이 장에서는...

- Interlocked 함수
- 임계 영역
- WaitForSingleObject와 WaitForMultipleObjects 함수
- 뮤텍스
- 세마포어
- 이벤트

4

스레드 동기화 방법

우리가 잠에서 깨어나게 하자. 이 우물이 노래를 하잖아.
— *생텍쥐페리*
어린왕자 중에서

이 장에서는 다수의 스레드를 함께 동작시키기 위해서 사용하는 Win32 API에 대해 설명한다. 먼저 가장 간단한 형태의 동기화 도구인 interlocked 함수와 임계 영역(critical section)에 대해서 살펴보고, 뮤텍스, 세마포어, 이벤트와 같이 좀더 복잡한 형태의 커널 객체 기반의 동기화 도구에 대해서 다루도록 한다. 이 장에서 설명하는 API를 통해서 대부분의 스레드 동기화 기법들을 사용할 수 있다.

Interlocked 함수

멀티스레드 프로그램은 대개의 경우에 스레드 사이에서 공유되는 변수를 갖고 있다. 이런 변수에 대한 접근은 동기화된 순서로 이루어져야 하고, 만약 이런 순서가 지켜지지 않는다면 예측하지 못한 결과가 나타날 수 있다. 예를 들어, 프로그램에서 사용되는 전체 아이템의 개수를 세는 전역 변수가 하나 있다고 하자. 스레드가 아이템의 처리를 끝낼 때마다 이 값은 하나씩 증가된다. 이 경우 스레드가 단순히 이전의 변수 값에 1을 추가하는 작업만을 한다고 하면 어떤 작업은 제대로 수행되지 않을 수도 있다. 더 자세히 알아보기 위해 아래와 같은 코드를 생각해 보자.

```
LONG g_lTotalItems = 0;

unsigned ThreadOneProc( LPVOID lpvUnused) {
.
.
// 첫째 스레드가 변수의 값을 증가시킨다.
g_lTotalItems = g_lTotalItems + 1;
.
.
.
}

unsigned ThreadTwoProc( LPVOID lpvUnused) {
.
.
// 둘째 스레드가 변수의 값을 증가시킨다.
g_lTotalItems = g_lTotalItems + 1;
.
.
.
}
```

어떻게 변수의 값이 잘못될 수 있는지 살펴보기 위해서 C 소스 코드가 정확히 기계어 코드를 일대일로 반영하고 컴파일러는 아무런 최적화(optimization)도 하지 않는다고 하자. 그리고 의사(pseudo)어셈블리 언어에서 g_lTotalItems를 증가시키는 코드는 아래와 같다고 하자.

```
load    RX,[g_lTotalItems]      ; 메모리에서 g_lTotalItems 의 값을 읽어서
                                ; 레지스터 RX에 넣는다.
inc     RX                      ; 레지스터 RX의 값을 증가시킨다.
store   [g_lTotalItems], RX     ; 레지스터 RX의 값을 메모리에 다시 저장한다.
```

g_lTotalItems의 값을 읽고 1을 증가시킨 후 마지막으로 결과가 다시 g_lTotalItems에 저장된다는 것에 주목하자. 문제는 첫째 스레드가 g_lTotalItems의 값을 읽고나서 바로 선점(preemption)당했을 때 발생한다. 둘째 스레드가 수행하게 되면 이 스레드는 첫째 스레드가 읽은 것과 같은 g_lTotalItems의 값을 읽어서 증가시킨다. 다시 제어권이 첫째 스레드로 돌아오면 이 스레드는 이전 변수값을 갖고 g_lTotalItems의 값을 증가시킨다. 결과적으로 변수의 값은 이전 계산 결과값보다 하나 작은 값을 갖는다.

이런 종류의 버그는 디버깅하기가 상당히 어렵다. 왜냐하면 이런 상황들은 간헐적으로 발생하기 때문이며, 또한 어떤 컴파일러가 코드를 만들어냈는 지와 어떤 방법으로 프로세서가 생성된 코드를 수행시키는 지에도 영향을 받는다. 따라서 이런 컴파일러의 특

이한 동작에 의존하는 것보다 코드를 처음 작성할 때 스레드 사이에서 공유 변수를 정확하게 관리하는 것이 바람직하다.

프로그래머는 이 장의 뒤에서 다룰 세련된 형태의 스레드 동기화 객체를 사용해서 이런 상황을 해결할 수도 있다. 그러나 일반적으로 멀티스레드 프로그래밍에서 정수 변수를 공유하는 것은 자주있는 일이기 때문에 Win32는 이런 작업을 위한 특별한 함수를 제공한다. Interlocked 함수라고 불리는 이런 함수는 매우 효율적으로 구현되어 시스템에 적은 부하를 주고, 멀티프로세서 시스템을 포함한 모든 시스템에서 동작할 수 있다. 이 함수는 변수가 최소 수행 단위로 읽히고 변경되는 것을 보장한다. 시스템상의 프로세스 중에서 어떤 다른 스레드도 이런 함수를 통해서 부분적으로 변수를 변경할 수 없다.

x86 멀티프로세서 시스템의 경우에 이런 함수는 변수가 4바이트 경계로 정렬(aligned)되는 조건을 필요로 한다. 대부분의 컴파일러는 디폴트에 의해서 자동적으로 이런 조건을 만족시킨다. 자세한 내용은 사용하는 컴파일러의 문서를 참조하기 바란다.

LONG 형태의 변수를 자동으로 증가시키려면 다음과 같이 한다.

```
LONG InterlockedIncrement( LPLONG      lplValue );
```

매개변수	타입	설명
lplValue	LONG*	값을 1만큼증가시킬 32비트 LONG 변수의포인터

증가시킨 값이 0보다 작으면 이 함수는 0보다 작은 값을 리턴하고, 0보다 크면 0보다 큰 값을 리턴한다. 만약 결과가 0이면 0을 리턴한다. 리턴값은 함수의 호출 결과가 0이 아니라면 호출 결과와 같을 필요는 없다. 초기의 Win32 API 문서는 리턴값이 증가시킨 값과 같다고 되어 있다. 하지만 모든 플랫폼에서 이 사실이 보장되는 것은 아니며, 이 사실에 의존적인 코딩을 하면 안 된다.

LONG 형태의 변수를 자동으로 감소시키려면 다음과 같이 한다.

```
LONG InterlockedDecrement( LPLONG      lplValue );
```

매개변수	타입	설명
lplValue	LONG*	값을 1만큼감소시킬 32비트 LONG 변수의포인터

감소시킨 값이 0보다 작으면 이 함수는 0보다 작은 값을 리턴하고, 0보다 크면 0보다 큰 값을 리턴한다. 만약 결과가 0이면 0을 리턴한다. 리턴값은 함수의 호출 결과가 0이 아니라면 호출 결과와 같을 필요는 없다. 한번 더 말하면, 초기의 Win32 API 문서는 리

턴값이 증가시킨 값과 같다고 되어 있다. 하지만 모든 플랫폼에서 이 사실이 보장되는 것은 아니며, 이 사실에 의존적인 코딩을 하면 안 된다.

현재 값을 읽고 다른 값과 최소 수행 단위 안에서 값을 바꾸기 위해서는 다음과 같이 한다.

```
LONG InterlockedExchange( LPLONG    lplTarget , LONG  lValue );
```

매개변수	타입	설명
lplTarget	LONG*	값이바뀔 32비트 LONG 변수의포인터
lValue	LONG	lplTarget에저장할새로운 32비트값

이 함수의 리턴값은 이전에 lplTarget에 저장되어 있던 값이다.

임계 영역

같은 프로세스의 스레드들은 상호 배제를 구현하기 위해서 Win32 임계 영역(Critical Section)을 사용할 수 있다. 임계 영역에는 한번에 하나의 스레드만이 들어갈 수 있다. 임계 영역에 들어가려고 하는 다른 스레드는 모두 블록당하며, 먼저 들어간 스레드가 임계 영역을 떠날 때까지 대기하게 된다. 임계 영역은 메모리 영역이나 코드 영역을 보호하는 데 사용된다. 임계 영역은 다른 프로세스에 속한 스레드를 동기화하는 데에는 사용할 수 없다.

임계 영역은 커널 객체가 아니기 때문에 스레드 동기화를 위해 제공되는 뮤텍스, 이벤트, 세마포어와 같은 다른 동기화 도구에 비해서 보다 효율적이라고 말할 수 있다. 다른 스레드가 임계 영역에 들어가 있지 않는 이상 임계 영역을 굉장히 빠른 속도로(뮤텍스를 얻는 것보다 훨씬 더 빠르다. 뮤텍스는 비슷한 기능을 하는 동기화 도구이며, 이 장의 뒷 부분에서 다룬다) 얻을 수 있다. 그러나 다른 커널 객체처럼 임계 영역도 사용자에게 불투명(opaque)한 구조를 갖고 있다. 즉 임계 영역의 자세한 내부 구조는 문서화되어 있지 않으며, 임계 영역을 직접 다루기보다는 Win32 API에 있는 함수들을 사용해서 조작해야 한다.

임계 영역을 사용하기 위해서는 스레드가 임계 영역을 동기화에 사용하기 전에 CRITICAL_SECTION 구조체를 할당하고 초기화하는 작업을 해야 한다. 이런 작업은 애플리케이션에 전역 변수를 선언하고 프로그램이 시작할 때 InitializeCriticalSection 함수를 호출함으로써 이루어진다.

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION    lpCriticalSection );
```

매개변수	타입	설명
lpCriticalSection	LPCRITICAL_SECTION	프로그램에의해서할당된불투명한임계영역구조체에대한포인터

이 함수는 절대로 실패하지 않으며, 임계 영역에는 관련된 HANDLE이 존재하지 않는다. 또한 이 함수에는 SECURITY_ATTRIBUTES 매개변수가 사용되지 않는다. 이런 사실은 임계 영역이 커널 객체가 아니라는 증거가 될 수 있다. 같은 프로세스 안에서 다수의 스레드에 의해 공유되는 데이터에 접근을 시도하기 전에 임계 영역을 획득하기 위해서는 EnterCriticalSection 함수가 사용된다.

```
VOID EnterCriticalSection( LPCRITICAL_SECTION    lpCriticalSection );
```

매개변수	타입	설명
lpCriticalSection	LPCRITICAL_SECTION	프로그램에의해서할당된불투명한임계영역구조체에대한포인터. 이구조체는반드시 InitializeCriticalSection 함수의호출에의해서적어도한번은초기화되어야한다.

이 함수는 리턴값이 없다. 특히 시간 초과 매개변수가 없다는 사실에 주의해야 한다. 임계 영역에 들어가려는 스레드는 임계 영역이 사용 가능해질 때까지 무기한으로 블록당하기 때문이다. 임계 영역은 같은 스레드에 의해 재귀적으로 계속해서 획득될 수 있다. 따라서 이미 EnterCriticalSection 함수를 호출하고 리턴한 스레드가 교착 상태에 빠지지 않고 다시 이 함수를 호출할 수 있다. 임계 영역은 내부적인 카운터를 갖고 있기 때문에 임계 영역을 획득한 횟수만큼 반드시 해제(release)해야 한다.

TryEnterCriticalSection 함수는 임계 영역에 블록킹하지 않고 들어가려고 할 때 사용되는 함수이다.

```
BOOL TryEnterCriticalSection( LPCRITICAL_SECTION    lpCriticalSection );
```

매개변수	타입	설명
lpCriticalSection	LPCRITICAL_SECTION	프로그램에의해서할당된불투명한임계영역구조체에대한포인터.

이 함수는 윈도우95에서는 사용할 수 없으며, 윈도우NT 4.0이나 그 이후의 버전에서만 사용할 수 있다.

임계 영역을 블로킹 없이 획득한 경우나 함수를 호출한 스레드가 이미 임계 영역에 들어가 있는 경우에 이 함수는 TRUE를 리턴한다. 그러나 다른 스레드가 이미 임계 영역을 소유한 경우에는 이 함수가 FALSE를 리턴한다. 이 함수가 임계 영역에 들어가는 데 사용될 경우에는 TryEnterCriticalSection 함수가 TRUE를 리턴하는 경우에만 조건적으로 LeaveCriticalSection 함수를 호출할 수 있다.

임계 영역을 해제하기 위해서는 LeaveCriticalSection 함수가 사용된다.

```
VOID LeaveCriticalSection( LPCRITICAL_SECTION      lpCriticalSection    );
```

매개변수	타입	설명
lpCriticalSection	LPCRITICAL_SECTION	프로그램에의해서할당된불투명한임계영역구조체에대한포인터. 이함수를호출하는스레드는반드시이전에 EnterCriticalSection 또는 TryEnterCriticalSection을성공적으로호출한상태이어야한다. 그렇지않을경우에는임계영역을대기하고있는스레드가무한히대기하게되는에러가발생한다.

이 함수는 리턴값이 없다. 이 함수는 EnterCriticalSection이나 TryEnterCriticalSection 함수의 호출이 성공할 경우에 같은 스레드에서 한번씩 대응되어 호출되어야 한다.

InitializeCriticalSection 함수에 의해서 할당된 자원을 해제하기 위해서는 DeleteCriticalSection 함수를 사용해야 한다.

```
VOID DeleteCriticalSection( LPCRITICAL_SECTION      lpCriticalSection    );
```

매개변수	타입	설명
lpCriticalSection	LPCRITICAL_SECTION	프로그램에의해서할당된불투명한임계영역구조체에대한포인터

이 함수는 리턴값이 없다. 이 함수가 호출되면 이 함수에 사용된 임계 영역은 더 이상 스레드 동기화를 위해서 사용할 수가 없다.

다음은 임계 영역의 사용을 나타내는 간단한 예제이다.

```
CRITICAL_SECTION cs;
.
.
.
// 사용하기 전에 임계 영역을 초기화한다.
InitializeCriticalSection(&cs);
.
.
.
// 임계 영역에 들어간다.
EnterCriticalSection(&cs);

// 임계 영역에 들어가려고 시도한다.
BOOLEAN bStatus = TryEnterCriticalSection(&cs);

// 임계 영역에서 나간다
LeaveCriticalSection(&cs);

// TryEnterCriticalSection    이 성공한 경우에는 다시 한번 나가야 한다.
if (bStatus)
    LeaveCriticalSection(&cs);
.
.
.
// 임계 영역의 사용이 끝나면 삭제한다.
DeleteCriticalSection(&cs);
```

TryEnterCriticalSection 함수의 호출이 성공하면 임계 영역을 두 번 나가야 한다는 사실을 주의해야 한다. 임계 영역은 스레드가 몇 번이나 임계 영역을 획득했는지를 나타내는 내부 카운터를 갖고 있기 때문에 획득한 만큼 반드시 임계 영역을 해제해야 한다. 임계 영역의 내부 카운트 값이 0일 때에만 다른 스레드가 임계 영역을 획득할 수 있다.

다수의 스레드가 같은 임계 영역에 들어가려고 경쟁하는 경우에는 높은 우선순위의 스레드가 임계 영역에 들어가며, 다른 스레드들은 대기하게 된다. 둘 이상의 스레드가 높은 우선순위에서 임계 영역에 들어가기 위해서 대기하고 있다면, 운영체제가 어떤 스레드를 먼저 임계 영역에 들어가게 할지 결정하게 된다. 대개의 경우에 같은 우선순위의 스레드 중에서는 먼저 임계 영역에 들어가기 위해서 대기하고 있던 스레드가 우선권을 갖는다. 하지만 이것은 일반적인 경향이고, Win32 운영체제가 이런 동작을 보장하는 것은 아니다.

WaitForSingleObject 함수와 WaitForMultipleObjects 함수

앞에서 모든 커널 객체는 두 가지 상태(시그널된(signaled), 시그널되지 않은(nonsignaled) 상태) 중 하나에 속한다는 것을 설명했다. 프로그래머는 종종 특정 커널 객체가 시그널되기 전까지 스레드를 대기하게 했다가 시그널된 후에 동작하도록 하는 방법을 사용한다. 대기하는 데 사용할 수 있는 커널 객체의 종류는 매우 다양하지만 특정 커널 객체나 또는 커널 객체들이 시그널되기를 대기하기 위한 함수는 WaitForSingle- Object와 WaitForMultipleObjects 두 가지이다. [표4-1]은 대기하는 데 사용할 수 있는 커널 객체의 종류와 각 객체의 시그널 상태를 변경시킬 수 있는 함수를 설명하고 있다. 프로세스와 스레드 커널 객체는 이미 '3장. 프로세스와 스레드 API 에서 설명했고, 이 장에서는 뮤텍스, 세마포어, 이벤트 객체에 대해서 설명하며, 파이프(pipe)는 '5장. 스레드간 통신 에서 설명하도록 한다. 이 표에 있는 함수는 이 책에서 다룰 모든 객체의 형태이다.

[표4-1] 대기하는데 사용할 수있는 커널객체

커널 객체	시그널된(Signaled)	시그널되지 않은(Nonsignaled)
프로세스	프로세스가종료한경우. 프로세스가활성화된스레드를하나도갖고있지않거나 (모든스레드가 ExitThread를호출했으면) 프로세스안의스레드가 ExitProcess를 호출하면프로세스가종료하게된다.	프로세스가적어도하나의활성화된스레드를갖고있는경우
스레드	스레드가종료한경우. 스레드가 ExitThread를 호출했거나스레드시작프로시저에서리턴했거나또는 TerminateThread가스레드를 종료시켰을경우가해당된다	스레드가종료하지않는경우
이벤트	SetEvent와 PulseEvent 함수에의해서 시그널상태가될수있다.	자동리셋이벤트는스레드가이벤트에 대한대기작업을끝낸후에운영체제에 의해서바로리셋된다(시그널되지않은 상태로된다). 수동리셋이벤트는반드시 ResetEvent 함수를호출해서명시적으로 시그널되지않은상태로만들어야한다.
뮤텍스	뮤텍스객체가컴퓨터의 CPU에서수행하고 있는어떤스레드에도소유되지않은경우. 스레드는뮤텍스의소유권을 ReleaseMutex 함수를호출해서포기할수있다.	뮤텍스를소유한스레드가종료하지않았거나, 자발적으로뮤텍스를해제하지않은 경우. 대기한후에스레드가성공적으로 뮤텍스를획득한경우에는그스레드에게 제어권을넘겨주기전에운영체제가뮤텍스를시그널되지않은상태로만든다.

커널 객체	시그널된(Signaled)	시그널되지 않은(Nonsignaled)
세마포어	세마포어객체와관련된카운트가 0보다큰 경우. 세마포어의초기값과최대값은세마 포어가 CreateSemaphore 함수를사용해서 생성될때지정된다. ReleaseSemaphore 함수는주어진세마포어의카운트를명확히 증가시키는데사용된다.	세마포어의카운트값이 0인경우. 카운트 가 0보다클때세마포어를기다리고있는 경우에는wait 함수가카운트를 1 만큼감 소시키고즉시리턴하게된다.
변화공지 (Change Notification)	주어진디렉토리나트리에서 FindFirst-ChangeNotification 함수에서지정한변화가 발생한경우	변화공지객체가처음으로생성된경우 또는 애플리케이션이FindNextChange-Notification을호출한경우
콘솔입력	콘솔의입력버퍼에읽지않은데이터가 있는경우. 콘솔의입력버퍼는 WriteFile 이나 WriteConsoleInput 함수를사용해서 쓸수있다.	입력버퍼가비어있는경우 . 콘솔의입력 버퍼는 ReadFile, ReadConsole, ReadConsoleInput 함수등을사용해서 읽을수있다.
파일	파일이나통신디바이스에대한중복된 (overlapped) 읽기나쓰기작업이운영체제에 의해서충족된경우. GetOverlappedResult 함수는시그널되기위한파일핸들을기다리는 데 사용될수 있다. 통신디바이스는 Wait-CommEvent 함수를주어진통신이벤트가발생 할때까지대기하도록하는데사용할수있다.	중복된쓰기나읽기가파일이나통신디 바이스에서대기하고있지않는경우 . 통 신디바이스의경우에는 SetCommMask 에의해서지정된이벤트중의하나가발 생할때까지파일이시그널되지않은상 태로남게된다.
파이프	파이프에읽지않은데이터가있는경우 . 주어진파이프에해당되는쓰기핸들을 이용해서 WriteFile 함수를사용하면 파이프에데이터를써넣을수있다 .	파이프가비어있는경우 . 주어진파이프 에해당되는읽기핸들을이용해서 ReadFile 함수를사용하면파이프의데이 터를읽을수있다 .
이름을가진 파이프	이름을가진파이프에읽지않은데이터가 있거나클라이언트가이름을가진파이프에 연결한경우. 이름을가진파이프에는 WriteFile, TransactNamedPipe, CallNamedPipe 함수를 이용해서데이터를써넣을수있다 . 클라이언트애플리케이션은 WaitNamedPipe 함수로파이프를사용할수있을때까지대기 할수있고 , CreateFile이나 CallNamedPipe 함수를 사용해서이름을가진파이프에연결 할수있다 . 이름을가진파이프는윈도우 95 에서는원격으로만열수있고 , 윈도우 NT 에서만생성할수있다 .	이름을가진파이프가비었거나이름을 가진파이프에접속하려는클라이언트가 없는경우. 이름을가진파이프의데이터 는 ReadFile, TransactNamedPipe 함수를 이용해서읽을수있다 . 서버는서버쪽의 이름을가진파이프의연결을끊을수있 고, DisconnectNamedPipe 함수를호출해 서이객체를시그널되지않은상태로만 들수있다 . 또한 서버는 ConnectNamed-Pipe 함수를사용해서이름을가진파이 프에새로운클라이언트가접속할때까지 대기할수있다.
메일슬롯 (Mailslot)	메일슬롯에읽지않은데이터가있는경우 . 클라이언트프로세스는 WriteFile 함수를 이용해서메일슬롯에데이터를써넣을 수있다.	메일슬롯이빈경우 . 서버프로세스는 ReadFile 함수를이용해서메일슬롯에 서데이터를읽을수있다 .

커널 객체	시그널된(Signaled)	시그널되지 않은(Nonsignaled)
I/O 완료포트 (Completino Port)	완료포트에해당되는파일에대한입력이나 출력작업이완료된경우 . 같은프로세스안의 스레드는 PostQueuedCompletionStatus 함수를이용해서 I/O 완료포트를시그널된 상태로만들수있다 .	완료포트에해당되는파일에대한입력 이나출력작업이초기화되고종료될때 까지의경우. GetQueuedCompletionStatus 함수를 이용해서주어진 I/O 완료포트에 완료공지가발생할때까지대기할수있다.

하나의 커널 객체가 시그널되기를 대기할 때는 WaitForSingleObject 함수를 사용할 수 있다.

```
DWORD WaitForSingleObject( HANDLE    hHandle , DWORD  dwMilliseconds  );
```

매개변수	타입	설명
hHandle	HANDLE	커널객체를나타낸다. [표 4-1]에서설명한커널객체중하나를 사용할수있다 .
dwMilliseconds	DWORD	지정한객체가시그널되지않을경우에이함수가리턴해야되 는시간초과간격을 ms 단위로지정한다. 무한히시그널되기를 기다리기위해서는 INIFINITE를사용한다.

이 함수는 실행 결과에 따라서 다음에 나열한 값 중의 하나를 리턴한다.

리턴값	의미
WAIT_FAILED	대기동작이실패했다는것을나타낸다 . GetLastError 함수를이용하면실패한이 유에대한자세한정보를얻을수있다 .
WAIT_OBJECT_0	커널객체가주어진시간간격안에시그널된상태로변했다 .
WAIT_TIMEOUT	커널객체가시그널되기전에주어진시간초과간격이모두지났다 .
WAIT_ABANDONED_0	커널객체가뮤텍스를사용했고 , 뮤텍스를소유한스레드가자발적으로뮤텍스의 소유권을해제하지않고종료하거나종료되었다 .

이 함수는 세 가지의 형태로 실패할 수 있기 때문에, 실패할 경우인지 판단하기 위해서 WAIT_FAILED에 대해 검사하는 대신에 성공하는 경우를 찾기 위해서 리턴값이 WAIT_OBJECT_0인지를 검사해야 한다. 예를 들어, 뮤텍스가 사용 가능해지기를 대기하고 있고, 다음과 같은 코드를 작성했다고 하자.

```
// WaitForSingleObject     함수가 성공했는지 확인하는 " 불확실한 " 방법
//
DWORD dwWaitResult;
```

```
dwWaitResult = WaitForSingleObject(hMutex, 1000);

if( WAIT_FAILED == dwWaitResult )
{
    // 실패한 경우를 처리한다.
}
 else
{
    // 뮤텍스를 소유한 것처럼 생각하고 진행한다(잠재적 버그를 갖고 있음).
}
```

위의 코드는 대기 시간 초과(WAIT_TIMEOUT)나 뮤텍스 포기(WAIT_ABANDONED_0)와 같은 경우를 성공한 경우로 잘못 판단할 가능성을 갖고 있다. WaitForSingleObject의 결과를 검사하는 안전한 코드의 형태는 다음과 같다.

```
// WaitForSingleObject     함수가 성공했는지 확인하는 " 안전한 " 방법
//
DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(hMutex, 1000);

if( WAIT_OBJECT_0 == dwWaitResult )
{
     // 뮤텍스를 소유한 것처럼 생각하고 진행한다.
}
else
{
     // 핸들을 대기하는 것을 실패했다.
     // 일반적인 실패(WAIT_FAILED),   대기 시간 초과(WAIT_TIMEOUT),
     // 뮤텍스 포기(WAIT_ABANDONED_0)    중에서 필요에 따라 실패 코드를
     // 분리하면 된다.
}
```

뮤텍스 포기에 대해서는 이 장의 뒷부분에서 자세히 설명하도록 한다.

때때로 프로그래머는 다수의 커널 객체가 모두 시그널되도록 대기하거나 이 중 하나가 시그널되도록 대기해야 할 필요가 있다. WaitForMultipleObjects 함수는 이런 경우에 사용할 수 있으며, 다음과 같은 원형을 갖는다.

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

매개변수	타입	설명
nCount	DWORD	lpHandles가가리키는배열에존재하는커널객체의개수를나타낸다. 객체의최대개수는 MAXIMUM_WAIT_OBJECTS로정의되며, 현재는 64의값을갖는다.
lpHandles	CONST HANDLE*	커널객체의핸들로구성된배열의포인터. 이배열은 [표 4-1]에나열한객체중어떤것의핸들도포함할수있다.
bWaitAll	BOOL	지정한객체들중하나라도시그널되면리턴할것인지또는모든커널객체가시그널되면 리턴할 것인지를결정하는플래그. 값이 TRUE이면모든객체가시그널될때까지대기하고, FALSE이면지정한객체중하나라도시그널되면리턴한다.
dwMilliseconds	DWORD	지정한객체또는객체들이시그널되지않았을경우에언제리턴할지를결정하는시간초과간격을 ms 단위로지정한다. 계속대기하기위해서는 INFINITE를사용하면된다.

이 함수는 실행 결과에 따라서 다음에 나열한 값 중에서 하나를 리턴한다.

리턴값	의미
WAIT_FAILED	대기동작이실패한것을나타낸다. GetLastError 함수를호출하면실패한이유에대한자세한정보를얻을수있다.
WAIT_OBJECT_0에서 (WAIT_OBJECT_0 + nCount -1)까지	리턴값이이범위에있거나 bWaitAll이 TRUE이면지정된모든객체가시그널된것이다. bWaitAll이 FALSE이면리턴값에서 WAIT_OBJECT_0을뺀값이핸들의배열중에서시그널된객체의개수를의미한다. 하나이상의객체가시그널된경우에는시그널된모든객체중에서가장낮은인덱스값을갖는핸들의인덱스값을리턴한다.
WAIT_TIMEOUT	커널객체가시그널되기전에주어진시간초과간격이모두지났다.
WAIT_ABANDONED_0에서 (WAIT_OBJECT_0 + nCount -1)까지	지정된객체중의하나가뮤텍스객체이고, 이객체가스레드에의해서자발적으로해제되지못했음을나타낸다. 포기한뮤텍스를나타내는핸들의인덱스는리턴값에서 WAIT_ABANDONED_0을뺀값이다.

WaitForSingleObject와 마찬가지로 이 함수의 리턴값은 주의 깊게 검사해야 한다. 그렇지 않으면 재현하기 어렵고 분석하기 힘든 버그를 프로그램 안에 만들 가능성이 있다. 먼저 성공적인 대기 작업에 대해서 테스트하는 것이 안전하고, 실패한 경우에 대해서는 한꺼번에 처리하는 것이 좋다. 물론 때때로 실패한 경우를 나누어서 분석해야 할지도 모른다. 그러나 이렇게 실패한 경우를 한꺼번에 처리함으로써 시간 초과나 뮤텍스 포기를 성공한 작업으로 잘못 처리하는 경우를 피할 수가 있다. WaitForMultipleObjects의 리턴값을 해석하는 것은 어떤 객체가 시그널됐는지 판단하거나 어떤 객체가 포기되었는지(뮤텍스 객체인 경우) 판단하기 위해서 리턴값에 산술적인 계산을 추가해야 하기 때문에 좀더 복

잡하다고 할 수 있다. 다음의 예제는 다수의 객체가 시그널될 때까지 대기하도록 일꾼 스레드를 어떻게 설계하는지 보여주고, WaitForMultipleObjects의 리턴값을 어떤 방법으로 안전하게 분석할 수 있는지 나타내고 있다.

```
// WaitForMultipleObjects     함수의 성공 여부를 안전하게 검사하는 방법
//
#define NUM_OBJECTS 2

DWORD   dwWaitResult;
HANDLE  hObjects[NUM_OBJECTS];

// 대기하고 있을 커널 객체들의 핸들 표를 초기화한다.
//
hObjects[0] = hThreadExitEvent;
hObjects[1] = hResourceMutex;

// 다음 중 하나가 발생할 때까지 계속해서 대기한다.
// (1) 이 스레드가 종료 명령을 받았고
// 다른 스레드가 수동 리셋 이벤트를 시그널 함으로서 알려진 경우
// (2) 이 스레드가 작업을 하기 위해서 필요한 자원이 사용 가능한 경우
// (hResourceMutex)
//
dwWaitResult = WaitForMultipleObjects(NUM_OBJECTS, hObjects, FALSE,
                                      INFINITE);

// 리턴한 이유를 판단한다.
//
if(
     (dwWaitResult >= WAIT_OBJECT_0) &&
     (dwWaitResult <= WAIT_OBJECT_0 + NUM_OBJECTS - 1)
)
{
     // 리턴값이 객체 중 하나가 시그널된 것을 나타내는 범위에 있는 경우이다.
     // 이제 이 객체가 스레드 종료 이벤트인지 자원 뮤텍스인지를 판단하자.
     //
     switch( dwWaitResult - WAIT_OBJECT_0 )
     {
          case 0:
             // hThreadExitEvent  에 의해 참조되는 이벤트가 시그널되었다.
             // 이 스레드를 종료할 필요가 있는 경우
             //
          break;
```

```
            case 1:
               // hResourceMutex  에 의해 참조되는 뮤텍스가 이 스레드에 의해서
               // 소유되었고, 이 스레드가 새로운 작업을 할 수 있게 된 경우
            break;
      }
}
else
{
      // 대기 작업이 실패한 경우를 처리한다.
      // 필요에 따라 실패한 이유가 일반적인 실패(WAIT_FAILED),  대기 시간 초과
      // (WAIT_TIMEOUT),   뮤텍스 포기(WAIT_ABANDONED_0)   인지 분석한다.
 }
```

대기하고 있는 객체가 뮤텍스인 경우에는 에러 처리가 좀더 복잡해진다. 뮤텍스를 갖고 있던 스레드가 뮤텍스를 해제하지 않고 종료하는 경우에 다른 스레드의 뮤텍스는 포기(abandoned)된다는 사실을 명심해야 한다. 대기하고 있는 객체 중 하나 이상이 뮤텍스이고 다른 스레드에 의해 그 뮤텍스가 포기됐다면, 리턴값은WAIT_ABANDONED_0에서 (WAIT_ABNDONED_0 + NUM_OBJECTS-1)의 범위 안에서 값을 갖게 된다.

WaitForMultipleObjects 함수를 프로그램에서 사용하는 경우에는 리턴값을 해석하는 것이 간단하지 않기 때문에 언젠가는 부득이하게 코드에 버그를 만들 수도 있다. 불행히도 이런 형태의 버그는 재현하기 힘들고(모든 실패의 모드가 예외적인 경우이기 때문에) 분석하기에도 힘들다. 이런 이유 때문에 작은 매크로나 아래의 코드와 같은 인라인(inline) 함수를 만들어서 리턴값을 분석하면 시간 낭비를 줄일 수 있다.

```
inline BOOL WaitSucceeded( DWORD dwWaitResult, DWORD dwHandleCount )
{
      // 대기 동작이 성공하면 TRUE 를 리턴하고 실패하면 FALSE 를 리턴한다.
      //
      return
         BOOL(
            (dwWaitResult >= WAIT_OBJECT_0) &&
            (dwWaitResult <= WAIT_OBJECT_0 + dwHandleCount - 1)
         );
}

inline BOOL WaitAbandoned( DWORD dwWaitResult, DWORD dwHandleCount )
{
      // 객체 중 하나가 포기된 뮤텍스이면 TRUE 를 리턴하고, 아니면 FALSE 를 리턴한다.
      //
      return
         BOOL(
            (dwWaitResult >= WAIT_ABANDONED_0) &&
```

```
            (dwWaitResult <= WAIT_ABANDONED_0 + dwHandleCount - 1)
        );
}

inline BOOL WaitTimedOut( DWORD dwWaitResult )
{
    // 대기 동작이 시간 초과이면 TRUE 를 리턴하고, 아니면 FALSE 를 리턴한다.
    //
    return BOOL(WAIT_TIMEOUT == dwWaitResult);
}

inline BOOL WaitFailed( DWORD dwWaitResult )
{
    // 대기 동작이 실패하면 TRUE 를 리턴하고, 아니면 FALSE 를 리턴한다.
    // 이 함수가 TRUE 를 리턴할 경우에는 GetLastError()  함수를 호출해서
    // 실패한 이유에 대한 자세한 정보를 얻을 수 있다.
    //
    return BOOL(WAIT_FAILED == dwWaitResult);
}

inline DWORD WaitSuccessIndex( DWORD dwWaitResult )
{
    // 포기된 객체의 0 기준(0-based)  인덱스를 리턴한다.
    // 이 함수는 WaitSucceeded()  함수가 TRUE 를 리턴한 경우에만
    // 호출되어야 한다.
    //
    return(dwWaitResult - WAIT_OBJECT_0);
}

inline DWORD WaitAbandonedIndex( DWORD dwWaitResult )
{
    // 포기된 객체의 0 기준(0-based)  인덱스를 리턴한다.
    // 이 함수는 WaitAbandoned()  함수가 TRUE 를 리턴한 경우에만
    // 호출되어야 한다. 리턴된 인덱스가 N번째 객체를 가리킨다고 해서
    // 이것이 N번째 뮤텍스를 가리키는 것은 아니다.
    //
    return(dwWaitResult - WAIT_ABANDONED_0);
}
```

앞에서 만든 유용한 함수를 이용해서 만든 안전한 리턴값 분석 코드는 아래와 같다.

```
// 리턴값을 분석하기 위한 유용한 함수를 사용해서
// WaitForMultipleObjects    함수의 성공 여부를 안전하게 검사하는 방법
//
#define NUM_OBJECTS 2
```

```
DWORD  dwWaitResult;
HANDLE hObjects[NUM_OBJECTS];

// 대기하고 있을 커널 객체들의 핸들 표를 초기화한다.
//
hObjects[0] = hThreadExitEvent;
hObjects[1] = hResourceMutex;

// 다음 중 하나가 발생할 때까지 계속해서 대기한다.
// (1) 이 스레드가 종료하도록 다른 스레드가 수동 리셋 이벤트를 시그널함으로서 알려진
// 경우 또는 (2) 이 스레드가 작업을 하기 위해서 필요한 자원이 사용 가능한 경우
// (hResourceMutex)
//
dwWaitResult= WaitForMultipleObjects(NUM_OBJECTS,hObjects,FALSE,
                                     INFINITE);

// 리턴한 이유를 판단한다.
//
if( WaitSucceeded(dwWaitResult, NUM_OBJECTS) )
{
     // 리턴값이 객체 중 하나가 시그널된 것을 나타내는 범위에 있는 경우이다.
     // 이제 이 객체가 스레드 종료 이벤트인지 자원 뮤텍스인지를 판단하자.
     //
     switch( WaitSuccessIndex(dwWaitResult) )
     {
          case 0:
              // hThreadExitEvent  에 의해 참조되는 이벤트가 시그널되었다.
              // 이 스레드를 종료할 필요가 있는 경우
              //
          break;

          case 1:
              // hResourceMutex  에 의해 참조되는 뮤텍스가 이 스레드에 의해서
              // 소유되었고, 이 스레드가 새로운 작업을 할 수 있게 된 경우
          break;
     }
}
else
{
     // 대기 작업이 실패한 경우를 처리한다.
     //
     if( WaitAbandoned(dwWaitResult, NUM_OBJECTS) )
     {
          // 객체 중 하나 이상이 뮤텍스인 경우에는
          // 어떤 뮤텍스가 포기되었는지 검사하기 위해서
```

```
            // WaitAbandonedIndex()  를 사용할 수 있다.
        }
        else if( WaitTimedOut(dwWaitResult) )
        {
            // 시간 초과 실패의 경우이다.
        }
        else
        {
            // 일반적인 실패의 경우를 처리한다.
        }
    }
```

WaitForSingleObject나 WaitForMultipleObjects 함수 모두 성공한 경우에는 WAIT_OBJECT_0 기준의값을 리턴하고, 뮤텍스 포기의 경우에는 WAIT_ABANDONED_0 기준의 값을 리턴하기 때문에 이런 함수는 WaitForMultipleObjects 뿐만 아니라 WaitForSingleObject 함수의 리턴값을 안전하게 분석하는 데에도 사용할 수 있다. 단지 dwHandleCount에 1을 인자로 넘겨주기만 하면 된다.

리턴값을 분석하는 것도 중요하지만, WaitForMultipleObjects 함수에 객체 핸들의 배열을 넘겨줄 때 핸들의 순서를 어떻게 할지 잘 생각해야 한다. 만약 객체 중 어떤 하나라도 시그널되기를 대기하고 있다면(bWaitAll이 FALSE), 시그널된 객체 중 가장 낮은 인덱스 값의 객체가 선택되고, 그 객체 핸들의 인덱스 값이 리턴된다. 예를 들어, 위의 코드를 뮤텍스 핸들이 hObjects에서 인덱스 0이 되도록 재배치하고 스레드 종료 이벤트 객체를 인덱스 1로 했다면, 스레드 종료 이벤트가 오랫동안 시그널된 상태로 있는 것을 확인할 수가 없다. 이런 상황은 대기 동작을 할 때마다 뮤텍스가 사용 가능해도 발생할 수 있다. 주 스레드는 이 일꾼 스레드가 스레드 종료 이벤트를 시그널링하도록 지정했더라도 WaitForMultipleObjects를 호출했을 때 뮤텍스가 더 이상 시그널된 상태가 아닐 때까지 일꾼 스레드는 얼마동안 자신이 하던 일을 계속해서 진행할 수 있다(무한히 진행할 수도 있다). 단지 일꾼 스레드는 스레드 종료 이벤트가 시그널된 것을 발견하게 될 뿐이다. 대부분의 경우에는 단순히 핸들을" 높은 우선순위 "의 이벤트 객체로(위의 스레드 종료 이벤트와 같이) 만들거나 핸들 배열의 앞쪽에 가까운 예외 조건에서만 시그널되는 객체로 만드는 것만으로 충분하다.

WaitForSingleObject 함수를 여러 번 사용하기 위해 bWaitAll 매개변수를 TRUE로 지정한 WaitForMultipleObjects 함수로 바꿀 때에는 여러 가지를 고려해야 한다. 프로그램이 두 개의 객체를 동시에 필요로 하는 경우에는 두 객체가 모두 사용 가능해질 때까지 대기한 후에 두 객체를 동시에 획득하는 것이 더 좋은 방법이다. 그렇지 않으면 교착 상태에 빠지게 될 수도 있다. 예를 들어, 객체 A를 획득했지만 객체 B는 다른 스레드에 의해서 획득된 경우, 둘째 스레드가 객체 B를 해제하기 위해서 객체 A가 필요하다면 스레드는

교착 상태에 빠지게 된다. 만약 A와 B를 동시에 획득하기 위해서 WaitForMultiple-Objects 함수를 사용했다면 교착 상태에 빠지는 것을 피할 수 있다.

뮤텍스

스레드 동기화를 위해서 가장 일반적으로 사용되는 커널 객체 중의 하나가 Win32 뮤텍스이다. 뮤텍스는 커널 객체이며, 시스템에 있는 자원의 소유권을 서로 배타적으로 획득할 수 있도록 한다. 그러므로 임계 영역과는 달리 뮤텍스는 서로 다른 프로세스에 속한 스레드가 동일 자원에 대해서 접근하는 경우에도 동기화하는 데 사용할 수 있다. 일반적으로 자원의 형태는 같은 프로세스나 다른 프로세스에 속한 다수의 스레드에 의해 공유되는 데이터 구조이다. 어디까지나 뮤텍스는 도구일 뿐이고, 멀티스레드를 사용하면서 발생하는 문제는 결국 프로그래머가 해결해야 한다.

뮤텍스 객체는 CreateMutex 함수를 사용해서 생성한다

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES  lpMutexAttributes ,
                    BOOL bInitialOwner , LPCTSTR lpName );
```

매개변수	타입	설명
lpMutexAttributes	LPSECURITY_ ATTRIBUTES	뮤텍스객체의보안속성을나타내는 SECURITY_ATTRIBUTES 구조체의포인터. 이값이 NULL이면뮤텍스는기본보안속성으로생성된다.
bInitialOwner	BOOL	CreateMutex 함수에서리턴한후에함수를호출한스레드가뮤텍스를소유할지의여부를결정하는플래그 . TRUE로 값을지정하면초기에뮤텍스를소유하게된다 . FALSE를지정하면초기에뮤텍스를소유하지않고생성한다 .
lpName	LPCTSTR	뮤텍스객체의이름을나타내는종료된 널문자열의포인터. 이름없이뮤텍스를생성하려면 NULL로지정하면된다. 이름을가진뮤텍스는다른프로세스의스레드가열어서사용할수있다 . 이름이없는뮤텍스는뮤텍스를생성한프로세스안의스레드에게리턴된핸들을통해서만보이게된다 .

CreateMutex의 호출이 성공하면 뮤텍스 객체의 핸들이 리턴된다. lpName을 지정했고 같은 이름을 가진 뮤텍스가 이미 존재한다면, 함수의 호출은 성공하고 GetLastError를 호출하면 ERROR_ALREADY_EXISTS를 리턴하게 된다. 함수의 호출이 실패하면 NULL

이 리턴되고 GetLastError를 호출하면 실패에 대한 좀더 자세한 정보를 얻을 수 있다. 뮤텍스, 세마포어, 이벤트와 파일 매핑은 모두 같은 이름 공간(namespace)을 사용한다는 사실에 주의해야 한다. lpName을 지정한 경우에 같은 이름을 뮤텍스가 아닌 다른 객체가 사용하고 있다면 이 함수의 호출은 실패하고, GetLastError를 호출하면 ERROR_INVALID_HANDLE을 리턴하게 된다. 뮤텍스 객체를 닫기 위해서는 CloseHandle 함수를 사용한다.

bInitialOwner 매개변수는 주의해서 사용해야 한다. 두 개의 프로세스가 CreateMutex를 사용해서 공유하는 뮤텍스를 생성하거나 이미 존재하는 뮤텍스를 여는 경우에는 bInitialOwner 매개변수를 FALSE로 지정하는 것이 안전하다. 이를 통해 CreateMutex의 호출로 리턴된 뮤텍스의 소유권에 대한 혼동을 없앨 수 있다. 현재 버전의 윈도우 95와 윈도우 NT에서* 지정된 뮤텍스가 이미 존재하는 경우에 이 매개변수를 무시하게 된다. 이것은 bInitialOwner 매개변수를 TRUE로 한다고 해도 CreateMutex에서 리턴한 후에 GetLastError 함수를 호출하면 ERROR_ALREADY_DEFINED를 리턴값으로 얻는다는 것을 의미한다. 뮤텍스에 대한 소유권을 얻지 못한 경우 뮤텍스의 소유권을 얻기 위해서는 앞에서 정의한 함수를 이용해서 뮤텍스를 대기하고 있어야 한다. 다음의 코드는 이름을 가진 뮤텍스를 생성할 때 bInitialOwner를 TRUE로 안전하게 지정하는 방법에 대해서 보여주고 있다.

```
//"TheMutex"라는 이름의 뮤텍스를 기본 보안 속성으로 생성한다.
// 그리고 뮤텍스가 생성된 다음 초기에 뮤텍스를 소유하겠다고 설정한다.
//
HANDLE hMutex = CreateMutex(NULL, TRUE /* bInitialOwner */,        "TheMutex");

if( NULL != hMutex )
{
    BOOL fMutexOwned = FALSE;

    // 실제로 뮤텍스를 가졌는지 확인한다.
    // 만약에 뮤텍스를 얻지 못했으면 얻게 될 때까지 대기한다.
    //
    if( GetLastError() == ERROR_ALREADY_EXISTS )
    {
        // 이미 뮤텍스가 존재하기 때문에 초기에 소유하는 옵션은 무시되었다.
        // 여기에서 다시 뮤텍스를 획득한다.
        //
        DWORD dwWaitResult = WaitForSingleObject(hMutex, INFINITE);

        fMutexOwned = WaitSucceeded(dwWaitResult, 1);
    }
    else
```

*역자주: 이 책이 나온 시점에는 윈도우 95와 윈도우 NT 4.0이 주요 운영체제로 사용되었다.

```
    {
        // 뮤텍스를 만드는 데 성공했다.
        // 그러므로 뮤텍스의 소유권까지 확보했다.
        //
        fMutexOwned = TRUE;
    }

    if( fMutexOwned )
    {
        . . . 뮤텍스에 의해서 공유되는 자원을 사용하는 코드 . . .

        // 뮤텍스의 소유권을 해제한다.
        //
        ReleaseMutex(hMutex);
    }

    // 뮤텍스 객체에 대한 참조를 닫는다.
    //
    CloseHandle(hMutex);
}
```

여기서 보듯이 CreateMutex의 리턴값은 GetLastError 함수를 사용해서 자세히 분석할 필요가 있다. 특히 bInitialOwner 플래그를 TRUE로 하고, 이름을 가진 뮤텍스 객체를 생성할 때에는 더욱 주의해야 한다.

프로세스가 다른 프로세스의 스레드에 의해서 이미 생성된 뮤텍스의 핸들을 확보하기를 원하거나, 만약 뮤텍스가 존재하지 않는 경우에 뮤텍스를 생성하지 않기를 원한다면 OpenMutex 함수를 사용할 수 있다.

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

매개변수	타입	설명
dwDesiredAccess	DWORD	lpName의 이름을 가진 뮤텍스를 가지려는 스레드의 접근 형태를 나타내는 비트 플래그. 뮤텍스의 모든 권한을 사용하기 위해 MUTEX_ALL_ACCESS를 사용한다. 뮤텍스를 단순히 대기하고 있는 스레드에게 핸들을 넘겨주는 것으로만 사용한다면 SYNCHRONIZE(윈도우 NT에만 해당)를 사용한다. 보안을 지원하는 시스템의 경우 요구한 접근 권한이 호출하는 프로세스의 보안성을 넘는 경우에는 OpenMutex 함수의 호출이 실패한다.

매개변수	타입	설명
bInheritHandle	BOOL	이함수를호출하는프로세스가 CreateProcess 함수를사용한경우 핸들이새로생성된프로세스에상속가능한지를나타낸다 . TRUE 는리턴된 핸들이상속가능하다는것을나타내고 , FALSE는리턴된 핸들이상속불가능하다는것을나타낸다 .
lpName	LPCTSTR	열려고하는뮤텍스의이름을나타내는종료된 널문자열의포인터 . 이값은 NULL로지정할수없다 .

이 함수는 다른 프로세스가lpName 매개변수로 지정한 같은 이름의 뮤텍스를 이미 생성하였고, dwDesiredAccess 플래그의 조건이 만족되는 경우에만 호출에 성공한다. 이 함수가 주어진 뮤텍스를 여는 데 성공했다면 뮤텍스 객체의 핸들이 리턴된다. 이 함수는 뮤텍스의 소유권을 확보하려고 하지 않는다. NULL이 리턴된 경우는 실패한 경우이며, GetLastError 함수를 사용해서 실패한 이유에 대한 좀더 자세한 정보를 얻을 수 있다. 뮤텍스 객체의 핸들을 닫으려면CloseHandle 함수를 사용한다.

앞에서 나열했던 여러 커널 객체와 마찬가지로 뮤텍스는 두 가지 상태(시그널되거나 시그널되지 않은 상태)를 갖는다. 뮤텍스를 소유하고 있는 스레드가 없을 경우에 그 스레드는 시그널된 상태이다. 스레드는 앞에서 설명한 대기 오퍼레이션을 이용해서 뮤텍스를 획득할 수 있다. 또 하나의 스레드는 뮤텍스를 해제하지 않고도 반복해서 뮤텍스를 획득할 수 있다. 하지만 아마도 이런 방법으로 애플리케이션을 설계하는 일은 거의 없을 것이다. 이런 기능을 지원하는 것은 스레드 자신에 의해서 교착 상태에 빠지는 것을 방지하기 위한 경우일 뿐이다. 내부적으로 뮤텍스를 갖고 있는 함수가 서로를 호출하면 이런 상황이 발생할 수 있다.

스레드가 뮤텍스를 해제하기 전에 같은 뮤텍스를 계속해서 획득하면 뮤텍스를 획득한 만큼 해제하는 작업을 해야 한다. WaitForSingleObject(또는 WaitForMultipleObjects) 함수를 사용할 때마다 ReleaseMutex를 사용한다면 이런 문제를 간단히 해결할 수 있다. 또한 세마포어와는 달리 뮤텍스를 소유하고 있는 스레드만이 뮤텍스의 소유권을 해제할 수 있다. 스레드가 뮤텍스를 가지고 있는 동안 이 뮤텍스를 획득하려는 다른 스레드는 블록당하며, 이 스레드가 뮤텍스를 해제해야만 블록 상태에서 빠져 나오게 된다.

뮤텍스의 소유권은ReleaseMutex 함수의 호출에 의해서 해제된다.

```
BOOL ReleaseMutex( HANDLE    hMutex );
```

매개변수	타입	설명
hMutex	HANDLE	뮤텍스객체의핸들

이 함수의 호출이 성공하면 TRUE가 리턴되고, 실패하면 FALSE가 리턴된다. 물론 GetLastError 함수를 이용하면 더 자세한 정보를 얻을 수 있다. ReleaseMutex 함수는 뮤텍스를 획득할 때마다 호출되어야 하며, bInitialOwner의 값을 TRUE로 하고 CreateMutex 함수를 호출한 경우에도 호출되어야 한다. 뮤텍스를 가진 스레드가 뮤텍스를 해제하기 전에 종료하거나 종료당하면 그 뮤텍스를 대기하던 다른 스레드 중 하나가 대기 함수에서 리턴하게 되며, 실패 코드로 WAIT_ABANDONED_0 값을 갖거나 다수의 객체를 대기하던 경우에는 (WAIT_ABANDONED_0)에서 (WAIT_ABANDONED_0 + nCount -1) 사이의 값을 갖는다. 그 스레드는 이제 뮤텍스의 소유자가 되며, WAIT_OBJECT_0 또는 WAIT_OBJECT_0 + nCount -1의 값을 리턴한 것처럼 ReleaseMutex 함수를 호출해야 한다. 이 사실은 앞에서 설명한 WaitSucceeded와 WaitAbandoned 함수 모두가 뮤텍스를 소유하고 있고, 해제하는 작업이 필요하다는 것을 의미한다. 대부분의 경우 포기된 뮤텍스는 스레드 프로그래밍의 어딘가에 논리적 버그가 있다는 것을 의미하기 때문에 최대한 빨리 이 문제를 찾아서 해결해주어야 한다.

[예제 4-1] DuelingThreads.cpp는 뮤텍스의 사용 방법과 WaitForSingleObject와 WaitForMultipleObjects 함수의 사용 방법을 보여주는 예제이다. 주 스레드는 뮤텍스와 세 개의 자식 스레드를 생성하고, 뮤텍스의 핸들은 자식 스레드에게 LPVOID 매개변수로 스레드 프로시저에 전달된다. 이 매개변수는 사용하기 전에 HANDLE 형태로 캐스팅하여 사용해야 한다. 자식 스레드는 WaitForSingleObject를 호출해서 뮤텍스를 대기하다가 획득한다. 뮤텍스는 0.5초간 소유되며 그 후에 해제된다. 따라서 자식 스레드는 뮤텍스를 획득하려고 다시 시도하기 전에 0.5초 동안 대기하게 되고, 각 자식 스레드는 이런 작업을 세 번씩 반복한다. 주 스레드는 자식 스레드가 종료한 후에 정리 작업(cleanup)을 하고 종료한다.

[예제4-1] DuelingThread1.cpp: 뮤텍스의 사용을 나타내는 간단한 예제

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

unsigned _stdcall ChildThreadProcedure( LPVOID lpMutex) {
      // 매개변수를 뮤텍스 핸들로 변환한다.
      HANDLE hMutex = (HANDLE) lpMutex;
      // 이 스레드의 ID를 저장한다.
      DWORD dwThreadId = GetCurrentThreadId();
      // 세 번 반복한다.
      for (int n =0; n < 3; n++) {
            // 뮤텍스를 획득하기 위해 대기한다.
            DWORD dwResult = WaitForSingleObject( hMutex, INFINITE);
```

```
            if (dwResult == WAIT_OBJECT_0) {
                printf("Thread 0x%08x - acquired the mutex.\n", dwThreadId);
                // 뮤텍스를 0.5초 동안 잡고 있는다.
                Sleep(500);

                printf("Thread 0x%08x - releasing the mutex.\n", dwThreadId);
                ReleaseMutex(hMutex);
                // 다시 시도하기 전에 0.5초 동안 대기한다.
                Sleep(500);
            }
            else {
                printf("Thread 0x%08x - error calling WaitForSingleObject().\n",
                        dwThreadId);
                return 0xFFFFFFFF;
            }
        }

        return 0;
}

int main(void) {
        HANDLE hChildThread[3];
        HANDLE hMutex;
        // 뮤텍스를 생성한다.
        hMutex = CreateMutex( NULL, FALSE, NULL);
        if (hMutex == NULL) {
            printf("Primary thread - error calling CreateMutex().\n");
            exit(0xFFFFFFFF);
        }

        // 자식 스레드를 생성한다.
        unsigned uUnusedThreadId;
        hChildThread[0] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                            (void*) hMutex, 0, &uUnusedThreadId);
        hChildThread[1] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                            (void*) hMutex, 0, &uUnusedThreadId);
        hChildThread[2] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                            (void*) hMutex, 0, &uUnusedThreadId);
        if (!hChildThread[0] || !hChildThread[1] || !hChildThread[2]) {
            printf("Primary thread - error creating child threads.\n");
            exit(0xFFFFFFFF);
        }

        // 자식 스레드가 모두 종료할 때까지 대기한다.
        DWORD dwResult = WaitForMultipleObjects( 3, hChildThread, TRUE, INFINITE);
```

```
    if (dwResult != WAIT_OBJECT_0) {
        printf( " Primary thread - error calling WaitForMultipleObjects().\n        ");
        exit(0xFFFFFFF);
    }

    // 모든 핸들을 정리한다.
    CloseHandle(hMutex);
    CloseHandle(hChildThread[0]);
    CloseHandle(hChildThread[1]);
    CloseHandle(hChildThread[2]);

    return 0;
}
```

[예제 4-1]을 컴파일하고 실행시키면 [예제 4-2]와 같은 결과를 얻을 수 있다. 한번에 하나의 스레드만이 뮤텍스를 소유한다는 것을 주목해야 한다. 다른 스레드는 이 스레드가 뮤텍스를 해제할 때까지 WaitForSingleObject 함수 안에서 계속 대기해야 한다.

[예제4-2] [예제 4-1]의 실행 결과로 가능한 한가지 예

```
Thread 0xfffac9fb - acquired the mutex.
Thread 0xfffac9fb - releasing the mutex.
Thread 0xfffface23 - acquired the mutex.
Thread 0xfffface23 - releasing the mutex.
Thread 0xfffacc4b - acquired the mutex.
Thread 0xfffacc4b - releasing the mutex.
Thread 0xfffac9fb - acquired the mutex.
Thread 0xfffac9fb - releasing the mutex.
Thread 0xfffface23 - acquired the mutex.
Thread 0xfffface23 - releasing the mutex.
Thread 0xfffacc4b - acquired the mutex.
Thread 0xfffacc4b - releasing the mutex.
Thread 0xfffface23 - acquired the mutex.
Thread 0xfffface23 - releasing the mutex.
Thread 0xfffac9fb - acquired the mutex.
Thread 0xfffac9fb - releasing the mutex.
Thread 0xfffacc4b - acquired the mutex.
Thread 0xfffacc4b - releasing the mutex.
```

세마포어

세마포어는 수치적인 카운트를 갖고 있는 커널 객체이다. 세마포어 객체는 카운트의 값이 0인 경우에 시그널되지 않은 상태로 있고, 카운터 값이 0보다 크면 시그널되며, 카운터 값이 0보다 큰 동안에는 계속 시그널된 상태로 있는다. 대개의 경우 세마포어는 제한된 수의 자원을 관리하기 위해서 임계 영역이나 뮤텍스와 함께 사용된다. 예를 들어, 서커스의 회전목마(merry-go-round)를 모델링하는 프로그램을 설계하는 경우에 탈 수 있는 말의 개수를 초기값으로 설정한 세마포어 객체를 생성하여 사용할 수 있다. 말을 타기 위해서 기다리는 사람들은 먼저 세마포어 객체에서 기다리고 있어야 한다는 전제 조건을 만족시켜야 한다(말하자면, 회전문 앞에서 줄을 서서 기다리고 있는 것처럼). 빈 자리가 만들어질 때마다 세마포어에 기다리고 있던 사람들은 말을 탈 수 있게 된다. 물론 두 사람이 말을 탈 준비가 된 경우에는 같은 말을 타기 위해서 경쟁하게 된다. 이런 경우에는 뮤텍스나 임계 영역을 사용하여 말을 타는 순서를 조정해줌으로써 두 사람이 말을 타는 데 분쟁이 생기는 것을 피할 수 있다.

세마포어를 생성하기 위해서는 CreateSemaphore 함수가 사용된다.

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

매개변수	타입	설명
lpSemaphoreAttributes	LPSECURITY_ ATTRIBUTES	세마포어객체의보안속성을나타내는 SECURITY_ ATTRIBUTES 구조체의포인터. 이값이 NULL이면세마포어는기본보안속성의로생성되고 상속할수없다.
lInitialCount	LONG	세마포어의초기카운트값을지정한다. 이값은 0 이상이고, lMaximumCount의값이하여야한다. 이값이 0이면세마포어는초기에시그널되지않은상태로존재하고, 이값이 0보다크면세마포어는초기에시그널된상태로존재한다.
lMaximumCount	LONG	세마포어의최대카운트값을나타낸다. 이매개변수의값은 lInitialCount의값이상의크기를갖는다.
lpName	LPCTSTR	세마포어의이름을나타내는종료된 널문자열의포인터. 이름이없는세마포어를생성하려면이값을 NULL로한다. 이름을가진세마포어는다른프로세스안에있는스레드에의해서열리고사용될수있다. 이름이없는세마포어는세마포어를생성한프로세스안의스레드에게만보이게된다.

CreateSemaphore 함수가 호출에 성공하면 세마포어 객체의 핸들이 리턴된다. lpName이 NULL이 아니고 세마포어와 같은 이름을 갖는 객체가 이미 존재한다면 이미 존재하는 세마포어의 핸들이 리턴되고, GetLastError는 ERROR_ALREADY_EXISTS를 리턴한다. 뮤텍스, 세마포어, 이벤트, 파일 매핑은 모두 같은 이름 공간을 사용한다는 사실에 주의해야 한다. 만약 lpName이 NULL이 아니고 세마포어가 아닌 다른 객체와 같은 이름을 갖는 경우에는 이 함수의 호출은 실패하고, GetLastError는 ERROR_INVALID_HANDLE을 리턴하게 된다.

세마포어 객체의 핸들을 닫기 위해서는 CloseHandle 함수를 사용한다. 세마포어 객체에 대한 마지막 핸들을 가진 스레드가 핸들을 닫게 되면, 세마포어 객체는 없어지게 된다.

같은 프로세스나 다른 프로세스에 있는 스레드에 의해서 이미 생성된 세마포어에 대한 핸들을 얻고자 할 때나 세마포어가 존재하지 않아도 세마포어를 생성하고 싶지 않을 때에는 OpenSemaphore 함수를 사용한다.

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

매개변수	타입	설명
dwDesiredAccess	DWORD	lpName의 이름을 가진 세마포어를 사용하려는 스레드가 세마포어에 적용하려고 하는 접근권한의 형태를 나타내는 비트맵 플래그. [표 4-2]에서 제시한 하나 또는 그 이상의 값을 가질 수 있다. 보안을 지원하는 시스템에서 주어진 세마포어에 대한 보안 해석기가 요구된 액세스를 갖는 프로세스의 호출을 허용하지 않는다면 OpenSemaphore 함수는 실패한다.
bInheritHandle	BOOL	CreateProcess 함수를 사용해서 프로세스를 생성한 경우 그 프로세스에 핸들을 상속할지의 여부를 나타내는 플래그. 이 값이 TRUE이면 핸들이 상속 가능함을 나타내고, FALSE이면 핸들은 상속 가능하지 않음을 나타낸다.
lpName	LPCTSTR	열려고 하는 세마포어의 이름을 나타내는 종료된 널 문자열의 포인터. 이 값은 NULL이 될 수 없다.

이 함수는 다른 프로세스가 이미 같은 이름의 세마포어를 생성했고, dwDesiredAccess 플래그가 만족되는 경우에만 호출에 성공한다. 이 함수가 주어진 세마포어를 여는데 성공하면 세마포어 객체의 핸들이 리턴된다. NULL이 리턴되면 함수의 호출이 실패했음을 나타내며, GetLastError를 사용하면 자세한 정보를 얻을 수 있다. 세마포어 객체의 핸들을 닫으려면 CloseHandle 함수를 사용한다.

[표 4-2] 세마포어객체의접근 권한

상수	설명
SEMAPHORE_ALL_ACCESS	주어진세마포어에대해가능한모든접근권한을요구
SEMAPHORE_MODIFY_STATE	ReleaseSemaphore 함수에서세마포어핸들을사용할수있도록 한다.
SYNCHRONIZE	리턴된핸들을대기오퍼레이션에넘겨주기 위한목적으로세마포어를연다(윈도우 NT에만해당).

세마포어가 시그널되기를 대기하고 있는 스레드는 세마포어의 카운트 값이 0 이상이 되고 세마포어가 시그널될 때까지 블록당한다. 세마포어가 시그널되면 대기하고 있던 스레드는 계속해서 수행하게 되며, 운영체제는 세마포어의 값을 하나만큼 감소시킨다. 세마포어의 값이 0 이상인 동안에는 스레드는 계속해서 수행하고, 세마포어의 값은 계속해서 감소한다. 카운트의 값이 0이 되면 세마포어는 시그널되지 않은 상태로 되고 대기오퍼레이션을 하는 스레드는 블록당한다. 세마포어의 카운트 값을 증가시키기 위해서는 ReleaseSemaphore 함수가 사용된다.

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

매개변수	타입	설명
hSemaphore	HANDLE	세마포어객체의핸들
lReleaseCount	LONG	증가시킬세마포어카운트값을나타내며, 이값은반드시 0 이상이어야한다. 세마포어를 lReleaseCount만큼증가시킨결과가 CreateSemaphore에서지정한최대카운트의값보다커지면세마포어의 카운트값은원래값을갖게되고이함수의호출은실패한다 .
lpPreviousCount	LPLONG	증가시키기전의카운트값을넣을 LONG 포인터. lpPreriousCount의값을알고싶지않으면 NULL을넣는다.

이 함수는 호출에 성공하면 TRUE를 리턴하고 실패하면 FALSE를 리턴하며, GetLastError를 이용하면 실패한 이유에 대한 자세한 정보를 얻을 수 있다. 뮤텍스 객체와 달리 세마포어는‘ 소유할 수 있는 ’객체가 아니다. 이것은 세마포어의 핸들을 갖고 있는 프로세스의 모든 스레드는 세마포어 객체를 시그널하기 위해서 ReleaseSemaphore 함수를 호출할 수 있다는 사실을 의미한다.

다음은 간단한 세마포어의 사용 예제이다. 서버가 최대 네 개의 동시 접속을 허용한다고 하자. 물론 항상 모든 스레드가 접속을 시도할 수는 있다. WaitForConnection과 ReleaseConnection 함수를 만들어서 접속 가능한 수를 4로 제한하고, 다른 스레드를 접속 가능할 때까지 대기하도록 할 수 있다.

```
// 최대 접속 가능 횟수로 세마포어의 카운트를 초기화한다.
HANDLE hConnectionSem = CreateSemaphore( NULL, 4, 4, NULL);
.
.
.
BOOLEAN WaitForConnection(void) {
      if (WAIT_OBJECT_0 == WaitForSingleObject( hConnectionSem, INFINITE) )
          return TRUE;
      else
          return FALSE;
}

BOOLEAN ReleaseConnection(void) {
      LONG lPreviousCount;
      ReleaseSempahore( hConnectionSem, 1, &lPreviousCount);
}
```

서버를 사용하는 모든 스레드는 연결할 때 먼저 WaitForConnection 함수를 호출해야 한다. 또한 스레드는 연결을 끊을 때에도 ReleaseConnection 함수를 호출해야 한다. 이런 함수의 세마포어를 사용해서 어떤 수의 연결 시도가 발생해도 연결 수를 네 개로 조절할 수 있다.

[예제 4-3]은 뮤텍스를 사용한 DuelingThreads1.cpp를 변형한 코드이다. 이 프로그램에서 세마포어는 세 개의 스레드 중 두 개의 스레드가 동시에 작동할 수 있도록 허용한다. 뮤텍스 대신 세마포어를 사용하는 것을 제외하면, 이 코드는 [예제 4-1]과 동일하다.

[예제4-3] DuelingThreads2.cpp: 세마포어의 사용법을나타내는 간단한 예제

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

unsigned _stdcall ChildThreadProcedure( LPVOID lpSemaphore) {
    // 매개변수를 세마포어 핸들로 변환한다.
    HANDLE hSemaphore = (HANDLE) lpSemaphore;

    // 이 스레드의 ID를 저장한다.
    DWORD dwThreadId = GetCurrentThreadId();
```

```
    // 세 번 반복한다.
    for (int n =0; n < 3; n++) {
        // 세마포어를 획득하기 위해 대기한다.
        DWORD dwResult = WaitForSingleObject( hSemaphore, INFINITE);

        if (dwResult == WAIT_OBJECT_0) {
            printf( "Thread 0x%08x - acquired the semaphore.\n      ", dwThreadId);
            // 0.5초 동안 세마포어를 갖고 있는다.
            Sleep(500);

            printf( "Thread 0x%08x - releasing the semaphore.\n      ", dwThreadId);
            LONG lPreviousCount;
            ReleaseSemaphore( hSemaphore, 1, &lPreviousCount);

            // 세마포어를 다시 시도하기 전에 0.5초 동안 대기한다.
            Sleep(500);
        }
        else {
            printf( "Thread 0x%08x - error calling WaitForSingleObject().\n        ",
                dwThreadId);
            return 0xFFFFFFFF;
        }
    }

    return 0;
}

int main(void) {
    HANDLE hChildThread[3];
    HANDLE hSemaphore;

    // 세마포어를 생성한다.
    hSemaphore = CreateSemaphore( NULL, 2, 2, NULL);
    if (hSemaphore == NULL) {
        printf( "Primary thread - error calling CreateSemaphore().\n        ");
        exit(0xFFFFFFFF);
    }

    // 자식 스레드를 생성한다.
    unsigned uUnusedThreadId;
    hChildThread[0] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                        (void*) hSemaphore, 0, &uUnusedThreadId);
    hChildThread[1] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                        (void*) hSemaphore, 0, &uUnusedThreadId);
```

```
    hChildThread[2] = (HANDLE)_beginthreadex( NULL, 0, ChildThreadProcedure,
                        (void*) hSemaphore, 0, &uUnusedThreadId);
    if (!hChildThread[0] || !hChildThread[1] || !hChildThread[2]) {
          printf( " Primary thread - error creating child threads.\n       ");
          exit(0xFFFFFFFF);
     }
    // 자식 스레드가 종료할 때까지 대기한다.
    DWORD dwResult = WaitForMultipleObjects( 3, hChildThread, TRUE, INFINITE);
    if (dwResult != WAIT_OBJECT_0) {
          printf( " Primary thread - error calling WaitForMultipleObjects().\n        ");
          exit(0xFFFFFFFF);
    }

    // 핸들을 모두 정리한다.
    CloseHandle(hSemaphore);
    CloseHandle(hChildThread[0]);
    CloseHandle(hChildThread[1]);
    CloseHandle(hChildThread[2]);

    return 0;
}
```

[예제 4-3]을 컴파일하고 실행시키면 [예제 4-4]와 같은 결과를 얻을 수 있다. 한번에 두 개의 스레드까지 세마포어를 획득할 수 있다는 사실을 주목해야 한다. 이 예제에는 세 개의 스레드가 사용되기 때문에 적어도 하나의 스레드는 다른 두 개의 스레드가 동작하는 동안 대기하고 있어야 한다.

[예제4-4] [예제 4-3]의 실행 결과로 가능한 한가지 예

```
Thread 0xfffb2f7b - acquired the semaphore.
Thread 0xfffb2da3 - acquired the semaphore.
Thread 0xfffb2f7b - releasing the semaphore.
Thread 0xfffb2da3 - releasing the semaphore.
Thread 0xfffb13cb - acquired the semaphore.
Thread 0xfffb13cb - releasing the semaphore.
Thread 0xfffb2da3 - acquired the semaphore.
Thread 0xfffb2f7b - acquired the semaphore.
Thread 0xfffb2f7b - releasing the semaphore.
Thread 0xfffb2da3 - releasing the semaphore.
Thread 0xfffb13cb - acquired the semaphore.
Thread 0xfffb13cb - releasing the semaphore.
Thread 0xfffb2da3 - acquired the semaphore.
Thread 0xfffb2f7b - acquired the semaphore.
Thread 0xfffb2f7b - releasing the semaphore.
```

```
Thread 0xfffb2da3 - releasing the semaphore.
Thread 0xfffb13cb - acquired the semaphore.
Thread 0xfffb13cb - releasing the semaphore.
```

이벤트

Win32 이벤트 객체는 가장 일반적인 형태의 스레드간 그리고 프로세스간 동기화 객체이다. 뮤텍스나 세마포어는 운영체제가 미리 만들어 놓은 시그널되는 방법과 시그널되지 않는 방법이 정해져 있지만, 이벤트 객체는 프로그래머가 지정한 의미 구조(semantic)를 사용한다. 일반적으로 조건부 변수(condition variable)라고 부르는 이벤트 객체는 매우 강력한 기능을 갖는 스레드 동기화 도구이다. 이벤트 객체의 기본적인 개념은 조건을 만족시킬 때까지 시그널되지 않은 상태로 대기한다는 것이며, 프로그램이 시그널을 주게 되면 시그널된 상태로 변한다. 따라서 프로그래머는 뮤텍스나 세마포어보다는 훨씬 유연하게 동기화 작업에 이벤트를 사용할 수 있다. 예를 들어, WaitForMultipleObjects와 같은 경우 수동 리셋 이벤트(hThreadExitEvent)가 일꾼 스레드에게 종료할 시점을 알리는 데 사용된 경우를 생각할 수 있다. 이런 방법을 통해서 자원 뮤텍스(hResourceMutex)가 다른 스레드에 의해 소유되는 동안 일꾼 스레드를 대기시켜서 CPU의 활용도를 최대로 높일 수 있다. 애플리케이션이 종료하면 주 스레드는 단지 이 스레드에게 종료 이벤트를 시그널시켜 주면 된다. 이렇게 하면 일꾼 스레드가 WaitForMultipleObjects에서 리턴하게 되고 정리 작업을 한 후에 종료한다.

이벤트에는 기본적으로 수동 리셋(manual-reset)과 자동 리셋(auto-reset)의 두 가지 형태가 있다. 이 두 가지 형태의 차이점은 스레드에 의해서 대기되고 시그널된 후에 이벤트 객체의 동작 방식에 있다. 자동 리셋 이벤트는 대기하는 작업을 끝낸 후에 시그널되지 않은 상태로 다시 변하고, 수동 리셋 이벤트는 프로그래머에 의해서 리셋되어야만 시그널되지 않은 상태로 변한다. 각각의 이벤트는 경우에 따라서 유용하게 사용될 수 있다.

이벤트는CreateEvent 함수를이용해서 생성하며, 이함수는 다음과같은 원형을갖는다.

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

매개변수	타입	설명
lpEventAttributes	LPSECURITY_ ATTRIBUTES	이벤트객체가갖는보안속성을나타내는 SECURITY_ ATTRIBUTES 구조체의포인터. 이값이 NULL이면이벤트는 기본보안속성으로생성된다.
bManualReset	BOOL	수동리셋이벤트객체를만들것인지자동리셋이벤트객체를 만들것인지를나타내는플래그. 자동리셋이벤트는대기하던 작업을마친후에시그널되지않은상태에서자동으로리셋된 다. 수동리셋이벤트는프로그램에의해서리셋될때까지시그 널된상태로남아있는다. TRUE로값을지정하면수동리셋이 벤트를생성하고, FALSE로값을지정하면자동리셋이벤트를 생성한다.
bInitialState	BOOL	이벤트객체의초기상태를지정하는플래그. TRUE를지정하 면이벤트를시그널된상태로초기화하고, FALSE로지정하면 시그널되지않은상태로초기화한다.
lpName	LPCTSTR	이벤트객체의이름을나타내는종료된 널문자열의포인터. 이 름을가진이벤트객체는다른프로세스의스레드에의해서열 릴수있다. 이름이없는이벤트객체는이벤트객체를생성한 프로세스안의스레드에서만사용할수있다. NULL을넘겨주면 이름이없는이벤트객체를생성한다.

이 함수의 호출이 성공하면 이벤트 객체의 핸들이 리턴된다. 이름을 가진 이벤트 객체를 지정했고 같은 이름의 이벤트 객체가 이미 존재하는 경우에는 이벤트 객체의 핸들을 리턴하고, GetLastError를 호출하면 ERROR_ALREADY_EXISTS를 리턴한다. 이 함수가 호출에 실패하면 NULL을 리턴하고, GetLastError를 사용하면 자세한 정보를 얻을 수 있다. 이 함수를 호출했을 때, 이름을 가진 이벤트를 지정했고 같은 이름의 이벤트 객체가 이미 존재하는 경우에는 bInitialState 매개변수는 무시된다. 이벤트 객체의 핸들을 닫기 위해서는 CloseHandle 함수를 사용한다.

같은 프로세스나 다른 프로세스의 스레드에 의해서 생성된 이벤트 객체의 핸들을 얻으려 하고 이벤트 객체가 존재하지 않는 경우에 이벤트 객체를 생성하고 싶지 않을 때에는 OpenEvent 함수를 사용한다.

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

매개변수	타입	설명
dwDesiredAccess	DWORD	lpName값을갖는이벤트를소유하려는스레드의접근형태를지정하 는비트 필드플래그. 이값은 [표 4-3]의값중에서하나이상의값을 가질수있다. 보안을지원하는시스템에서는적용하려는권한이함수 를호출하는프로세스의권한보다높은경우함수를호출할수없다.

매개변수	타입	설명
bInheritHandle	BOOL	CreateProcess 함수를호출해서생성한프로세스에게핸들을상속할지의여부를나타내는플래그. TRUE를지정하면상속가능하게되고, FALSE를지정하면상속가능하지않게된다.
lpName	LPCTSTR	열려고하는이벤트의이름을나타내는종료된 널문자열의포인터. 이값은 NULL이될수없다.

이 함수는 다른 프로세스가 lpName과 같은 이름으로 이벤트 객체를 생성했고, dwDesiredAccess 플래그가 만족되는 경우에만 호출할 수 있다. 이 함수가 주어진 이벤트를 여는 데 성공하면 이벤트 객체의 핸들이 리턴된다. NULL이 리턴되는 경우에는 이 함수의 호출이 실패한 경우이고, GetLastError를 사용하면 좀더 자세한 정보를 얻을 수 있다. 핸들을 닫기 위해서는 CloseHandle 함수를 사용한다.

[표 4-3] 이벤트접근성을나타내는상수

상수	설명
EVENT_ALL_ACCESS	주어진이벤트에적용할수있는모든권한을요구
EVENT_MODIFY_STATE	SetEvent, ResetEvent, PulseEvent 함수에서의이벤트핸들의사용을가능하도록한다.
SYNCHRONIZE	대기오퍼레이션에핸들을넘겨줄목적으로만사용되는이벤트를연다 (윈도우 NT에만해당).

이벤트 객체를 시그널된 상태로 만들기 위해서는 SetEvent 함수가 사용된다.

```
BOOL SetEvent( HANDLE    hEvent );
```

매개변수	타입	설명
hEvent	HANDLE	이벤트객체의핸들

이 함수는 호출에 성공하면 TRUE를 리턴하고, 실패하면 FALSE를 리턴한다. SetEvent 함수를 호출한 결과는 어떤 형태의 이벤트가 시그널됐는가에 따라 다르다. 자동 리셋 이벤트가 호출된 경우에는 이 함수는 대기하고 있는 스레드가 해제될 때까지 이벤트를 시그널된 상태로 만든다. 다시 말하면 이 함수가 호출되었을 때 특정 이벤트를 대기하고 있던 스레드가 있었다면 그 중 하나가 해제되고, 이벤트는 그와 동시에 리셋되어 시그널되지 않은 상태로 변한다. 이 함수가 호출되었을 때 이벤트를 대기하고 있는 스레드가 하나도 없었다면 이벤트는 이 이벤트를 대기하는 스레드가 생길 때까지 시그널된 상태로 남

게 된다. 이런 스레드가 생기게 되면 이 이벤트는 운영체제에 의해서 자동적으로 시그널되지 않은 상태로 변하게 된다. 수동 리셋 이벤트에 이 함수가 호출되었을 때 이 이벤트를 대기하고 있는 모든 스레드가 대기 상태에서 해제되며, 이벤트 객체는 ResetEvent 함수를 이용해서 명시적으로 시그널되지 않은 상태가 될 때까지 시그널된 상태로 남는다.

ResetEvent 함수는 다음과 같은 원형을 갖는다.

```
BOOL ResetEvent( HANDLE    hEvent  );
```

매개변수	형태	설명
hEvent	HANDLE	이벤트객체의핸들

이 함수는 호출에 성공하면 TRUE를 리턴하고, 호출에 실패하면 FALSE를 리턴한다. GetLastError 함수를 이용하면 호출에 실패한 자세한 정보를 얻을 수 있다. 이 함수는 일반적으로 수동 리셋 이벤트와 함께 사용된다. 왜냐하면 자동 리셋 이벤트는 운영체제에 의해서 자동으로 리셋되어 시그널되지 않은 상태로 변하기 때문이다. 대기하고 있는 스레드가 없었다면 자동 리셋 이벤트에 ResetEvent 함수를 사용해서 이벤트를 시그널되지 않은 상태로 만들 수도 있지만, 이런 식으로 코딩하는 것은 매우 위험하다. 왜냐하면 이런 식의 코딩은 자동 리셋 이벤트 객체의 일반적인 동작 형태를 간섭하는 것이고, 언젠가 이 코드를 관리하는 사람이 큰 곤경에 빠질 가능성이 높기 때문이다.

Win32는 이벤트 객체의 상태를 변경하기 위해서 SetEvent와 ResetEvent 함수의 조합된 형태를 지원한다. PulseEvent 함수는 같은 이벤트를 대기하고 있는 일련의 스레드를 깨우는 데 사용할 수 있다.

```
BOOL PulseEvent( HANDLE    hEvent  );
```

매개변수	타입	설명
hEvent	HANDLE	이벤트객체의핸들

PulseEvent는 호출에 성공하면 TRUE를 리턴하고, 실패하면 FALSE를 리턴한다. GetLastError를 사용하면 호출에 실패한 이유에 대한 자세한 정보를 얻을 수 있다. 이 이벤트를 대기하고 있는 스레드가 없는 경우에 PulseEvent가 호출되면 이벤트를 리셋하여 시그널되지 않은 상태로 만든 후 리턴한다. 수동 리셋 이벤트 객체가 호출되면, PulseEvent 함수는 이 함수가 호출되었을 때 그 이벤트 객체를 대기하고 있던 모든 스레드를 깨우고 자동으로 이벤트를 시그널되지 않은 상태로 리셋한다. 자동 리셋 이벤트가 호출되면, PulseEvent 함수는 자동 리셋 이벤트 객체가 호출되었을 때의 SetEvent 함수처

럼 동일하게 동작한다. 즉 여러 개의 스레드가 대기하더라도 하나의 스레드만을 깨우고 자동으로 시그널되지 않은 상태로 이벤트를 리셋한다.

PulseEvent 함수에 대해서 한 가지 더 주의해야 할 점이 있다. WaitForMulitpleObjects 함수의 호출을 대기하고 있는 여러 개의 객체 중에서 한 스레드에 의해 대기하고 있는 이벤트가 이 함수를 호출하면 이 이벤트는 시그널되었다가 대기 중인 스레드를 깨우지 않고 다시 시그널되지 않은 상태로 변하며, 대기 오퍼레이션에서 리턴한다. 대기 중인 스레드가 모든 커널 객체가 시그널되기(bWaitAll이 TRUE)를 대기하고 있고 다른 객체들이 PulseEvent 함수가 호출된 시점에서 시그널된 상태가 아니었다면 PulseEvent 함수에 의해서 다수의 객체를 대기하는 작업은 실패하고, PulseEvent는 대기 중인 스레드를 깨우지 않고 주어진 이벤트를 시그널되지 않은 상태로 리셋한다. PulseEvent를 호출한 시점에서 대기하고 있던 다른 모든 객체가 시그널된 상태였다면 이벤트를 시그널한 결과로 다수의 객체를 대기하는 동작이 성공하고 대기 중이던 스레드는 계속해서 수행할 수 있다.

[표 4-4]는 객체의 상태가 SetEvent, ResetEvent, PulseEvent에 의해서 변경되었을 때 수동 리셋 이벤트와 자동 리셋 이벤트 객체의 행동 양식을 요약한 것이다.

[표 4-4] 수동 리셋 이벤트와 자동리셋 이벤트 객체의 행동양식

동작	수동 리셋 이벤트	자동 리셋 이벤트
SetEvent	이벤트를시그널된상태로만들고 ResetEvent가호출될때까지이벤트를시그널된상태로둔다. ResetEvent가호출될때까지대기중인모든스레드를해제시킨다.	이벤트를시그널된상태로만들고대기하고있던스레드중하나가해제될때까지이벤트를시그널된상태로둔다. 한스레드가대기상태에서해제되면이벤트는자동적으로시그널되지않은상태가된다.
ResetEvent	이벤트를시그널되지않은상태로리셋한다.	이벤트를시그널되지않은상태로리셋한다. 운영체제가자동으로이벤트를리셋하기때문에이함수는일반적으로자동리셋이벤트에는사용되지않는다.
PulseEvent	이벤트를시그널된상태로만들고, 이함수가호출될때이벤트를대기하고있던모든스레드를해제하고 이벤트를시그널되지않은 상태로리셋한다. 이함수가호출될때 이벤트를대기하는스레드가없으면 이벤트는시그널되지않은상태로 리셋된다.	이벤트를시그널된상태로만들고, 이함수가호출될때대기하고있던스레드중의 하나를 대기 상태에서해제하고이벤트를 시그널되지않은상태로리셋한다. 대기하고있던스레드가 없는경우 이벤트는시그널되지않은상태로 리셋된다.

이 장을 끝내면서 수동 리셋 이벤트와 자동 리셋 이벤트의 동작 방식을 알 수 있는 작은

프로그램을하나 소개하려고 한다. [예제4-5]는 프로그램EventDemo.cpp의 예제이다.

[예제4-5] EventDemo.cpp: 이벤트 커널 객체의 데모 프로그램

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

// 주 스레드와 자식 스레드 사이에 공유되는 이벤트 핸들을
// 저장하기 위한 전역 변수
HANDLE g_hAutoResetEvent = NULL;
HANDLE g_hManualResetEvent = NULL;

// 자식 스레드들의 프로시저
unsigned __stdcall ThreadStartFunc( void *lpThreadParameterId) {
    DWORD dwStatus;

    // 스레드가 시작되었음을 사용자에게 알린다.
    printf( " Thread #%d id=0x%08x, started.\n     ",
            (int) lpThreadParameterId,
            GetCurrentThreadId());

    // 자동 리셋 이벤트를 대기한다.
    printf( " Thread #%d id=0x%08x, waiting for auto-reset event.\n        ",
            (int) lpThreadParameterId,
            GetCurrentThreadId());
    dwStatus = WaitForSingleObject( g_hAutoResetEvent, INFINITE);
    if (dwStatus == WAIT_OBJECT_0) {
        printf( " Thread #%d id=0x%08x, wait for auto-reset event succeeded.\n         ",
                (int) lpThreadParameterId,
                GetCurrentThreadId());
    }
    else {
        printf( " Thread #%d id=0x%08x, wait for auto-reset event failed.\n        ",
                (int) lpThreadParameterId,
                GetCurrentThreadId());
        return 1;
    }

    // 수동 리셋 이벤트를 대기한다.
    printf( " Thread #%d id=0x%08x, waiting for manual-reset event.\n        ",
            (int) lpThreadParameterId,
            GetCurrentThreadId());
    dwStatus = WaitForSingleObject( g_hManualResetEvent, INFINITE);
    if (dwStatus == WAIT_OBJECT_0) {
```

```
        printf( " Thread #%d id=0x%08x, wait for manual-reset event succeeded.\n          ",
                (int) lpThreadParameterId,
                GetCurrentThreadId());
    }
    else {
        printf( " Thread #%d id=0x%08x, wait for manual-reset event failed.\n          ",
                (int) lpThreadParameterId,
                GetCurrentThreadId());
        return 2;
    }

    // 종료한다.
    printf( " Thread #%d id=0x%08x, exiting.\n      ",
            (int) lpThreadParameterId,
            GetCurrentThreadId());
     return 0;
}

int main(void) {
    DWORD dwStatus;
    BOOL bStatus;
    HANDLE ahThreads[2];
    DWORD dwThreadID1;
    DWORD dwThreadID2;

    // 자동 리셋 이벤트를 생성한다.
    g_hAutoResetEvent = CreateEvent( NULL, FALSE, FALSE, NULL);
    if (g_hAutoResetEvent) {
        printf( " Primary thread created auto-reset event.\n       ");
    }
    else {
        printf( " Primary thread unable to create auto-reset event.\n        ");
        exit(-1);
    }

    // 수동 리셋 이벤트를 생성한다.
    g_hManualResetEvent = CreateEvent( NULL, TRUE, FALSE, NULL);
    if (g_hManualResetEvent) {
        printf( " Primary thread created manual-reset event.\n       ");
    }
    else {
        printf( " Primary thread unable to create manual-reset event.\n        ");
        exit(-2);
    }
```

```
// 첫째 자식 스레드를 생성한다.
ahThreads[0] = (HANDLE) _beginthreadex(NULL, 0, ThreadStartFunc, (void*) 1,
                0, (unsigned *)&dwThreadID1);
if (ahThreads[0]) {
     printf( " Primary thread created child thread #1 with id=0x%08x.\n          ",
              dwThreadID1);
}
else {
     printf( " Primary thread unable to create child thread #1.\n          ");
     exit(-3);
}

// 둘째 자식 스레드를 생성한다.
ahThreads[1] = (HANDLE) _beginthreadex(NULL, 0, ThreadStartFunc, (void*) 2,
                0, (unsigned *)&dwThreadID2);
if (ahThreads[1]) {
     printf( " Primary thread created child thread #2 with id=0x%08x.\n          ",
              dwThreadID2);
}
else {
     printf( " Primary thread unable to create child thread #2.\n          ");
     exit(-4);
}

// 이벤트를 시그널하기 전에 자식 스레드가 수행을 시작하도록 한다.
// 프로그램이 제대로 동작하는 데에는 전혀 상관이 없지만, 이렇게 코드를
// 구성함으로써 둘째 스레드에게 자동 리셋 이벤트를 처음으로 시그널링할
// 수 있는 동일한 기회를 줄 수 있다.
Sleep(1000);

// 자동 리셋 이벤트를 시그널한다.
printf( " Primary thread signaling auto-reset event.\n        ");
bStatus = SetEvent(g_hAutoResetEvent);
if (!bStatus) {
    printf( " Unable to signal auto-reset event.\n       ");
     exit(-5);
}

// 수동 리셋 이벤트를 시그널한다.
printf( " Primary thread signaling manual-reset event.\n        ");
bStatus = SetEvent(g_hManualResetEvent);
if (!bStatus) {
     printf( " Unable to signal manual-reset event.\n       ");
     exit(-6);
}
```

```
    // 자식 스레드 하나가 종료할 때까지 대기한다.
    printf( " Primary thread waiting for one of the child threads to exit.\n          ");
    dwStatus = WaitForMultipleObjects( 2, ahThreads, FALSE, INFINITE);
    if ((dwStatus == WAIT_OBJECT_0) || (dwStatus == WAIT_OBJECT_0 + 1)) {
          printf( " Wait for one child thread to exit succeeded.\n       ");
    }
    else {
          printf( " Wait for one child thread to exit failed.\n       ");
          exit(-7);
    }

    // 자동 리셋 이벤트를 다시 시그널한다.
    printf( " Primary thread signaling auto-reset event again.\n        ");
    bStatus = SetEvent(g_hAutoResetEvent);
    if (!bStatus) {
         printf( " Unable to signal auto-reset event again.\n       ");
         exit(-8);
    }

    // 스레드 두 개가 모두 종료할 때까지 대기한다.
    printf( " Primary thread waiting for both child threads to exit.\n         ");
    dwStatus = WaitForMultipleObjects( 2, ahThreads, TRUE, INFINITE);
    if (dwStatus == WAIT_OBJECT_0) {
         printf( " Wait for both child threads to exit succeeded.\n       ");
    }
     else {
         printf( " Wait for both child threads to exit failed.\n       ");
         exit(-9);
    }

    // 정리 작업
    CloseHandle(ahThreads[0]);
    CloseHandle(ahThreads[1]);
    CloseHandle(g_hAutoResetEvent);
    CloseHandle(g_hManualResetEvent);

    // 종료
    printf( " Primary thread exiting.\n    ");
    return 0;
}
```

주 스레드는 main에서 자동 리셋 이벤트와 수동 리셋 이벤트를 각각 하나씩 생성한다. 그리고나서 주 스레드는 같은 프로시저를 사용하는 두 개의 자식 스레드를 생성하고, 스레드 번호(1 또는 2)를 lpThreadParameterId 매개변수로 넘겨준다. 자식 스레드가 생성된 후에 주 스레드는 1초간 슬립(sleep)해서 모든 자식 스레드가 자동 리셋 이벤트를 기다리게 한다.

자식 스레드 프로시저는 매우 간단하다. 먼저 자동 리셋 이벤트를 대기하고 나서 수동 리셋 이벤트를 대기한다. 모든 경우의 에러 검사 코드가 완료되었고, 필요한 정보를 사용자에게 출력하도록 코딩되었다.

자식 스레드는 먼저 자동 리셋 이벤트를 대기하기 때문에 주 스레드가 슬립하고 난 후 g_hAutoResetEvent를 시그널했을 때 스레드 중 하나만 블록 상태에서 벗어난다. 그리고나서 주 스레드는 수동 리셋 이벤트를 시그널하며, 이 이벤트는 한동안 시그널된 상태로 남아있는다. g_hAutoResetEvent가 시그널될 때 블록 상태에서 벗어난 자식 스레드는 g_hManualResetEvent를 대기하고, 다른 스레드는 WaitForSingleObejct 함수가 대기하는 g_hAutoResetEvent에 대해서 블록당한 상태로 남는다.

메인 프로시저는 bWaitAll 매개변수가 FALSE로 정해진 WaitForMultipleObjects 함수를 호출해서 두 개의 자식 스레드 중 하나가 종료할 때까지 대기한다. 먼저 종료하는 스레드는 g_hAutoResetEvent가 시그널될 때 블록 상태에서 벗어난다. 다음으로 주 스레드는 다른 자식 스레드가 대기하던 자동 리셋 이벤트를 다시 시그널한다. 이 자식 스레드는 즉시 g_hManualResetEvent를 대기하던 WaitForSingleObject 함수에서 리턴한다. 왜냐하면 앞에서 호출한 SetEvent에서 이 이벤트는 시그널된 상태로 남아있었고 그 이후에는 ResetEvent를 호출하지 않았기 때문이다.

이제 남아있는 자식 스레드가 종료하고, 주 스레드가 둘째 WaitForMultipleObjects의 블록 상태에서 벗어난다. 이 WaitForMultipleObjects 함수는 bWaitAll을 TRUE로 했기 때문에 모든 스레드가 종료해야만 리턴한다. 마지막으로 모든 열린 핸들이 닫히고 주 스레드는 정상적으로 수행을 종료한다.

[예제 4-6]은 EventDemo.cpp 프로그램에 의한 결과의 예이다.

[예제4-6] EventDemo.cpp 프로그램의 실행 결과

```
Primary thread created auto-reset event.
Primary thread created manual-reset event.
Primary thread created child thread #1 with id=0xfffabeeb.
Primary thread created child thread #2 with id=0xfffab883.
Thread #1 id=0xfffabeeb, started.
Thread #1 id=0xfffabeeb, waiting for auto-reset event.
Thread #2 id=0xfffab883, started.
```

```
Thread #2 id=0xfffab883, waiting for auto-reset event.
Primary thread signaling auto-reset event.
Primary thread signaling manual-reset event.
Primary thread waiting for one of the child threads to exit.
Thread #1 id=0xfffabeeb, wait for auto-reset event succeeded.
Thread #1 id=0xfffabeeb, waiting for manual-reset event.
Thread #1 id=0xfffabeeb, wait for manual-reset event succeeded.
Thread #1 id=0xfffabeeb, exiting.
Wait for one child thread to exit succeeded.
Primary thread signaling auto-reset event again.
Primary thread waiting for both child threads to exit.
Thread #2 id=0xfffab883, wait for auto-reset event succeeded.
Thread #2 id=0xfffab883, waiting for manual-reset event.
Thread #2 id=0xfffab883, wait for manual-reset event succeeded.
Thread #2 id=0xfffab883, exiting.
Wait for both child threads to exit succeeded.Primary thread exiting.
```