|  |
| --- |
| * **CreateThread** |
| 윈도우 운영체제에서 새로운 스레드를 생성하려면 아래의 함수를 사용한다. |
| **HANDLE** **CreateThread( LPSECURITY\_ATTRIBUTES** *lpThreadAttributes***,**  **SIZE\_T** *dwStackSize***,**  **LPTHREAD\_START\_ROUTINE** *lpStartAddress***,**  **LPVOID** *lpParameter***,**  **DWORD** *dwCreationFlags***,**  **LPDWORD** *lpThreadId* **);** |
| 각 인자에 대한 설명은 다음과 같다. |
| [in] ***lpThreadAttributes*** |
| 새롭게 생성되는 스레드 커널 객체의 보안속성(보안 설명자, 상속여부)을 지정한다. |
| [in] ***dwStackSize*** |
| 새롭게 생성되는 스레드를 위한 스택의 크기를 지정한다. 0 을 지정할 경우에는 디폴트 크기가 지정된다. |
| [in] ***lpStartAddress*** |
| 새로운 스레드가 수행할 함수의 주소(이름)을 지정한다. 스레드 함수의 모양은 아래와 같아야 한다. |
| **DWORD WINAPI ThreadProc( LPVOID lpParameter );** |
| [in] ***lpParameter*** |
| 스레드 함수의 인자로 전달한 32bit의 데이터. 주로 포인터를 사용해서 전달한다. |
| [in] ***dwCreationFlags*** |
| 스레드 생성을 제어할 플래그를 지정한다. 아래와 같은 값들을 전달 할 수 있다. |
| **CREATE\_SUSPENDED** : 스레드를 Suspend(중지)상태로 생성한다. ResumeThread를 호출 하기 전 까지는 실행되지 않는다.  **STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION** : XP부터 사용가능.2번째 인자로 전달한 스택의 크기만큼을 예약하게 한다. 만일 이 값을 지정하지 않는다면 2 번째 인자로 전달한 스택에 크기 만큼 확정한다. |
| [out] ***lpThreadId*** |
| 생성된 스레드의 ID를 담아올 변수의 주소를 전달한다. 윈도우 2000이후의 운영체제에서는 ID값이 필요 없다면 이 인자로 NULL을 전달할 수도 있다. |
| * **참고 - CreateRemoteThread (지금을 사용하진 않지만 언젠간 사용할끼다.)** |
| 윈도우 NT/2000/XP에는 다른 프로세스의 주소 공간에 스레드를 생성 할 수 있는 함수도 제공한다.(95/98 은 안됨) 아래의 함수가 제공된다. |
| HANDLE CreateRemoteThread( HANDLE hProcess,  LPSECURITY\_ATTRIBUTES lpThreadAttributes,  SIZE\_T dwStackSize,  LPTHREAD\_START\_ROUTINE lpStartAddress,  LPVOID lpParameter,  DWORD dwCreationFlags,  LPDWORD lpThreadId); |
| 첫 번째 인자로 스레드를 생성할 프로세스의 핸들을 전달한다. 나머지 파라미터는 CreateThread와 동일하다. 단, 4번째 인자와 5번째 인자인 함수와 전달인자는 반드시 스레드를 생성할 프로세스의 주소공간에 있는 값이 어야 한다. |

|  |
| --- |
| 각 스레드 연기 카운트(Suspend Count)를 가지고 있다. Suspend Count 가 1이상이면 스레드는 실행을 중지 하고 대기 상태에 있게 된다. |
| 스레드의 Suspend Count를 늘리거나 줄이려면 아래 함수 들을 사용한다. |
| **DWORD SuspendThread( HANDLE** *hThread* **);** - Suspend Count 를 1 증가한다.( 스레드는 중지 상태가 된다.)  **DWORD** **ResumeThread(** **HANDLE** *hThread* **);** - Suspend Count 를 1 감소한다.( 0 이 되면 스레드는 재개 상태가 된다.) |
| Suspend Count 는 음수가 될 수 없으며(항상 0이상) 최대 MAXIMUM\_SUSPEND\_COUNT 까지 가능하다. |
| HANDLE hThread = CreateThread( 0, 0, ThreadFunc, &thread\_data, CREATE\_SUSPENDED, 0);  // other thread initialize  ResumeThread( hThread ); // now thread run |
| CreateThread()의 5번째 인자로 CREATE\_SUSPENDED를 지정하면 스레드가 실행을 중지 한 상태로 만들어 진다.( Suspend Count가 1 인 상태) |
| * **Sleep()** |
| 스레드는 스스로 잠깐 동안 실행을 중시 시킬 수 있는데, 아래 함수를 사용한다. |
| **VOID** **Sleep(** **DWORD** *dwMilliseconds* **);** |
| *dwMilliseconds* 인자로 전달된 시간 만큼 현재 스레드의 실행을 중지 시킨다. |

|  |  |  |
| --- | --- | --- |
| **Thread Priority(스레드 우선순위)** | | |
| * **BOOL** **SetPriorityClass(** **HANDLE** *hProcess***, DWORD** *dwPriorityClass* **);** * **DWORD** **GetPriorityClass(** **HANDLE** *hProcess* **);** * **BOOL** **SetThreadPriority(** **HANDLE** *hThread***, int** *nPriority* **);** * **int** **GetThreadPriority(** **HANDLE** *hThread* **);** | | |
|  | **Process Priority Class** | **Thread Priority Level** |
| 1 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 1 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 1 | NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 1 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 1 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 2 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 3 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 4 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 4 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 5 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 5 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 5 | Background NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 6 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 6 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 6 | Background NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 7 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 7 | Background NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 7 | Foreground NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 8 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 8 | NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 8 | Foreground NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 8 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 9 | NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 9 | Foreground NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 9 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 10 | Foreground NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 10 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 11 | Foreground NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 11 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 11 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 12 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 12 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 13 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 14 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 15 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 15 | HIGH\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |
| 15 | IDLE\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |
| 15 | BELOW\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |
| 15 | NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |
| 15 | ABOVE\_NORMAL\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |
| 16 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_IDLE |
| 17 | REALTIME\_PRIORITY\_CLASS | -7 |
| 18 | REALTIME\_PRIORITY\_CLASS | -6 |
| 19 | REALTIME\_PRIORITY\_CLASS | -5 |
| 20 | REALTIME\_PRIORITY\_CLASS | -4 |
| 21 | REALTIME\_PRIORITY\_CLASS | -3 |
| 22 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_LOWEST |
| 23 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_BELOW\_NORMAL |
| 24 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_NORMAL |
| 25 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_ABOVE\_NORMAL |
| 26 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_HIGHEST |
| 27 | REALTIME\_PRIORITY\_CLASS | 3 |
| 28 | REALTIME\_PRIORITY\_CLASS | 4 |
| 29 | REALTIME\_PRIORITY\_CLASS | 5 |
| 30 | REALTIME\_PRIORITY\_CLASS | 6 |
| 31 | REALTIME\_PRIORITY\_CLASS | THREAD\_PRIORITY\_TIME\_CRITICAL |

|  |  |
| --- | --- |
| **Thread Syncronization (Thread 동기화 )** | |
|  | 아래 코드를 보자 |
|  | #include <windows.h>  #include <stdio.h>  void Delay() { for (int i = 0; i < 10000000; ++i ) ; }  DWORD WINAPI ThreadFunc(void\* p) {  char\* name = (char\*)p;  static int x = 0;  for ( int i = 0; i < 20; ++i ) {  x = 100; Delay();  x = x+1; Delay();  printf("%s : %d\n", name, x); Delay();  }  printf("%s Finish...\n", name);  return 0;  }  void main() {  HANDLE hThread[2];  hThread[0] = CreateThread( 0, 0, ThreadFunc, "A", 0, 0);  hThread[1] = CreateThread( 0, 0, ThreadFunc, "\tB", 0, 0);  WaitForMultipleObjects( 2, hThread, TRUE, INFINITE);  CloseHandle( hThread[0] );  CloseHandle( hThread[1] );  } |
|  |  |
| **다른 스레드의 종료를 대기하는 법** | |
|  | 스레드가 종료될 때는 스레드 Kernel Object 가 Signal 된다. 아래 2개의 함수를 사용해서 특정 KO가 Signal 될때를 대기 할 수 있다. |
|  | **DWORD** **WaitForSingleObject(** **HANDLE** *hHandle***, DWORD** *dwMilliseconds* **);** |
|  | **DWORD** **WaitForMultipleObjects(** **DWORD** *nCount***, const HANDLE\*** *lpHandles***,**  **BOOL** *bWaitAll***, DWORD** *dwMilliseconds* **);** |
|  |  |
| **전역변수와 지역변수** | |
|  | 지역 변수 : 스택에 놓인다. 스레드당 1개씩 만들어진다. 멀티 스레드 환경에서 안전하다. |
|  | 전역 변수 : static data 공간에 놓인다. 프로세스당 1개, 즉 동일 프로세스 내의 모든 스레드가 공유한다. 여러 스레드에서 접근할 경우 반드시 동기화가 필요하다. |

|  |
| --- |
| Critical Section은 1개의 스레드 만이 들어 갈수 있는 공간이다. CRITICAL\_SECTION이라는 type의 변수를 전역적으로 만들고 아래의 4가지 함수를 사용한다. |
| **void InitializeCriticalSection( LPCRITICAL\_SECTION** *lpCriticalSection* **);**  **void** **EnterCriticalSection(** **LPCRITICAL\_SECTION** *lpCriticalSection* **);**  **void** **LeaveCriticalSection(** **LPCRITICAL\_SECTION** *lpCriticalSection* **);**  **void** **DeleteCriticalSection(** **LPCRITICAL\_SECTION** *lpCriticalSection* **);** |

|  |  |  |
| --- | --- | --- |
| **Atomic Operation(원자 연산)** | | |
|  | 다음 코드를 살펴보자. | |
|  | #include <stdio.h>  #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 = %d\n", value);**  } | |
|  | 1. 스레드 함수인 ThreadFunc 는 전역변수 value 를 천만번 증가(++value) 시킨다. 2. Main 함수에서는 5개의 스레드를 생성해서 ThreadFunc 함수를 수행한다. 3. 주 스레드는 모든 스레드가 종료된 후 value의 값을 출력 하고 있다. | |
|  | Value 값은 얼마가 될까 ? | |
|  |  | |
|  | 왜 이런 결과가 나왔을까 ?  **Atomic Operation Function(원자 연산 함수) reference** | |
| * **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* );** | |

|  |  |
| --- | --- |
| **Thread Local Storage** | |
|  | 아래 코드를 살펴보자 |
|  | #include <stdio.h>  #include <windows.h>  int NextOddNumber()  {  static int n = -1;  n = n+2;  return n;  }  void main()  {  printf("%d\n", NextOddNumber() );  printf("%d\n", NextOddNumber() );  printf("%d\n", NextOddNumber() );  } |
|  | NextOddNumber() 함수는 1부터 홀수를 차례대로 리턴하는 함수이다. |
|  | 이 함수는 멀티 스레드 환경에서 안전한가 ? |
|  |  |
| **Static TLS** | |
|  | int NextOddNumber()  {  **\_\_declspec(thread) static int n = -1;**  n = n+2;  return n;  } |
|  |  |

|  |
| --- |
| 한정된 자원을 n개의 스레드 에서 만 접근하려고 할 때 세마포어를 사용한다. 세마포어를 사용하면 n개의 스레드가 자원을 사용 할 수 있게 한다. |
|  |
| 일반적인 세마포어의 규칙은 다음과 같다. |
| * COUNT 는 0 보다 같거나 크고 MAX COUNT 보다 작거나 같다. * COUNT 가 0 보다 크면 Signal 되고 0 이면 Non-signal 된다. * WaitForSingleObject()를 통과하면 COUNT가 1 감소한다. * ReleaseSemaphore()함수를 사용하면 COUNT를 n만큼 증가할 수 있다.(단, MAX COUNT보다 커지지는 않는다.) |
|  |
| 아래의 함수를 사용해서 세마포어를 생성한다. |
| **HANDLE** **CreateSemaphore( LPSECURITY\_ATTRIBUTES** *lpSemaphoreAttributes***,**  **LONG** *lInitialCount***,**  **LONG** *lMaximumCount***,**  **LPCTSTR** *lpName* **);** |
| 또한, 아래 함수를 사용해서 세마포어의 COUNT를 증가한다. |
| **BOOL** **ReleaseSemaphore( HANDLE** *hSemaphore***,**  **LONG** *lReleaseCount***,**  **LPLONG** *lpPreviousCount* **);** |