드 함수에 전달할 인자를 지정하는 용도이다.

lpStartAddress 가 가리키는 함수 호출 시 전달할 인자를 지정하는 것이다.

Main 의 argv 생각하면 된다.

다섯 번 째 인자는 쓰레드의 생성 및 실행을 조절하기 위한 전달인자이다

CREATE\_SUSPENDED 가 전달되면, 생성과 동시에 Blocked 상태가 된다.

그러나 아래에서 나올 함수 ResumeThread 가 호출되면 실행한다.

XP 이상에서는 인자로 STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION 을 전달 할 수 있는데, 이 경우 dwStackSize 를 통해 전달되는 값의 크기는 reserve 메모리 크기를 의미하게 되고, 그렇지 않을 경우 commit 메모리 크기를 의미한다(이 내용은 나중에 설명한다. 그냥 넘어가자)

여섯 번 째 인자는 쓰레드 ID 를 전달받기 위한 변수의 주소값을 전달한다.

굳이 필요없다면 NULL 을 전달하면 되는데, ME 이하에서는 NULL을 전달할 수 없다.

이 함수가 실행되면, 쓰레드의 핸들이 반환된다.

그러면 Xp 에서 생성 가능한 최대 쓰레드의 개수는 몇 개 일까??

대답은 의외로 간단한데, 메모리가 허용하는 만큼 가능하다.

/\*

CountThread.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <tchar.h>

#define MAX\_THREADS (1024\*10)

DWORD WINAPI ThreadProc( LPVOID lpParam )

{

DWORD threadNum = (DWORD) lpParam;

while(1)

{

\_tprintf(\_T("thread num: %d \n"), threadNum);

Sleep(5000);

}

return 0;

}

DWORD cntOfThread = 0;

int \_tmain(int argc, TCHAR\* argv[])

{

DWORD dwThreadId[MAX\_THREADS];

HANDLE hThread[MAX\_THREADS];

while(1)

{

hThread[cntOfThread] =

CreateThread (

NULL,

0,

ThreadProc,

(LPVOID)cntOfThread,

0,

&dwThreadId[cntOfThread]

);

if (hThread[cntOfThread] == NULL)

{

\_tprintf(\_T("MAXIMUM THREAD SIZE: %d \n"), cntOfThread);

break;

}

cntOfThread++;

}

for(DWORD i=0; i<cntOfThread; i++)

{

CloseHandle(hThread[i]);

}

return 0;

}

위 소스에 대한 설명은 필요없다고 본다.

만약 예를 들어, 한 프로세스에서 두 가지의 쓰레드를 생성시켯다고 하자.

그러면 기존에 있던 메인 쓰레드까지 총 3개가 될 것이다.

여기서 메인 쓰레드와 생성된 쓰레드와의 return 문에는 다른 의미가 있다

메인은 프로세스 전체를 이끌기 때문에, 메인 쓰레드의 return 문은 프로세스 종료로 이어진다.

반면 실행 중 생성된 쓰레드의 return 문은 해당 쓰레드의 종료를 의미한다.

위 소스를 실행해 보면,

위 그림 같은 결과가 나온다 ( 컴퓨터 마다 다름)

상식적으로 생각하면, MAXIMUM THREAD SIZE: 이 문장이 제일 마지막으로 나와야 한다고 생각할 수도 있다.

계속 하다보면 언젠간 그렇게 나올 수도 있다.

이 것은 이 것을 의미한다.

CreateThread(…);

Printf(…);

이렇게 하고 실행하면, printf 함수가 먼저 호출될까? 아니면 쓰레드가 생성되어 쓰레드 함수가 먼저 호출될까?

답은 알 수 없다 이다.

위 출력 결과를 보면, 1446 개 까지의 쓰레드를 생성할 수 있다고 한다.

그렇다면, 스택 사이즈를 0 (디폴트 1M) 에서 10M 로 바꿔보자. (1024 \* 1024 \* 10)

이번에는 181이 나온다.

정확하지는 않지만, 대충 1446 / 10 = 146 개로 대충 맞는다.

그렇다면, 디폴트 사이즈의 반인 0.5M 로 하고 실행을 하겠다.

그런데 결과가 1451개로 나온다.

따라서, 이 결과에서 지금처럼 가당치도 않은 작은 메모리를 할당 하려하면, 무조건 1M 으로 할당하는 것을 할 수 있다.

Sleep 함수의 인자는, Millisecond 단위로 지정하고, 0을 인자로 전달하면 현재 자신에게 할당된 타임 슬라이스를 포기하고, 우선순위가 같은 다른 쓰레드에게 양보하는 것이다.

만약 우선순위가 같은 쓰레드가 없으면, 바로 실행을 재개한다.

쓰레드의 소멸 ( 쓰레드 생성에 대한 추가적인 이야기 포함 )

앞서 말했듯, 쓰레드의 소멸방법은 함수 내에서 return 문을 사용하는 것이 가장 이상적이다.

그렇다고 해서, 모든 경우에 최선의 방법이란 존재하지 않는다.

Return 이외에 방법도 얼마든지 있을 것이다.

쓰레드를 종료할 수 있는 추가적인 방법을 보자.

상황을 보자.

Case 1 : 쓰레드 종료 시 return 을 이용하면 좋은 경우(거의 대부분)

아주 간단한 예를 들어, 1부터 10까지 더한다고 하자.

그런데 이 일은 입,출력 작업이 아주 많이 요구되서 Blocked 상태에 자주 놓인다

(이건 어디까지나 가정이다. 그렇다고 보자)

따라서, 이 작업을 총 세개의 쓰레드에 나눠준다고 한다.

이 경우 정해진 시간 동안 CPU 가 보다 많은 일을 처리하고, Blocked 상태에 놓여도 세 개의 쓰레드가 나눠서 감당하기 때문에 속도가 높아진다.

그렇다면, 우선 쓰레드 함수를 먼저 디자인하자.

쓰레드가 할 일은 매우 단순하게 두 개의 숫자를 전달받으면 그 사잇값을 모두 더해주면 된다.

/\*

ThreadAdderOne.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <tchar.h>

DWORD WINAPI ThreadProc( LPVOID lpParam )

{

DWORD \* nPtr = (DWORD \*) lpParam;

DWORD numOne = \*nPtr;

DWORD numTwo = \*(nPtr+1);

DWORD total = 0;

for(DWORD i=numOne; i<=numTwo; i++)

{

total += i;

}

return total;

}

전달 인자는 DWORD arrParam[2] = {1,5}; 같은 배열이 전해져야 한다.

종료값은 앞서 배운 GetExitCodeProcess 함수를 통해 얻을 수 있다.

문제는 쓰레드의 종료코드를 어떻게 main 쓰레드가 확인하느냐 이다.

다음 예제는 위의 쓰레드 함수를 기반으로 1부터 10까지 덧셈을 하는 main 쓰레드 이다.

/\*

ThreadAdderOne.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <tchar.h>

DWORD WINAPI ThreadProc( LPVOID lpParam )

{

DWORD \* nPtr = (DWORD \*) lpParam;

DWORD numOne = \*nPtr;

DWORD numTwo = \*(nPtr+1);

DWORD total = 0;

for(DWORD i=numOne; i<=numTwo; i++)

{

total += i;

}

return total;

}

int \_tmain(int argc, TCHAR\* argv[])

{

DWORD dwThreadID[3];

HANDLE hThread[3];

DWORD paramThread[] = {1, 3, 4, 7, 8, 10};

DWORD total = 0;

DWORD result = 0;

hThread[0] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[0]),

0, &dwThreadID[0]

);

hThread[1] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[2]),

0, &dwThreadID[1]

);

hThread[2] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[4]),

0, &dwThreadID[2]

);

if(hThread[0] == NULL || hThread[1] == NULL || hThread[2] == NULL)

{

\_tprintf(\_T("Thread creation fault! \n"));

return -1;

}

WaitForMultipleObjects(3, hThread, TRUE, INFINITE);

GetExitCodeThread(hThread[0] , &result);

total += result;

GetExitCodeThread(hThread[1] , &result);

total += result;

GetExitCodeThread(hThread[2] , &result);

total += result;

\_tprintf(\_T("total (1 ~ 10): %d \n"), total);

CloseHandle(hThread[0]);

CloseHandle(hThread[1]);

CloseHandle(hThread[2]);

return 0;

}

GetExitCodeThread 의 인자는 쓰레드의 핸들, 종료코드를 저장할 메모리 주솟값 이다.

Case 2 : 쓰레드 종료 시 ExitThread 함수 호출이 유용한 경우 ( 특정 위치에서 쓰레드의 실행을 종료시키고자 하는 경우)

ExitThread 함수는 인자로 쓰레드 종료코드를 받고, 리턴값은 없다.

여기에 등록되는 종료코드는 GetExitCodeThread 를 통해 얻는다.

이 함수의 장점은 언제 어디서나 쓰레드를 종료 시킬 수 있고, return 문을 사용하는 것 보다 코드를 이해하기 좋다고 할 수 있다.

쓰레드 기반으로 프로그래밍을 한다 하더라도 쓰레드 함수 내에서 모든 것을 처리할 수는 없다.

만약, 쓰레드 함수에서 A 함수를, A 함수는 B 함수를, B 함수는 C 함수를 호출한다고 한다.

만약 C 함수에서 쓰레드를 종료한다고 하면, return 으로 종료하려면 다시 B , A 로 가야 하지만, ExitThread 함수를 호출하면 바로 종료가 된다.

하지만, C++ 로 할 경우, A,B 함수의 스택 프레임에 C++ 객체가 존재한다고 가정 할 경우, 그 객체의 소멸자는 호출되지 않는다. 따라서 메모리 누수가 날 수도 있다.

이로써, C,C++ 구분 없이 return 문이 가장 좋은 것 같다.

Case 3 : 쓰레드 종료 시 TerminateThread 함수 호출이 유용한 경우 ( 외부에서 쓰레드를 종료시키고자 하는 경우)

TerminateThread 는 핸들과 종료코드를 인자로 받는다.)

이 함수의 문제점은 강제 종료라는 점이다.

따라서, 대표적으로 메모리 혹은 할당받은 리소스 헤제 등을 처리하지 못하고 바로 종료된다.

이 함수는 안 쓰는 것을 추천한다.

2. 쓰레드의 성격과 특성

힙, 데이터 uddur, 그리고 코드 영역의 공유에 대한 검증

쓰레드는 메모리를 공유한다. 따라서 이전 예제 같이 복잡하게 할 필요가 없다.

이번 예제에서는 total 을 Data 영역에 전역변수로 선언을 하게 된다.

/\*

ThreadAdderTwo.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <tchar.h>

static int total = 0;

DWORD WINAPI ThreadProc( LPVOID lpParam )

{

DWORD \* nPtr = (DWORD \*) lpParam;

DWORD numOne = \*nPtr;

DWORD numTwo = \*(nPtr+1);

for(DWORD i=numOne; i<=numTwo; i++)

{

total += i;

}

return 0; // 정상 종료

}

int \_tmain(int argc, TCHAR\* argv[])

{

DWORD dwThreadID[3];

HANDLE hThread[3];

DWORD paramThread[] = {1, 3, 4, 7, 8, 10};

hThread[0] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[0]),

0, &dwThreadID[0]

);

hThread[1] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[2]),

0, &dwThreadID[1]

);

hThread[2] =

CreateThread (

NULL, 0,

ThreadProc,

(LPVOID)(&paramThread[4]),

0, &dwThreadID[2]

);

if(hThread[0] == NULL || hThread[1] == NULL || hThread[2] == NULL)

{

\_tprintf(\_T("Thread creation fault! \n"));

return -1;

}

WaitForMultipleObjects(3, hThread, TRUE, INFINITE);

\_tprintf(\_T("total (1 ~ 10): %d \n"), total);

CloseHandle(hThread[0]);

CloseHandle(hThread[1]);

CloseHandle(hThread[2]);

return 0;

}

위 예제보다 소스도 짧고, 이해하기도 좋다.

하지만, 위 예제는 한 가지 문제점이 있다.

동시 접근에 있어서의 문제점

위의 문제점은 total 이라는 전역변수에 둘 이상의 쓰레드가 동시접근을 한다는 것이다.

그냥 생각하기에는, 쓰레드는 돌아가면서 실행되니, 그럴 경우가 없다고 할 수도 있지만, 그래도 문제가 될 수 있다.

덧셈을 간략하게 세 단계로 하면, 레지스트로 데이터 이동 -> ALU 에 의해 덧셈 -> 결과값을 메모리에 저장 을 하게 된다.

현재 total 의 값을 10이고, 두 개의 쓰레드가 각각 6 , 9를 더하려는 상황이라 보자.

A 쓰레드의 값을 6 , B 쓰레드의 값을 9 라고 하자.

CCPU 는 현재 A를 실행중이다. 따라서 total 의 값을 레지스터로 이동하고 6의 덧셈도 끝 난 상태이다.

그리고 현재 연산결과는 레지스터에 저장되어 있다.

이제 남은 일은 이 값을 메모리에 저장하는 것이다. 그런데, 이 찰나에 스케줄러에 의해 실행의 대상이 B 가 되었다.

따라서 현재의 상황을 메모모리에 저장을 하고, 쓰레드 B 는 연산을 시작한다.

그럼 total 값은 19가 되고, B 가 종료 되었으므로 다시 쓰레드 A 가 기회를 얻는다.

따라서, 기존의 저장된 데이터를 복원하게 되는데, 이 때, 값은 16이 된다.

이것은 무엇을 의미할까?

우리는 보통 printf 함수, scanf 함수가 실행 중간에는 절대로 컨텍스트 스위칭이 안 일어난다고 생각하는 것이다.

하지만, 실상은 값을 1 증가 , 감수 시키는 ++ , -- 을 실행시키는 과정에서도 컨텍스트 스위칭은 빈번히 발생한다.

때문에, 둘 이상의 쓰레드가 같은 메모리 영역을 동시 참조하는 것은 문제가 있다.

프로세스로부터의 쓰레드 분리

이 부분은 자식 프로세스 를 공부한 6장에서와의 같은 내용이다.

프로세스에서 일어나는 문제 ( Usage Count 등 ) 은 쓰레드에서도 똑같이 일어난다.

이 부분은 6장을 다시 보기 바란다.

ANSI 표준 C 라이브러리와 쓰레드

우리가 C 언어를 배울 때 가장 처음 접하는 것이 printf 라는 표준 C 라이브러리 함수다.

뿐만 아니라, 특히 문자열 처리와 입,출력에 대해서는 표준 C 라이브러리를 많이 활용한다.

문제는, 표준 C 라이브러리가 구현 될 당시만 해도 쓰레드에 대한 고려가 전혀 이뤄지지 않았다는 점이다.

따라서, 멀티 쓰레드 기반으로 구현하면 동일한 메모리 영역을 동시 접근하는 불상사가 생긴다.

대표적 예가 문자열을 특정 기준에 따라 나누는 strtok 함수이다.

/\*

strtok.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <tchar.h>

int \_tmain(int argc, TCHAR\* argv[])

{

TCHAR string[] =

\_T("Hey, get a life!")

\_T("You don't even have two pennies to rub together."); // 분리 할 문자열

TCHAR seps[] = \_T(" ,.!"); // 공백 ‘,’ ‘\n’을 분리의 대상으로 선택.

// 토큰 분리 조건, 문자열 설정 및 첫 번째 토큰 반환

TCHAR \* token = \_tcstok( string, seps );

// 계속해서 토큰을 반환 및 출력

while( token != NULL )

{

\_tprintf( \_T(" %s\n"), token );

token = \_tcstok( NULL, seps );

}

return 0;

}

Strtok 함수를 호출되면서 처음에 등록된 문자열은 어딘가에 저장되어야만 한다.

그래서 두 번째 부터는 NULL 을 인자로 줘서 출력할 것이다.

우리는 전역, 혹은 static 으로 선언된 배열에 문자열이 저장되어 있음을 예측할 수 있다.

이 경우, 메모리의 동시 참조의 문제가 발생할 수 있다.

해결책은 MS 에서 멀티 쓰레드에 안전한 ANSI 표준 라이브러리를 제공하고 있다.

따라서, 프로젝트 설정 – C/C++ / Code Generation 에서 런타임 라이브러리를 Multi-hreaded Debug DLL 로 바꿔주면 된다.

이제 한 가지 일을 더 해줘야 한다.쓰레드를 생성 할 때,CreateThread 함수 대신 \_beginthreadex 함수를 사용하면 된다..

이 함수는 내부적으로 CreateThread 를 호출하지만, 그 전에 독립적 메모리 블록을 할당한다.

Multi- 로 시작하는 이름의 표준 C 라이브러리 함수는 이렇게 할당된 쓰레드 각각의 메모리 블록을 기반으로 연산한다.

이로써 멀티 쓰레드 기반에서 안정성이 확보되는 것이다.

\_beginthreadex 함수는 전달인자의 순서와 의미가 CreateThread 함수와 동일하다.

다만 선언된 매개변수 자료형과 반환형에 차이가 있기 때문에, 약간 형 변환이 요구된다.

/\*

CountThreadMultiThread.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <process.h>

#include <tchar.h>

#define MAX\_THREADS (1024\*10)

unsigned int WINAPI ThreadProc( LPVOID lpParam )

{

DWORD threadNum = (DWORD) lpParam;

while(1)

{

\_tprintf(\_T("thread num: %d \n"), threadNum);

Sleep(1000);

}

return 0;

}

DWORD cntOfThread = 0;

int \_tmain(int argc, TCHAR\* argv[])

{

DWORD dwThreadId[MAX\_THREADS];

HANDLE hThread[MAX\_THREADS];

// 생성 가능한 최대 개수의 쓰레드 생성

while(1)

{

hThread[cntOfThread] = (HANDLE)

\_beginthreadex (

NULL, // 디폴트 보안 관리자

0, // 디폴트 스택 사이즈

ThreadProc, // 쓰레드 amin 함수 설정

(LPVOID)cntOfThread, // 쓰레드 함수의 전달 인자

0, // 디폴트 쓰레드 생성 속성

(unsigned \*)&dwThreadId[cntOfThread] // 쓰레드 ID 저장을 위한 주소값 전달

);

// 쓰레드 생성 확인

if (hThread[cntOfThread] == NULL)

{

\_tprintf(\_T("MAXIMUM THREAD SIZE: %d \n"), cntOfThread);

break;

}

cntOfThread++;

}

for(DWORD i=0; i<cntOfThread; i++)

{

CloseHandle(hThread[i]);

}

return 0;

}

실행 결과는 차이가 없다.

앞으로는 멀티 쓰레드 기반 프로그래밍을 한다는 가정하에 \_beginthreadex 함수를 사용한다.

한가지 주의할 점은 쓰레드를 종료하는 방법이다.

만약 ExitThread 함수를 활용하고자 하면, \_endthreadex 를 사용하기 바란다.

인자는 동일하다.

이 함수는 \_beginthreadex 함수에서 메모리를 할당했으므로, 종료할 때는 그 메모리를 반환해야 한다. 그래서 사용하는 함수이다.

그럼 return 문을 이용하면 메모리 반환이 안될까?

된다! 쓰레드 함수에서 return 문을 이용할 경우, \_endthreadex 함수가 자동 호출된다.

그냥 return 에 의한 종료가 만사Ok 이다!.

참고,

\_beginthread , \_endthread 라는 이름의 함수가 있는데, 이 것들은 쓰레드 생성을 간결하게 하려고 만든 함수인데, 기능도 제한적이고 핸들도 사용 할 수 없다.

이러한 문제점들 때문에 많은 전문가들이 가급적이면 사용하지 말라고 하는 함수이다.

3. 쓰레드의 상태 컨트롤

쓰레드의 상태는 프로그램이 실행되는 과정에서 수도없이 변경된다.

이 것은 상황에 따라 운영체제의 관리방법에 따른 것이므로, 프로그래머가 건드리는 것은 아니다.

그러나, 경우에 따라서 프로그래머가 직접 변경해 주어야 하는 상황도 있다.

특정 쓰레드를 지목하면서, 그 쓰레드의 실행을 잠시 멈추기 위해 Blocked 상태로 만들거나, 다시 실행시키기 위해 Ready 상태로 두기 위해서 필요하다.

쓰레드의 상태 변화

앞서 프로세스의 상태 변화에도 설명했는데, Windows 에서는 상태 변화의 주체가 프로세스가 아니라 쓰레드이다.

하지만, 내용은 별 달라지 것이 없고, 프로세스와 관련해 이해하는 것을 그대로 쓰레드로 이해하면 된다.

따라서 이 내용은 생략하겠다. 프로세스의 상태와 거의 100% 동일하다.

Suspend & Resume

이제 쓰레드의 상태를 컨트롤 하자.

DWORD suspendThread(

HANDLE hThread

);

이 함수는 blocked 상태로 만드는 함수이고, 핸들을 인자로 받는다.

DWORD ResumeThread(

HANDLE hThread

);

이 함수는 Ready 상태로 만드는 함수이고, 핸들을 인자로 받는다.

쓰레드의 커널 오브젝트에는 SuspendThread 함수의 호출 빈도수를 기록하기 위한 서스펜드 카운트라는 멤버가 있다.

현재 실행중인 쓰레드의 카운트는 0이고, SuspendThread 함수가 호출 될 때 마다 카운트는 증가한다.

그런데 ResumdThread 는 엄밀히 말하면 Ready 상태로 두는 것이 아니라, 이 카운트를 하나 감소하는 것이다.

따라서 카운트가 2이면, ResumeThread 함수가 두 번 호출되어야 한다.

함수의 반환값을 통해 서스펜드 카운트를 확인할 수 있다.

예제를 보자.

/\*

SuspendCount.cpp

\*/

#include <stdio.h>

#include <windows.h>

#include <process.h>

#include <tchar.h>

unsigned int WINAPI ThreadProc( LPVOID lpParam )

{

while(1)

{

\_tprintf( \_T("Running State! \n") );

Sleep(10000);

}

return 0;

}

int \_tmain(int argc, TCHAR\* argv[])

{

DWORD dwThreadId;

HANDLE hThread;

DWORD susCnt;

hThread = (HANDLE)

\_beginthreadex (

NULL,

0,

ThreadProc,

NULL,

CREATE\_SUSPENDED ,

(unsigned \*)&dwThreadId

);

if (hThread == NULL)

\_tprintf(\_T("Thread creation fault! \n") );

susCnt = ResumeThread(hThread);

\_tprintf(\_T("suspend count: %d \n"), susCnt);

Sleep(10000);

susCnt = SuspendThread(hThread);

\_tprintf(\_T("suspend count: %d \n"), susCnt);

Sleep(10000);

susCnt = SuspendThread(hThread);

\_tprintf(\_T("suspend count: %d \n"), susCnt);

Sleep(10000);

susCnt = ResumeThread(hThread);

\_tprintf(\_T("suspend count: %d \n"), susCnt);

susCnt = ResumeThread(hThread);

\_tprintf(\_T("suspend count: %d \n"), susCnt);

WaitForSingleObject(hThread, INFINITE);

return 0;

}

이 예제는 설명이 필요 없을 것이다.

Running State! 문자열 출력 횟수는 시스템마다 다를 수 있지만, 출력되는 서스펜드 카운트는 어떤 시스템이던 완전히 동일하게 출력된다.

4. 쓰레드의 우선순위 컨트롤

Windows 에서는 프로세스가 우선순위를 갖는 것이 아니라, 프로세스 안의 쓰레드가 우선순위를 가진다.

9장에서 말한 프로세스의 우선순위를 가리켜 기준 우선 순위라고 한다.

(IDLE\_PRIORITY\_CLASS , NORMAL\_PRIORITY\_CLASS 등..)

쓰레드는 상대적인 우선순위를 갖는다.

#define THREAD\_PRIORITY\_LOWEST -2

#define THREAD\_PRIORITY\_BELOW\_NORMAL -1

#define THREAD\_PRIORITY\_NORMAL 0

#define THREAD\_PRIORITY\_ABOVE\_NORMAL 1

#define THREAD\_PRIORITY\_HIGHEST 2

쓰레드의 우선순위는 프로세스의 기준 우선순위와 쓰레드의 상대 우선순위를 조합해서 결정된다.

참고로 위 표의 우선순위 상수값은 Windows 버전별 차이가 있다..

따라서 우선순위가 정확히 몇 이라고 이해하기 보다는 어느것 보다는 높겠다 정도로만 이해하면 된다.

프로세스 내 모든 쓰레드의 상대 우선순위는 THREAD\_PRIORITY\_NORMAL 이다.

이를 변경 , 참조할 때는 다음 두 함수를 쓴다.

BOOL SetThreadPriority(HANDLE hThread, int nPriority);

Int GetThreadPriority(HANDLE hThread);

이 함수에 대해서는 설명하지 않아도 알 것이다.