13장. 스레드 동기화 기법 1

**두 가지 관점에서의 스레드 동기화**

- 실행 순서 동기화

스레드의 실행 순서를 정의하고, 이 순서에 반드시 따르도록 하는 것이 스레드 동기화 이다.

- 메모리 접근에 대한 동기화

메모리 접근에 있어서 동시 접근을 막는 것 또한 스레드의 동기화에 해당한다.

**스레드 동기화에 있어서의 두 가지 방법**

Windows에서 제공하는 동기화 기법은 제공하는 주체에 따라 크게 두 가지로 나뉜다.

- 유저 모드 동기화

동기화가 진행되는 과정에서 커널의 힘을 빌리지 않는 동기화 기법이다. 다라서 동기화를 위해서 커널 모드로의 전환이 불필요하기 때문에 성능상의 이점이 존재, 대신 그만큼 기능의 제한이 따른

- 커널 모드 동기화

동기화 관련 함수가 호출될 때 커널 모드로의 변경이 필요하고, 성능의 저하로 이어지게 된다. 하지만 그만큼 유저 모드 동기화에서 제공하지 못하는 기능을 제공 받을 수 있다.

2. 임계 영역(Critical Secsion) 접근 동기화

“메모리 접근 동기화” 에 대해 이해

**임계 영역에 대한 이해**

[예제 13-1 CriticalSecsion.cpp]

위 예제의 전역 변수 gTotalCount와 같이 둘 이상의 스레드가 동시에 실행할 경우 문제가 생길 수 있는 코드 블록을 가리켜 **임계 영역(Critical Section)** 이라 한다.

즉, 문제의 원인이 될 수 있는 코드 블록을 가리켜 임계 영역 이라 한다. 할당된 메모리 공간을 가리켜 임계 영역이라 하는 것이 아니다.

요약하면, **임계 영역(Critical Section)이란 배타적 접근(한 순간에 하나의 스레드만 접근)이 요구되는 공유 리소스(전역 변수 같은)에 접근하는 코드 블록을 의미 한다.**

위 문제의 해결 책은 임계 영역의 동시접근은 막는 것이다. 즉 동기화 기법을 통해서 임계 영역을 한 순간에 하나의 스레드만 실행 될 수 있도록 제한하면 된다.

|  |  |
| --- | --- |
| 유저 모드 동기화 | 크리티컬 섹션 기반의 동기화 : 메모리 접근 동기화에 사용할 예정 |
| 인터락 함수 기반의 동기화 : 메모리 접근 동기화에 사용할 예정 |
| 커널 모드 동기화 | 뮤텍스 기반의 동기화 : 메모리 접근 동기화에 사용할 예정 |
| 세마포어 기반의 동기화 : 메모리 접근 동기화에 사용할 예정 |
| 이름 있는 뮤텍스 기반의 프로세스 동기화 : 프로세스간 동기화에 사용할 예정 |
| 이벤트 기반의 동기화 : 실행 순서 동기화에 사용할 예정 |

3. 유저 모드의 동기화

**크리티컬 섹션(Critical Section) 기반의 동기화**

대부분의 동기화 기법은 화장실 앞에 걸어 놓은 화장실 열쇠에 비교하면 이해가 수월 하다.

임계 영역을 화장실에 비유하자. 이 화장실에 들어가기 위해서는 화장실 앞에 걸려 있는 열쇠를 가져야만 한다. 열쇠가 걸려 있다면 이 열쇠로 문을 열고 화장실에 들어가면 된다. 일을 다 보고 난 후에는 다시 열쇠를 화장실 앞에 걸어 놓는다. 그래야 다음 사람이 화장실에 들어갈 수 있다. 이것이 바로 크리티컬 섹션의 동기화 방식이다.

크리티컬 섹션의 동기화를 사용하려면 크리티컬 섹션 오브젝트라는 것을 만들어 초기화해야 한다. 크리티컬 센션 오브젝트는 자료형 CRITICAL\_SECTION 변수를 말한다.

|  |
| --- |
| CRITICAL\_SECTION gCriticalSection; |

생성된 크리티컬 섹션 오브젝트를 가지고 다음의 초기화 과정을 거쳐야 한다. 초기화 과정을 통해서 크리티컬 섹션 오브젝트는 사용 가능한 상태가 된다. (화장실 앞에 열쇠를 걸어 놓는 행위)

|  |
| --- |
| void InitializeCriticalSection {  LPCIRITICAL\_SECTION lpCriticalSection;  } |

- lpCriticalSection : 초기화하고자 하는 크리티컬 섹션 오브젝트의 주소값을 인자로 전달

열쇠를 걸어 놓았으니 화장실에 들어가기 위해서는 열쇠를 사용해야 한다.

열쇠의 사용 행위는 두가지가 있다. 하나는 화장실에 들어가기 위해서 열쇠를 획득하는 행위이고, 하나는 화장실에서 나온후 열쇠를 제자리에 걸어두는 행위이다. 다음은 두가지 역할을 하는 함수들이다.

|  |
| --- |
| void EnterCriticalSection {  LPCIRITICAL\_SECTION lpCriticalSection;  } |

-lpCriticalSection : 임계영역에 진입하기 위해 필요한 크리티컬 섹션 오브젝트의 주소값을 인자로 전달, 만약 누군가에 의해서 이미 이 함수가 호출된 상태라면, 호출된 함수는 블로킹 되며, 크리티컬 섹션 오브젝트가 반환되면, 블로킹 상태를 빠져 나오게 된다.

|  |
| --- |
| void LeaveCriticalSection {  LPCIRITICAL\_SECTION lpCriticalSection;  } |

-lpCriticalSection : 임계 영역을 빠져 나오고 나서 호출하는 함수이며 반환 시 크리티컬 섹션 오브젝트를 반환하여 크리티컬 섹션 오브젝트 획득을 위해 블로킹되어 있는 다음 스레드가 오브젝트를 획득하여 작업을 수행할 수 있게한다.

|  |
| --- |
| // 임계 영역 진입을 위해 크리티컬 섹션 오브젝트 획득  EnterCriticalSection (&CriticalSection);  임계 영역  // 크리티컬 섹션 오브젝트 반환  LeaveCriticalSection(&CriticalSection); |

끝으로 초기화 함수가 호출되는 과정에서 할당된 리소스들을 해제하는 함수

|  |
| --- |
| void DeleteCriticalSection {  LPCRITICAL\_SECTION lpCriticalSection;  } |

- lpCriticalSection : 리소스 해제를 원하는 크리티컬 섹션 오브젝트의 주소 값 전달.

[예제 13-2 CriticalSectionSync.cpp]

**인터락 함수(Interlocked Family Of Function) 기반의 동기화**

변수 하나의 접근 방식을 동기화 하는 것이 목적이라면, 인터락 함수를 사용하는 것도 나쁘지 않다. 인터락 함수는 함수 내부적으로 한 순간에 하나의 스레드에 의해서만 실행되도록 동기화 되어있다.

|  |
| --- |
| LONG InterlockedIncrement (  LONG volatile \*Addend  ) |

Addend : 값을 하나 증가시킬 32비트 변수의 주소값을 전달. 둘 이상의 스레드가 공유하는 메모리에 저장된 값을 이 함수를 통해서 증가 시킬 경우, 동기화된 상태에서 접근하는 것과 동일한 안정성을 보장받을 수 있다.

|  |
| --- |
| LONG InterlockedDecrement (  LONG volatile \*Addend  ) |

Addend : 값을 하나 감소시킬 32비트 변수의 주소값을 전달. 둘 이상의 스레드가 공유하는 메모리에 저장된 값을 이 함수를 통해서 증가 시킬 경우, 동기화된 상태에서 접근하는 것과 동일한 안정성을 보장받을 수 있다.

인터락 함수는 **원자적 접근(Atomic Access),** 즉 한순에 하나의 스레드만 접근하는 것을 보장해 주는 함수이다. 따라서 멀티 스레드 환경에서 문제가 되었던 임계영역 문제가 발생하지 않는다. 또한, 인터락 함수들도 유저 모드 기반으로 동작하기 때문에 속도가 상당히 빠르다. 이 함수를 활용할 경우. 보다 간결하게 스레드에 안전한 형태가 될 수 있다.

위 두 함수 뿐만 아니라, MS에서 제공하는 다양한 인터락 함수들이 존재한다.

**Volatile 키워드**

Volatile 키워는 C,C++ 언어의 ANSI 표준 키워드이며, 크게 두 가지 의미를 지닌다.

1. 최적화를 수행하지 마라

컴파일러는 프로그래머가 프로그래밍해 놓은 코드를 컴파일 하는 과정에서 코드의 최적화를 수행한다.

예를 들어

|  |
| --- |
| int function () {  int a = 10;  a = 20;  a = 30;  cout << a;  } |

위 코드는 컴파일의 코드 최적화 과정을 거친다면, 불필요한 과정을 없애고

|  |
| --- |
| int function () {  int a = 30;  cout << a;  } |

위 코드 처럼 최적화를 진행한다.

그런데, 이런 최적화가 문제가 되는 상황이 존재한다.

설명을 위해 임베디드 시스템을 예로 들면, 임베디드 시스템을 구성할 때에는 메모리 맵(Memory Map) 디자인이라고 하는 과정을 거치게 되는데, 이 과정에서 출력을 위해 사용 되는 LCD나 소리를 내기 위해 사용되는 오디오 칩과 같은 하드웨어 장치에도 주소를 할당하게 된다.

|  |
| --- |
| int function () {  int \*pSound = 0x30000; // 오디오 장치 주소값 지정  \*pSound = 1; // ‘도’ 음을 내기 위해 1 전송  \*pSound = 2; // ‘미’ 음을 내기 위해 2 전송  \*pSound = 3; // ‘솔’ 음을 내기 위해 3 전송  \*pSound = 1; // ‘도’ 음을 내기 위해 다시 1 전송  . . . . .  } |

이코드를 컴파일 하면 문제가 발생한다. 컴파일러가 다음과 같이 최적화를 수행해 버리기 때문이다.

|  |
| --- |
| int function () {  int \*pSound = 0x30000; // 오디오 장치 주소값 지정  \*pSound = 1;  . . . . .  } |

위 와 같은 문제를 해결하고자 volatile 키워드를 사용하게 된다.

2. 메모리에 직접 연산하라.

Volatile로 선언되면 해당 데이터는 절대로 캐쉬되지 않는다. 인터락 함수들의 선언을 보면 포인터로 선언된 전달인자가 volatile로 선언되어 있음을 볼 수 있다. 이는 전달되는 포인터를 이용해서 함수 내부적으로 최적화를 수행하지 않으며, 해당 포인터가 가리키는 메모리 영역을 캐쉬하지 않겠다는 것을 의미한다. 따라서 여러분이 변수를 직접 volatile로 선언하고, 이렇게 선언된 변수의 주소값을 인자로 전달할 필요는 없다.

3. 커널 모드 동기화

커널 모드 동기화는 유저 모드 동기화에 비하면, 모드 전환 과정이 들어 가기 때문에 느리지만, 커널레벨에서 제공해주는 동기화 기법이기 때문에, 더 다양한 기능을 제공한다.

**뮤텍스(Mutex) 기반의 동기화**

뮤텍스 기반 동기화 기법의 경우에는 열쇠에 비유할 수 있는 것이 뮤텍스 오브젝트이고, 이는 크리티컬 섹션 오브젝트와 달리 함수를 통해서 만들어 진다.

|  |
| --- |
| HANDLE CreateMutex (  LPSECURITY\_ATTRIBUTES lpMutexAttributes,  BOOL bInitialOwner,  LPCTSTR lpName  ); |

- lpMutexAttributes : 보안속성을 지정

- bInitialOwner : 뮤텍스 오브젝트를 생성하는 스레드에게 기회를 먼저 줄 수 있다. 크리티컬 섹션 처럼 먼저 차지하는 사람이 임자가 되게 할 수도 있고(FALSE), 뮤텍스를 생성하는 스레드가 먼저 기회를 얻을 수 있다.(TRUE)

- lpName : 뮤텍스에 이름을 붙여주기 위함, 이름은 널 문자로 끝나는 문자열로 지정하면 된다. 이름을 주었을 때 생성 되는 뮤텍스를 가리켜 Named Mutex(이름있는 뮤텍스)라 표현한다.

이렇게 생성된 뮤텍스가 커널 오브젝트인 점만 보더라고, 뮤텍스는 커널레벨 동기화 기법임을 알 수 있다. 이렇게 생성되는 뮤텍스는 크리티컬 섹션 오브젝트와 달리 초기화 함수(InitializeCriticalSection)의 호출이 필요 없다. 이미 위 함수 호출 과정에서 모든 초기화가 이루어 지기 때문이다.

커널 오브젝트는 상태를 지니는데, 하나는 Signaled 상태, 다른 하나는 Non-Signaled 상태이다. 열쇠에 비유 했을 때, Non-Signaled 상태가 되고, 취득한 열쇠를 반환했을 때 Signaled 상태가 될 것이다.

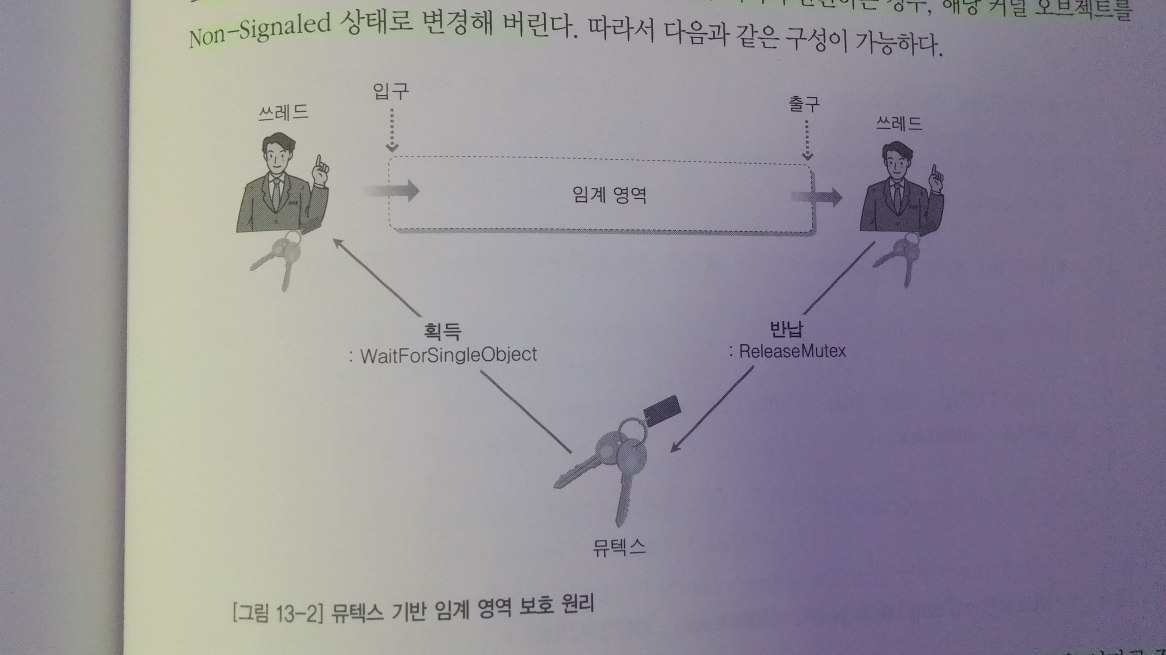
즉, 뮤텍스는 누군가에 의해 획득이 가능할 때 Signaled 상태에 놓인다. 따라서 WaitForSingleObject 함수를 임계 영역 진입을 위한 뮤텍스 획득의 용도로 사용한다.

뮤텍스를 반환할 때에는 다음 함수를 이용해 반환하며 뮤텍스는 Signaled 상태가 된다.

|  |
| --- |
| BOOL ReleaseMutex (  HANDLE hMutex;  ); |

- hMutex : 반환할 뮤텍스의 핸들을 인자로 전달한다.

한가지 특징은 WaitForSingleObject 함수의 자동 되돌림 특성이다. 이 함수는 인자로 전달된 핸들의 커널 오브젝트가 Signaled 상태가 되어서 반환하는 경우, 해당 커널 오브젝트를 Non-Signaled상태로 변경해 버려 다음과 같은 구성이 가능하다.



실행 순서는

(1) WaitForSingleObject 함수로 뮤텍스의 획득을 기다린다.

(2) 다른 스레드에서 ReleaseMutext를 호출하여 뮤텍스를 Signaled 상태로 되돌린다.

(3) WaitForSingledObject 함수를 통해 대기 중이던 스레드는 Signaled 상태의 뮤텍스를 획득하고 Non-Signaled 상태로 변경

(4) 다른 스레드의 임계 영역으로의 진입을 제한

그렇가면 할당 받은 리소스의 해제는 어떻게 되는가? 뮤텍스는 커널 오브젝트 이므로 CloseHandle 함수를 호출하여 반환한다.

[예제 13-3 CriticalSectionSyncMutex.cpp]

**세마포어(Semaphore) 기반의 동기화**

Modern Operating Systems의 저자 Tanenbaum 교수는, “세마포어 중에서 단순화된 세마포어를 가리켜 뮤텍스라 한다.”

뮤텍스와 세마포어의 차이점은 바로 카운트 기능이다. 세마포어는 카운트 기능이 존재하지만 뮤텍스에는 존재하지 않는다.

즉, **세마포어는 임계영역에 접근 가능한 스레드 개수를 조절하는 기능이 존재한다.**

다음은 세마포어 오브젝트를 생성하는 함수이다.

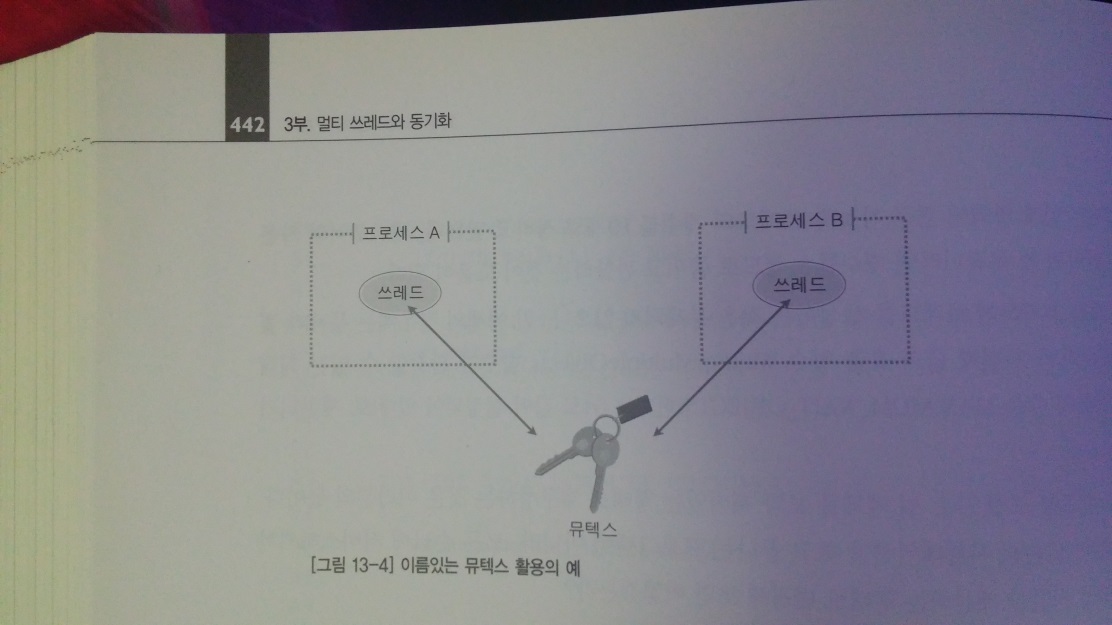
|  |
| --- |
| HANDLE CreateSemaphore (  LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,  LONG lInitialCount,  LONG lMaximumCount,  LPCTSTR lpName  ); |

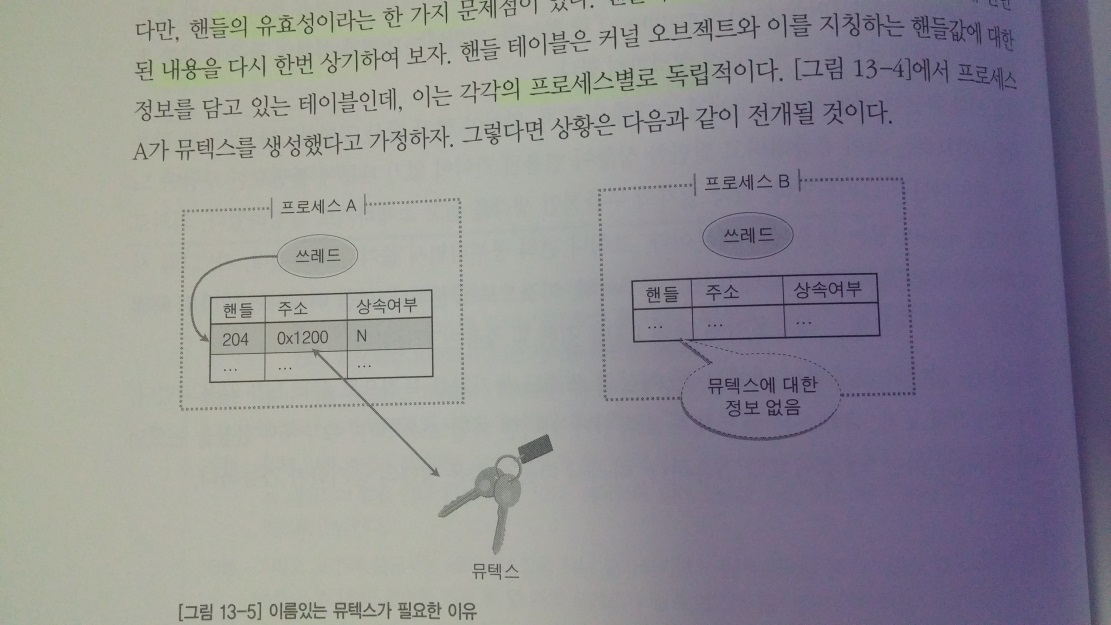
- lpSemaphoreAttributes : 보안 속성을 지정하기 위한 매개 변수

- lInitialCount : 임계 영역에 접근 가능한 스레드의 개수를 제한한다.

- lMaximumCount : 세마포어가 지닐 수 있는 값의 최대 크기를 지정한다. 이 값이 1이 될 경우 뮤텍스와 동일한 기능을 하는 바이너리 세마포어가 구성된다. lInitialCount 값보다 커야한다.

- lpName : 세마포어에 이름을 붙이기 위해 사용한다.





(1) 세마포어의 카운트를 10으로 설정한다.

(2) WaitForsingleObject를 호출하여 세마포어 하나를 반환 받으면, 세마포어의 카운트는 9가 된다.

(3) 여러 스레드에게 값을 반환하는 중 카운트가 0이 되는 순간 Non-Signaled 상태로 변환한다.

(4) 세마포어를 소유한 스레드에서 세마포어를 반환하면 카운트가 1증가 하면서 Signaled 상태가 된다.

임계영역을 빠져나온 스레드는 ReleaseSemaphore 함수를 통해 세마포어 오브젝트를 반환한다.

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

- hSemaphore : 반환하고자 하는 세마포어의 핸들을 인자로 전달

- lReleaseCount : 반환 하면서 증가시킬 세마포어의 값의 크기를 결정한다. 최대를 넘기는 잘못된 값을 반환할 경우 반환 함수가 실패 한다.

- lpPreviousCount : 변경되기 전 세마포어 카운트 값을 저장할 변수를 지정한다. 필요 없다면 NULL

[예제 13-4 MyounDongKyouJa.cpp]

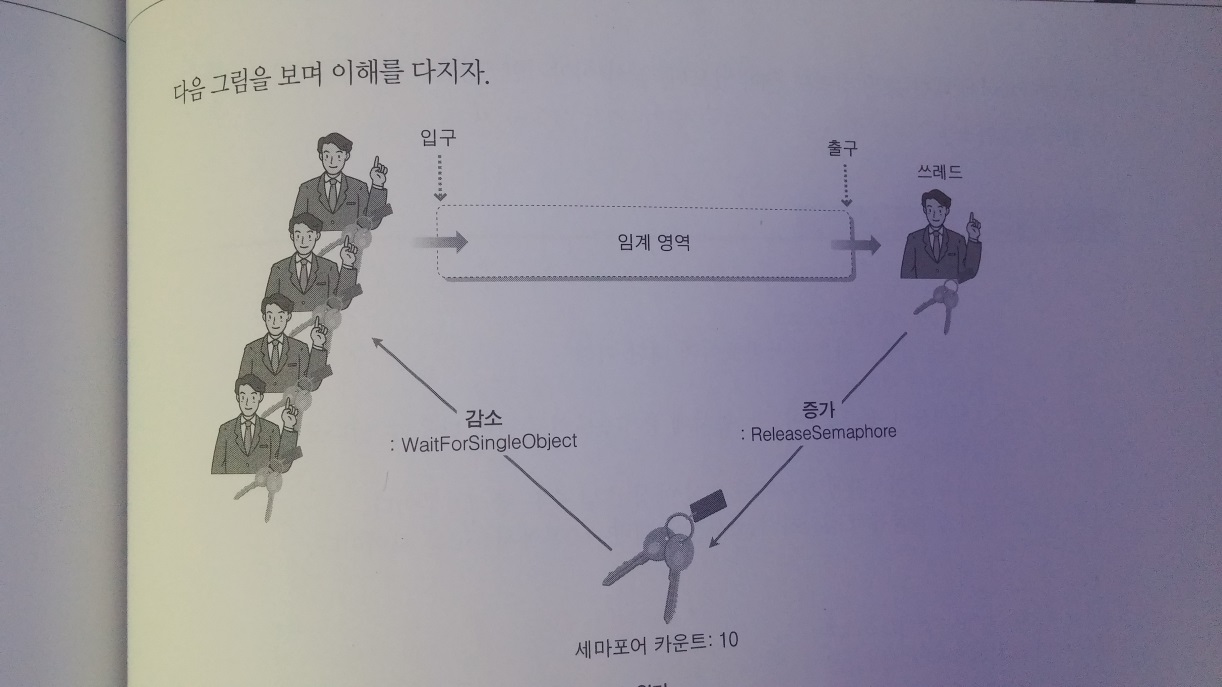
**이름있는 뮤텍스(Named Mutex) 기반의 프로세스 동기화**

뮤텍스와 세마포어의 전달인자를 자세히 보면 오브젝트에 이름을 붙여줄 수 있는데, 오브젝트에 이름을 붙여 생성할 경우 “이름 있는 뮤텍스 or 세마포어”라고 한다.

지금까지의 동기화 오브젝트는 하나의 프로세스 내에서 둘 이상의 스레드 동기화를 목적으로 뮤텍스가 사용되었다.

결론부터 말하자면, 이름있는 스레드는 서로 다른 프로세스 영역에 존재하는 스레드가 뮤텍스를 이용해서 동기화되는 것이다.

애초에 커널 동기화 오브젝트는 프로세스에서 커널로 생성요청을 하지만 오브젝트는 커널에서 관리하기 때문에, 다른 프로세스에서 접근하지 못할 이유가 없다. 하지만 **핸들의 유효성 문제가 존재**한다. 오브젝트 생성 요청을 한 프로세스의 핸들 테이블에는 동기화 오브젝트가 등록되지만, 다른 프로세스에서는 등록 되지 않기 때문이다.



이러한, 문제해결을 위해서 뮤텍스에 이름을 붙여주기로 결정한 것이다. 이 이름은 Windows라는 운영체제 내에서 유일한 이름이다. 때문에 이 이름을 통해서 Windows가 관리하고 잇는 커널 오브젝트에 접근 가능한 핸들 정보를 얻을 수 있다.

[예제 13-5 NamedMutex.cp]

예제를 실행하면 서로 다른 프로세스에 속해 있는 두 개의 스레드에서 동기화가 가능하다는 것을 확인할 수 있다.

서로 다른 프로세스에서 하나의 뮤텍스를 공유하기 위해서는 OpenMutex 함수를 사용한다.

|  |
| --- |
| HANDLE OpenMutex (  DWORD dwDesiredAccess,  BOOL bInheritHandle,  LPCTSTR lpName  ); |

- dwDesiredAccess : 이름있는 뮤텍스로의 접근 권한을 지정하는 것이다. 전달인자로 MUTEX\_ALL\_ACCESS을 전달해서 접근할 수 있는 권할을 요청해야 한다.

- bInheritHandle : 핸들의 상속 유무를 결정하기 위한 전달인자.

- lpName : 얻고자 하는 핸들 정보의 커널 오브젝트 이름을 전달

**뮤텍스의 소유와 WAIT\_ABANDONED**

뮤텍스의 WaitForSingleObject 함수의 반환 값을 보면 WAIT\_ABANDONED 라는 것이 있다. 일반적으로는 확인하기 어려운 반환값인데, 이 반환값은 무엇을 의미하는가?

뮤텍스의 경우 획득한 스레드가 직접 반환하는 것이 원칙이다. 그러나 세마포어와 그 외의 동기화 오브젝트는 다른 스레드가 반환한다 해도 문제가 되지 않는다.

이런 특성을 가진 뮤텍스를 사용하는 도중, 뮤텍스를 소유한 스레드가 예상치 못하게 종료되어 버렸다. 이럴 경우 Windows는 스레드의 상태와 뮤텍스의 상태를 감시 하고 있기 때문에, 더 이상 정상적인 방법으로 반환이 불가능한 뮤텍스를 대신 반환해 주고, 다음 대기자 스레드에게 뮤텍스를 소유할 수 있도록 해준다. 이때 **다음 대기자 스레드는 WAIT\_ABANDONED 값을 반환**하게 된다.

[예제 13-6 MUTEX\_WAIT\_ABANDONED.cpp]