다중 쓰레드 프로그래밍 기법

(Multi-Threaded Programming Techniques)

Win32 환경은 기본적으로 쓰레드를 기본적인 실행 단위로 하는 환경이다. 그렇기 때문에, 최근에 멀티 쓰레딩이라는 단어를 심심치 않게 듣게 되며 또한 조금만 복잡한 어플리케이션을 개발할 때에도 멀티 쓰레드를 지원하지 않으면 이는 Win32 어플리케이션으로서의 자격이 없다고 까지 말하고 있다.
그렇다면 멀티 쓰레드를 지원하는 어플리케이션이란 어떤 것을 말하며, 이를 지원하려면 어떤 지식을 알아야 하는가 ? 이번 장은 여기에 대한 궁금증을 파헤쳐볼 것이다.
마이크로소프트의 Win32 환경 중에서 가장 괄목할 만하게 성장한 기술이라고 생각되는 것 중의 하나가 멀티 태스킹이라고 할 수 있다. 윈도우 95 나 NT 에서는 프로세스와 쓰레드란 개념을 가지고 멀티 태스킹을 행한다. 프로세스(Process)란 실행 중인 프로그램의 인스턴스를 가리키는 것으로 예를 들어, 델파이 4 인스턴스 하나와 MS 워드 인스턴스 하나를 현재 실행하고 있다면 두 개의 프로세스가 시스템에서 실행되고 있는 것이다. 쓰레드(Thread)는 이러한 프로세스 내에서 실행되는 경로(path)를 의미하는 것으로, 프로세스가 운영체제에 의해서 생겨날 때, 쓰레드도 같이 생성된다. 이렇게 프로세스와 같이 생겨나는 쓰레드를 메인 쓰레드라고 한다. 모든 프로세스는 최소한 하나의 쓰레드를 가지고 있으며, 메인 쓰레드가 실행되면 프로세스 내에서 다른 쓰레드를 생성할 수 있게 된다.
다시 말해, 프로세스는 고유한 메모리, 파일 핸들, 그리고 다른 시스템 자원을 갖는 실행중인 프로그램을 말하며 모든 프로세스는 쓰레드라는 개별적인 실행 경로를 가지는 것이다. 대부분의 경우 프로세스 내의 모든 쓰레드는 프로세스의 모든 코드와 데이터 공간을 공유하는데, 예를 들어 두 쓰레드가 동일한 전역 변수를 공유할 수 있다. 쓰레드는 기본적으로 운영체제에 의해 관리되며 각 쓰레드는 고유한 스택을 가지고 있다.
멀티 쓰레드의 예를 들어보자. 어떤 프로그램이 데이터베이스에 접근해서 정보를 검색하고, 동시에 프린터에 인쇄를 한다고 하자. 이를 하나의 프로세스에서 하나의 쓰레드로 구현하려면 데이터베이스 검색이 끝나고 인쇄를 하던가, 아니면 인쇄를 하고 데이터베이스 검색을 해야 한다. 그렇지만 데이터베이스 검색 쓰레드와 인쇄 쓰레드를 각각 생성해서 동시에 실행시킨다면 이것이 곧 멀티 쓰레드가 실행되는 것이다.

멀티 태스킹과 멀티 쓰레드

프로세스는 실행을 위해 메모리에 적재된 프로그램을 말한다. 각각의 프로세스는 가상 주소 공간을 가지고 있으며, 코드와 데이터 그리고 파일, 파이프, 동기화 객체 등의 여러가지 시스템 리소스로 이루어져 있다. 이들은 하나의 쓰레드로 시작하지만, 추가적인 쓰레드를

생성할 수 있다.
쓰레드는 다른 쓰레드에 의해서 실행되는 부분을 포함해서, 프로그램의 어떤 부분의 코드이든 실행할 수 있다. 쓰레드는 운영체제가 CPU 의 시간을 할당하는 기본 단위가 되며, 프로세스의 모든 쓰레드 들은 가상 주소 공간을 공유하기 때문에 전역 변수와 프로세스의 시스템 리소스에 접근할 수 있다.
멀티 태스킹 운영체제는 가능한 CPU 시간을 쓰레드에 분할해서 사용하게 되는데, Win32 API 는 선점형 멀티 태스킹에 적합하도록 디자인 되었기 때문에, 현재 실행 중인 쓰레드가 지정된 시간이 지나면 실행이 중단되고 다른 쓰레드가 작동하게 되어 있다. 이때 각 쓰레드에 대한 주소 공간이나 스택 등의 내용을 담고 있는 구조체를 저장했다가 복구하는 작업을 하게 된다.

언제 멀티 태스킹을 사용하는가 ?

다음과 같은 경우에는 멀티 쓰레드 프로그래밍이 매우 유용하다.

- 멀티 윈도우의 입력을 처리할 때
- 여러 개의 통신 장비에서 입력을 처리할 때
- 각각의 태스크들에 우선권(priority)를 부여할 필요가 있을 때

멀티 프로세스를 이용해서 멀티 태스킹을 하는 것보다 하나의 프로세스 공간에서 멀티 쓰레드를 사용하는 것이 좋은 점은 다음과 같다. (멀티 프로세스와 멀티 쓰레드의 차이점은 모두 알고 계시죠 ? 멀티 프로세스는 프로세스 공간이 여러 개인 경우이며, 멀티 쓰레드는 쓰레드가 여러 개)

- 시스템은 프로세스를 생성하고 실행하는 것보다 쓰레드를 생성하고 실행하는 것을 훨씬 빠르게 수행한다. 이는 쓰레드에 대한 코드는 이미 프로세스의 주소 공간에 매핑이 되어 있는데에 비해, 새로운 프로세스에 대한 코드는 반드시 다시 로드 되어야 하기 때문이다.
- 프로세스 내의 모든 쓰레드 들은 같은 주소 공간을 사용하기 때문에, 프로세스의 전역 변수에 접근할 수 있다. 그러므로, 이를 이용해서 쓰레드 간의 통신이 용이하다.
- 프로세스내의 모든 쓰레드는 파일이나 파이프 등의 프로세스의 리소스에 대한 핸들을 사용할 수 있다

멀티 쓰레드를 사용할 때에 반드시 장점만 있는 것은 아니다. 여러 개의 쓰레드가 같은 리소스에 접근할 경우에는 이들이 충돌하지 않도록 동기화 해주어야 한다. 특히 통신 포트나 디스크 드라이브 등의 시스템 리소스에 접근하는 경우이거나, 파일이나 파이프 핸들 등의

공유 리소스에 대한 핸들, 그리고 프로세스의 전역 변수 등에 대해서는 각별히 주의해서 사용해야 한다. 적절한 동기화를 해주지 못하면 deadlock 등의 심각한 상황에 빠질 수 있다.

Win32 API 의 쓰레드에 대한 지원

이 장에서 Win32 API 를 이용해서 직접 멀티 쓰레드를 구현하는 예제를 하나 제시하고 있다. 그렇지만, 멀티 쓰레드를 이해하기 위해서는 실제로 Win32 API 에서 쓰레드에 대해서 어떤 내용을 지원하고 있는지를 아는 것이 큰 도움이 된다. 여기서는 이들에 대해서 알아보기로 한다.

- 스케쥴 우선권 (Scheduling Priorities)

각 쓰레드의 스케쥴 우선권은 다음의 criteria 에 의해 결정된다.

- 프로세스의 priority class (high, normal, idle)
- 프로세스의 priority class 내의 쓰레드 priority level (lowest, below normal, normal, above normal, highest)
- 시스템이 쓰레드의 base priority 에 적용하는 dynamic priority boost

디폴트로 프로세스의 priority class 는 normal 로 설정된다. 그리고, CreateProcess API 함수는 부모 프로세스가 자식 프로세스의 priority class 를 지정하는 것을 허용한다. 프로세스의 priority class 를 변경하기 위해서는 SetPriorityClass 함수를 사용하며, 현재의 priority class 는 GetPriorityClass 를 사용해서 얻을 수 있다.
이와 비슷하게 쓰레드의 priority level 은 SetThreadPriority 함수와 GetThreadPriority 함수를 이용해서 변경하거나, 얻을 수 있다.
각 쓰레드의 base priority 가 결정되면, 스케쥴러가 이를 이용해서 쓰레드의 dynamic priority 를 결정하게 된다.
스케쥴러는 쓰레드의 dynamic priority 를 올리거나 내림으로써 쓰레드에 특정 사건이 생겼을 때의 반응성을 높일 수 있다. 예를 들어, 윈도우가 키보드 입력이나 마우스 움직임 등의 입력 메시지를 받게 되면 스케쥴러는 해당 윈도우를 소유한 프로세스에 있는 모든 쓰레드의 priority 를 상향 조정한다. 이렇게 상향 조절된 dynamic priority 는 쓰레드가 하나의 time slice 를 완료할 때마다 레벨이 하나씩 낮아져서, 결국에는 base priority 에 도달하게 된다.
이러한, dynamic priority boost 외에, 스케쥴러는 foreground window 에 대한 프로세스의 priority class 를 상향 조정한다. Foreground window 는 background 프로세스와 최저 같거나 높은 priority 를 가지게 된다. 사용자는 이렇게 설정된 foreground priority boost 를

ALT+ TAB 이나 ALT+ ESC 키를 같이 누름으로써 해제할 수 있다.
각 priority 클래스의 의미는 다음과 같다.

Priority	의 미
HIGH_PRIORITY_CLASS	프로세스가 정확하게 실행되기 위해서는 즉시 실행되어야 하는 time-critical 한 작업을 하고 있다는 의미로, normal 또는 idle priority class 를 가지는 다른 쓰레드 들에 우선한다. 이를 사용하는 예로는 CTRL+ALT+DEL 키를 누르면 실행되는 윈도우의 작업 관리자를 들 수 있다. CPU 를 독차지 하므로 지극히 제한된 용도로만 사용하여야 한다.
IDLE_PRIORITY_CLASS	시스템이 실제로 실행시킬 쓰레드가 없을 때 실행되는 정도의 priority class 이다. 스크린 세이버가 여기에 해당된다.
NORMAL_PRIORITY_CLASS	특별한 스케쥴링이 필요 없는 정상적인 프로세스임을 나타낸다.
REALTIME_PRIORITY_CLASS	가장 높은 수준의 priority class 로, 중요한 작업을 수행하는 운영체제의 쓰레드 보다 높은 priority 이다. 이런 쓰레드가 동작할 경우 디스크 캐쉬가 제대로 동작하지 않거나, 마우스 동작이 반응을 하지 않는 등의 현상이 나타날 수 있다.

멀티 쓰레드 프로세스에 대한 priority class 를 선택하고 나서, SetThreadPriority API 함수를 이용해서 상대적인 priority 를 적용시킬 수 있다. 보통은 프로세스의 입력 쓰레드(input thread)에 높은 priority 를 적용하여 사용자의 입력에 대한 반응성을 높인다. 또한, 비교적 CPU 시간을 많이 잡아먹은 쓰레드일 수록 상대적으로 낮은 priority 를 부여하여 필요할 때마다 다른 쓰레드가 실행될 수 있도록 한다. 주의할 점은 높은 priority 를 가진 쓰레드가 낮은 priority 의 쓰레드의 작업이 완료되기를 기다릴 경우에 반드시 wait 함수나 critical section, Sleep 기능을 하는 함수를 이용해야 한다. 만약 여기에서 loop 를 이용할 경우 deadlock 에 들어갈 수 있다.

- 쓰레드의 생성

Win32 API 에서는 CreateThread 함수를 이용해서 프로세스에 새로운 쓰레드를 생성하게 된다. 이 때 새로운 쓰레드가 실행할 코드의 주소를 지정해야 한다.
CreateThread 함수에서 사용되는 파라미터는 다음과 같다.

- 새로운 쓰레드의 핸들에 대한 보안 속성(security attributes). 이러한 보안 속성에는 핸들이 자식 프로세스에 의해 상속될 수 있는지 여부를 결정하는 상속 플래그(inheritance flag)가 포함된다. 또한, 실제로 쓰레드 핸들에 접근하기 전에 접근을 허

용할 것인지 여부를 결정할 때 시스템에 의해 사용되는 보안 설명자(security descriptor)도 포함하고 있다.
- 새로운 쓰레드에 대한 초기 스택 크기. 쓰레드의 스택은 프로세스의 메모리 공간에 자동으로 할당된다. 또한 필요에 따라 커지기도 하고, 쓰레드가 종료되면 해제된다.
- 쓰레드가 생성될 때 일시중지 상태(suspended state)로 시작할 것인지 여부를 결정하는 플래그. 일시중지 상태로 설정되면 쓰레드는 ResumeThread 함수가 호출될 때까지는 실행되지 않는다.

● 멀티 쓰레드의 동기화

멀티 쓰레드로 동작하는 프로그램에서 공유된 자원에 접근할 경우나, 독립된 코드가 적절한 순서로 실행되어야 하는 경우에 동기화가 필요하다. Win32 API 에서는 이러한 동기화에 사용할 수 있는 다음과 같은 여러가지 객체를 제공한다.

- 동기화 객체: 뮤텍(mutex), 세마포어(semaphore), 이벤트(event)
- 파일 핸들 (File handle)
- 콘솔 입력 버퍼 핸들 (Console input buffer handle)
- 명명된 파이프 핸들 (Named pipe handle)
- 통신 디바이스 핸들 (Communication device handle)
- 프로세스 핸들 (Process handle)
- 쓰레드 핸들 (Thread handle)

이들 객체의 상태가 변할 때까지 실행을 중지하게 할 때에는 WaitForSingleObject, WaitForMultipleObjects, WaitForSingleObjectEx 또는 WaitForMultipleObjectsEx 함수를 사용한다.
이들 객체 중 일부는 특정 이벤트가 발생할 때까지 쓰레드가 실행되지 않도록 하는데 유용하다. 예를 들어, 콘솔 입력 버퍼 핸들의 경우 키보드가 눌리거나, 마우스 버튼이 눌릴 때까지 실행을 중지할 수 있고, 프로세스와 쓰레드 핸들의 경우에는 특정 프로세스나 쓰레드가 종료될 때 실행 여부를 결정하게 할 수 있다.
다른 객체의 경우 공유된 자원을 동시에 접근하지 못하도록 하는데 유용하게 사용할 수 있다. 예를 들어 각각의 쓰레드는 뮤텍 객체에 대한 핸들을 가질 수 있는데, 공유 자원에 접근하기 전에 쓰레드는 반드시 뮤텍의 상태가 signaled 로 변할 때까지 대기하는 함수를 호출해야 한다. 뮤텍이 signaled 되면 대기 중인 쓰레드 중에서 하나의 쓰레드가 자원에 접근할 수 있게 되며, 뮤텍 객체의 상태는 다시 non-signaled 로 변경되어 다른 쓰레드의 접근을 하용하지 않게 된다. 쓰레드가 자원의 사용을 끝내면 다른 쓰레드가 접근할 수 있도록 뮤텍 객체의 상태가 signaled 로 변경된다.

단일 프로세스의 쓰레드들일 경우에는 임계섹션 객체(critical section)를 사용할 수 있으며 효율이 조금 더 뛰어나다.

- 쓰레드 로컬 저장소 (Thread Local Storage)

하나의 프로세스에 있는 모든 쓰레드 들은 가상 메모리 공간과 전역 변수를 공유하게 된다. 그렇지만 각각의 쓰레드에 대한 정적인 저장소가 필요한 경우가 있는데, 이럴 때에 물론 쓰레드 함수의 지역 변수를 사용할 수도 있지만, 이러한 지역 변수는 특정 함수 내에서만 값을 유지하기 때문에 그 유용성이 떨어진다. 이와 같이 쓰레드에 독립적이면서, 전역 변수의 역할을 해줄 수 있는 것이 쓰레드 로컬 저장소(Thread Local Storage, TLS)이다. TLS를 사용할 때에 쓰레드는 인덱스를 할당하고, 각 쓰레드에 대한 값을 저장하거나 불러올 수 있게 된다.
TLS를 사용하는 가장 전형적인 방법은 다음과 같다.

1. TlsAlloc 함수를 사용해서 DLL이나 프로세스에 대한 TLS 인덱스를 할당한다.
2. TLS 인덱스를 사용할 필요가 있는 각각의 쓰레드는 동적 저장소(dynamic storage)를 할당하고, TlsSetValue 함수를 사용해서 인덱스와 동적 저장소의 포인터를 연결시킨다.
3. 쓰레드가 저장소에 접근할 필요가 있으면, TlsGetValue 함수에 TLS 인덱스를 지정해서 포인터 값을 불러온다.
4. 더이상 필요하지 않은 동적 저장소는 각각의 쓰레드에 의해 해제되고, 모든 쓰레드가 TLS 인덱스의 사용을 끝내면 TlsFree 함수에 의해 TLS 인덱스를 해제한다.

TLS는 DLL을 사용할 때에도 유용하다. DllEntryPoint 함수에서 프로세스나 쓰레드의 초기 TLS 작업을 수행하고, DLL이 새로운 프로세스와 함께 동작하게 될 때 entry-point 함수가 TlsAlloc 함수를 호출하여 그 프로세스에 대한 TLS 인덱스를 할당하게 한다. 그리고, DLL이 그 프로세스의 새로운 쓰레드와 함께 동작하게 될 때 entry-point 함수에서 그 쓰레드에 대한 동적 저장소를 할당하고, TlsSetValue 함수를 이용해서 데이터를 저장한다. DLL은 TLS 인덱스를 각 프로세스의 전역 변수에 저장하고, DLL의 함수들은 TLS 인덱스를 이용해서 TlsGetValue 함수를 호출하면 쓰레드에서 데이터에 접근할 수 있게 된다.

델파이 쓰레드의 구조

델파이에서 쓰레드는 TThread 클래스를 상속받아 만드는 새로운 클래스에서 구현하는 것이 보통이다. 일단 상속을 받고 나서 몇 개의 중요한 함수를 작업에 맞도록 수정해야 하는데, 이런 작업에 필수적으로 포함되는 메소드가 Create, Execute, Destroy 메소드이다. Create 메소드에서는 쓰레드가 필요로 하는 리소스를 할당해 주어야 하며, Destroy 메소드

에서는 사용한 리소스를 해제해 주는 역할을 해야 한다. Execute 메소드가 실제 멀티 쓰레드 어플리케이션을 만들 때 가장 핵심이 되는데, 일단 쓰레드가 생성되고 나면 보통 Execute 메소드가 호출된다. 그러므로, 실제로 실행되어야 하는 모든 작업에 대한 코드가 Execute 메소드에 위치해야 하는 것이다.

TThread 클래스의 Create 메소드는 CreateSuspended 라는 파라미터를 가진다. 이 파라미터는 Boolean 데이터 형으로 쓰레드가 생성될 때 바로 Execute 메소드를 호출할 것인지 여부를 결정한다. 만약 커스텀 Create 메소드를 사용하는 경우라면 보통 이 파라미터는 True 로 설정한다. 이렇게 함으로써 쓰레드가 사용할 모든 리소스를 할당하기 전에 Execute 메소드가 실행되지 않도록 할 수 있게 된다. 그리고 나서, Resume 메소드가 호출되면 쓰레드가 실행된다. 그러므로, 보통 커스텀 Create 메소드를 사용하는 경우라면 일단 사용할 리소스를 할당하고 나서, 맨 마지막에 Resume 메소드를 호출한다.

TThread 클래스에는 Terminated 라는 프로퍼티가 있는데, 이 프로퍼티는 쓰레드가 현재 실행되고 있는지 여부를 나타내는 프로퍼티이다. Execute 메소드는 실행될 때 이 프로퍼티를 항상 검사를 하게 되어 있어서, 이 프로퍼티가 True 로 설정되면 메소드의 실행을 중단한다. 이 프로퍼티에 맞추어 TThread 클래스는 OnTerminate 라는 이벤트를 제공하는데, 이 이벤트는 쓰레드가 종료되고나서 아직 쓰레드 객체가 메모리에서 해제되기 전에 발생한다. 이 이벤트에 적절한 코드를 대입해서 쓰레드가 끝나기 전에 필요한 여러가지 작업을 하게 할 수 있다. 예를 들어, 쓰레드가 계산한 데이터를 실제로 보여주는 등의 작업을 지시할 수 있겠다.

TThread 클래스의 메소드 중에 Execute 메소드 만큼 중요한 메소드가 또 있는데, Synchronize 메소드가 그것이다. 멀티 쓰레드가 실행되다 보면 여러 개의 쓰레드가 동일한 리소스에 동시에 접근할 수가 있는데 이로 인해 문제가 생길 가능성이 있게 된다. 예를 들어, 두 개의 쓰레드가 폼에 데이터를 동시에 쓰려고 하면 하나의 객체를 그리다가, 다른 쓰레드가 이 동작을 방해할 것이다. 이로 인해 에러가 발생할 소지도 있고, 심할 경우 리소스를 빌리는 과정에서 소위 데드락(Dead Lock)이라고 불리우는 리소스 잠김 현상이 나타날 수도 있다. 델파이에서는 이런 문제를 해결하기 위해서 문제가 생길 소지가 있는 리소스를 사용하는 코드에서는 반드시 Synchronize 메소드를 통해서 실행하도록 하고 있다. 보통은 델파이의 VCL 라이브러리에 접근할 때 이 방법을 사용한다.

그렇지만 Synchronize 메소드를 사용할 때에는 기본적으로 메시지를 메인 VCL 쓰레드에서 실행시키게 하는 것이므로, 만약에 쓰레드에서 실행되는 대부분의 코드가 Synchronize 메소드에서 실행된다면 이것은 싱글 쓰레드 어플리케이션을 실행시키는 것과 크게 다르지 않게 된다는 것을 명심해야 한다.

그러므로, 리소스가 공유될 때를 제외하고는 Synchronize 메소드에서 지정한 코드를 과도하게 사용하는 것은 피해야 한다.

쓰레드의 활용

보통 TThread 클래스를 이용하는 경우는 몇 가지의 통신 디바이스를 통해서 입력을 받는 경우가 가장 전형적인 경우이며, 그 밖에도 파일을 다루는 경우거나 작업을 할 때 작업에 대한 중요도가 차이가 날 때 보다 중요한 작업에는 높은 priority 를 부여하고, 덜 중요한 작업에는 보다 낮은 priority 를 부여하여 작업하면 보다 효율성을 높일 수 있을 것이다.
그 밖에 멀티 쓰레드 프로그래밍을 해야 할 경우로는 다음과 같은 경우가 있다.

1. 초기화가 오래 걸려서 사용자가 오래 기다려야 하는 상황이 발생할 경우
2. 다른 작업을 할 때 틈틈이 각종 리소스를 정리하는 등의 작은 작업을 백 그라운드에서 실행시키고자 할 경우
3. 어플리케이션의 주 기능을 저하시키지 않으면서도 애니메이션 등의 효과 등을 보여주고자 할 경우
4. 다른 작업을 하면서, 백 그라운드로 커다란 쿼리 등을 실행시켜 전체적인 성능을 향상시키고자 할 경우

이번에는 쓰레드를 사용할 때 반드시 알아야 할 주의사항 몇 가지를 알아보도록 하자.

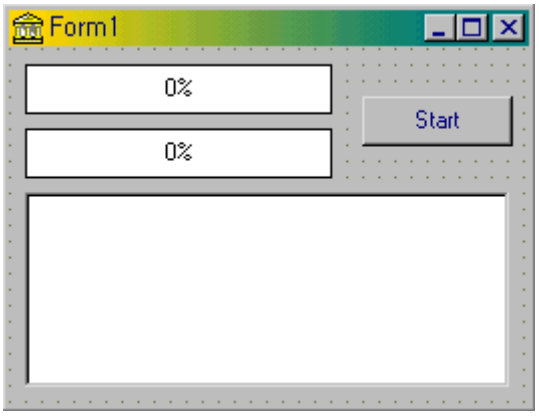
1. 너무 많은 쓰레드를 동시에 작동시키면 시스템의 속도가 심각하게 저하된다. 보통 단일 프로세서 시스템일 경우 프로세스당 16 쓰레드를 넘지 않는 것이 좋다.
2. 여러 개의 쓰레드가 동일한 리소스를 수정할 경우 반드시 동기화를 시켜야 한다.
3. VCL 컴포넌트나 폼을 업데이트하는 메소드는 메인 VCL 쓰레드 내에서 호출되어야 한다.

앞에서도 간단히 설명했지만 TThread 의 주요 프로퍼티로는 쓰레드의 중요도를 결정하는 Priority, 쓰레드의 실행을 Resume 메소드가 실행될 때까지 일시 정지시킬 수 있는 Suspended, 쓰레드의 실행을 중지시킬 때 사용하는 Terminated 프로퍼티 등이 있다.
Suspended 프로퍼티와 Terminated 프로퍼티를 설정하는 것은 각각 Suspend, Terminate 메소드를 실행하는 것과 같은 역할을 하게 된다.

첫번째 예제

쓰레드 객체를 사용할 때에는 일단 TThread 클래스에서 새로운 클래스를 상속받고, 기본적으로 Execute 메소드를 override 하는 것이 가장 기본적인 방법이다. 또한 꼭 기억해야 할 것은 VCL 객체에 접근할 때에는 Synchronize 메소드를 사용해서 메인 쓰레드에 접근해야 한다는 것이다. 우리의 첫번째 예제는 파일을 복사하는 과정을 TGauge 컴포넌트에 그

려주는 루틴을 쓰레드 객체를 이용해서 제작하는 것이다.
일단 다음 그림과 같이 폼을 디자인 하도록 하자.



여기서 Start 버튼(Button1)을 클릭하면 같은 파일을 다른 파일로 복사하는 두 개의 쓰레드가 실행되고, 각각의 쓰레드의 진행과정을 TGauge 컴포넌트에 그려 준다. 그리고, 각각의 쓰레드가 끝나면 TMemo 컴포넌트에 몇 번 쓰레드가 끝났다는 메시지가 나타나도록 해보자.
이렇게 파일을 조작하거나, 인쇄, 통신을 할 때에 멀티 쓰레드 프로그래밍이 많이 사용된다. 먼저 복사를 하는 쓰레드의 이름을 TCopyThread 라고 하자. 그리고, 실제 복사를 할 때 약간의 시간이 지나야 실행시키는 재미가 있으므로, 조금은 커다란 파일을 복사해 보자. 여기서는 델파이 4 의 Help 디렉토리에 있는 del4vcl.hlp 파일을 선택했다. 이 파일의 크기는 10MB 가 넘기 때문에 이번 어플리케이션에 비교적 적절하게 사용될 수 있을 것이다. 그리고, 각 쓰레드가 끝났을 때 적용할 ThreadDied 프로시저를 다음과 같이 TForm1 의 private 섹션에 선언한다.

```
private
  procedure ThreadDied(Sender: TObject);
```

그리고, Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  T1, T2: TCopyThread;
begin
  Memo1.Clear;
  T1 := TCopyThread.Create('C:₩Program Files₩Borland₩Delphi4₩Help₩del4vcl.hlp',
    'C:₩Temp₩test1.hlp', 1);
```

```
  T2 := TCopyThread.Create('C:₩Program Files₩Borland₩Delphi4₩Help₩del4vcl.hlp',
    'C:₩Temp₩test2.hlp', 2);
  T1.OnTerminate := ThreadDied;
  T2.OnTerminate := ThreadDied;
  T1.Resume;
  T2.Resume;
  Button1.Enabled := False;
end;
```

눈치 빠른 독자 들은 TCopyThread 의 Create 메소드를 호출하는 것을 보면 파라미터가 3 개 있는 것으로 보아 커스텀 Create 메소드라는 것을 알 수 있을 것이다. 또한 뒤쪽에 각 쓰레드에 대해 Resume 메소드를 호출한다는 것은 쓰레드가 생성될 때 CreateSuspeded 가 True 로 설정되었다는 것을 의미한다.
TCopyThread 의 Create 메소드는 첫번째 파라미터로 읽어 올 파일 이름, 두번째 파라미터로 덮어 쓸 파일 이름을 지정하며, 세번째 파라미터는 실행되는 쓰레드가 몇 번 쓰레드인지 나타내려고 추가한 것이다. 이 값을 이용해서 몇 번 쓰레드가 끝났다는 메시지를 메모에 기록한다.
각 쓰레드의 OnTerminate 이벤트 핸들러로 ThreadDied 프로시저를 대입하였다. 그러면 ThreadDied 프로시저를 구현해보자.

```
procedure TForm1.ThreadDied(Sender: TObject);
begin
  Memo1.Lines.Add(Format('%d 번 쓰레드 종료.', [(Sender as TCopyThread).ReturnValue]));
end;
```

이제는 실제로 TCopyThread 클래스를 선언하고 구현할 차례이다. type 선언문에 다음과 같은 클래스 선언문을 추가한다.

```
  TCopyThread = class(TThread)
  private
    FSource, FTarget: string;
    FSize, BytesCopied: integer;
  protected
    procedure Execute; override;
  public
    procedure DisplayProgress;
```

```
    constructor Create(const Source, Target: string; Return: integer);
    property ReturnValue;
  end;
```

그다지 어렵지는 않지만 간단하게 선언부를 설명하면, Create 메소드에서 읽어 올 파일과 기록할 파일의 이름과 쓰레드의 번호를 얻어온다. Button1 의 OnClick 이벤트 핸들러를 기준으로 하면 1, 2 가 각각의 쓰레드 번호가 된다. 그리고 Execute 메소드를 override 하는데, 이 메소드에서 private 섹션에서 정의한 필드 값을 이용해서 실제 쓰레드를 수행한다. 이때 TGauge 컴포넌트를 사용하는 메소드가 DisplayProgress 인데 이와 같이 대부분의 VCL 컴포넌트에 접근하는 경우에는 Synchronize 메소드를 사용해서 호출해야 하므로, 따로 선언해 준다.
그럼 먼저 Create 메소드를 구현해 보자

```
constructor TCopyThread.Create(const Source, Target: String; Return: Integer);
begin
  inherited Create(True);
  FreeOnTerminate := True;
  FSource := Source;
  FTarget := Target;
  ReturnValue := Return;
end;
```

단순한 코드 이므로 별 설명이 필요없을 것으로 생각된다. FreeOnTerminate 프로퍼티는 True 로 설정되었을 때 쓰레드가 종료됨과 동시에 메모리에서 객체가 해제되게 하는 것이다. 보통 True 로 설정하는 경우가 많다.
그러면, TGuage 에 표시하는 DisplayProgress 메소드를 구현하자.

```
procedure TCopyThread.DisplayProgress;
begin
  if ReturnValue = 1 then
    Form1.Gauge1.Progress := Round(BytesCopied/FSize*100.0)
  else
    Form1.Gauge2.Progress := Round(BytesCopied/FSize*100.0);
end;
```

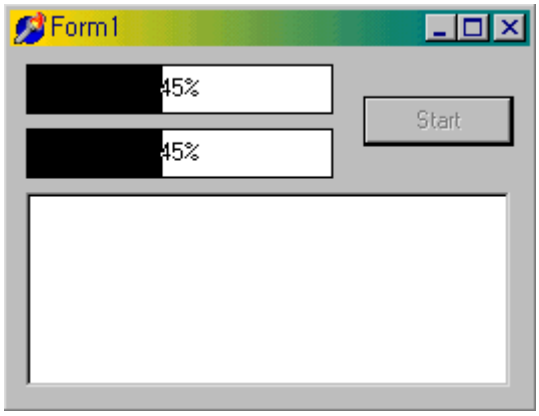
간단히 설명하면 ReturnValue 가 1, 즉 쓰레드를 생성할 때 세번째 파라미터를 1 로 설정한

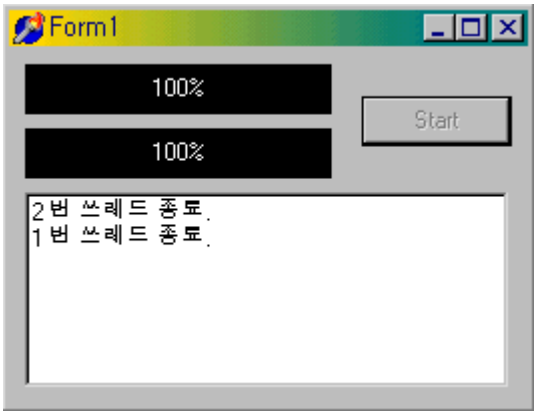
Copy 쓰레드의 진행상황은 Gauge1 에 표시되고, 다른 쓰레드는 Gauge2 에 표시하라는 이야기이다. 이때 진행된 상황의 계산은 파일의 실제 사이즈와 복사된 바이트를 나누어서 표시하면 된다. 마지막으로 Execute 메소드를 구현하면 첫번째 예제가 완성된다.

```
procedure TCopyThread.Execute;
var
  FIn, FOut: TFileStream;
  ChunkSize: Integer;
begin
  FIn := TFileStream.Create(FSource, fmOpenRead or fmShareDenyWrite);
  try
    FSize := FIn.Size;
    BytesCopied := 0;
    FOut := TFileStream.Create(FTarget, fmCreate);
    try
      while BytesCopied < FSize do
      begin
        ChunkSize := FSize – BytesCopied;
        if ChunkSize > $8000 then ChunkSize := $800;
        Inc(BytesCopied, FOut.CopyFrom(FIn, ChunkSize));
        Synchronize(DisplayProgress);
      end;
    finally
      FOut.Free;
    end;
  finally
    FIn.Free;
  end;
end;
```

실제 복사하는 루틴은 TFileStream 클래스를 이용하였다. 비교적 사용하기 편리한 루틴이므로 파일을 다루는 어플리케이션을 만들 때 다양하게 응용할 수 있을 것이다. 눈 여겨 보아야할 부분은 FSize 값에 먼저 FIn 파일 스트림 객체를 생성하면서 원본 파일의 크기를 대입하고, 실제로 복사를 블록 단위로 하면서 BytesCopied 값을 증가시키는 부분이다. 이렇게 한 블록이 복사될 때 마다 Synchronize(DisplayProgress)를 호출하여 복사가 진행되는 상황을 한 눈에 파악할 수 있게 된다. Synchronize 메소드의 파라미터에는 이와 같이

직접 프로시저의 이름을 대입해서 사용하면 된다.
다음 그림들은 이 예제의 실제 실행화면 들이다.





델파이 4 에서 향상된 쓰레드 관리

델파이 4 는 멀티 쓰레드 어플리케이션의 개발을 돕기 위해 몇 가지 새로운 클래스를 제공하고 있다. TCriticalSection 클래스는 멀티 쓰레드 어플리케이션에서 하나의 쓰레드의 실행을 막을 수 있는 클래스이다. 이 클래스는 특정 작업이 끝나기 전에 다른 쓰레드가 실행되면 안되는 경우, 이를 보호할 수 있다. 그렇지만, TCriticalSection 은 다른 모든 쓰레드에서 실행되지 않도록 하는 것이므로, 함부로 남용하면 실행 속도를 심각하게 저해할 수도 있다.
이 클래스를 사용하려면 일단 CriticalSection 객체를 생성하고, 보호할 코드의 앞에서 Enter 또는 Acquire 메소드를 호출한다. 그리고, 코드의 끝 부분에 Leave 또는 Release 메소드를 호출한다.
또 하나의 새로운 클래스는 TThreadList 클래스이다. 이 클래스는 멀티 쓰레드 환경에서도 사용할 수 있는 TList 클래스라고 생각하면 된다. 각각의 TThreadList 객체는 TList 객체를 관리하게 되는데, TList 의 아이템을 사용할 때에는 LockList 메소드를 사용하면 사용할 TList 객체를 반환하며, 사용하고 나서 UnLock 을 호출하면 된다. 간단한 사용 방법은 다음과 같다.

```
with MyThreadList.LockList do
  try
    for X := 0 to Count-1 do
    Something(Items[X]);
  finally
    MyThreadList.UnlockList;
  end;
```

멀티 쓰레드 데이터베이스 어플리케이션

멀티 쓰레드 데이터베이스 어플리케이션을 제작하는 데에는 몇가지 고려해야 할 점이 있는데 먼저 여기에 대해서 알아보자.

1. 데이터베이스에 접근하는 각각의 쓰레드는 자신의 세션을 가져야 한다.

BDE 를 통해 데이터베이스에 접근할 때에는 세션을 통해서 연결하게 되어 있다. 각각의 델파이 프로그램은 명시적으로 적어주지 않더라도 기본적인 디폴트 세션이 자동적으로 사용되며, 모든 데이터는 이를 통해서 접근하게 되어 있다. 그런데, 멀티 쓰레드를 사용할 경우 동시에 여러 개의 데이터베이스 작업이 이루어질 수 있다. 이 때 TSession 객체가 하나 밖에 없다면 병목 현상이 일어날 수 밖에 없는 것이다. 이러한 현상의 해결책은 각각의 쓰레드에 자신의 세션을 가지게 하는 것이다. 기본적으로 BDE 는 멀티 쓰레드에 영향을 받지 않도록 디자인 되어 있기 때문에 여러 개의 세션을 통해서 동시에 데이터에 접근하는 것을 허용한다.

2. TQuery 나 TTable 을 사용할 때에는 Synchronize 메소드를 사용하지 않는다.

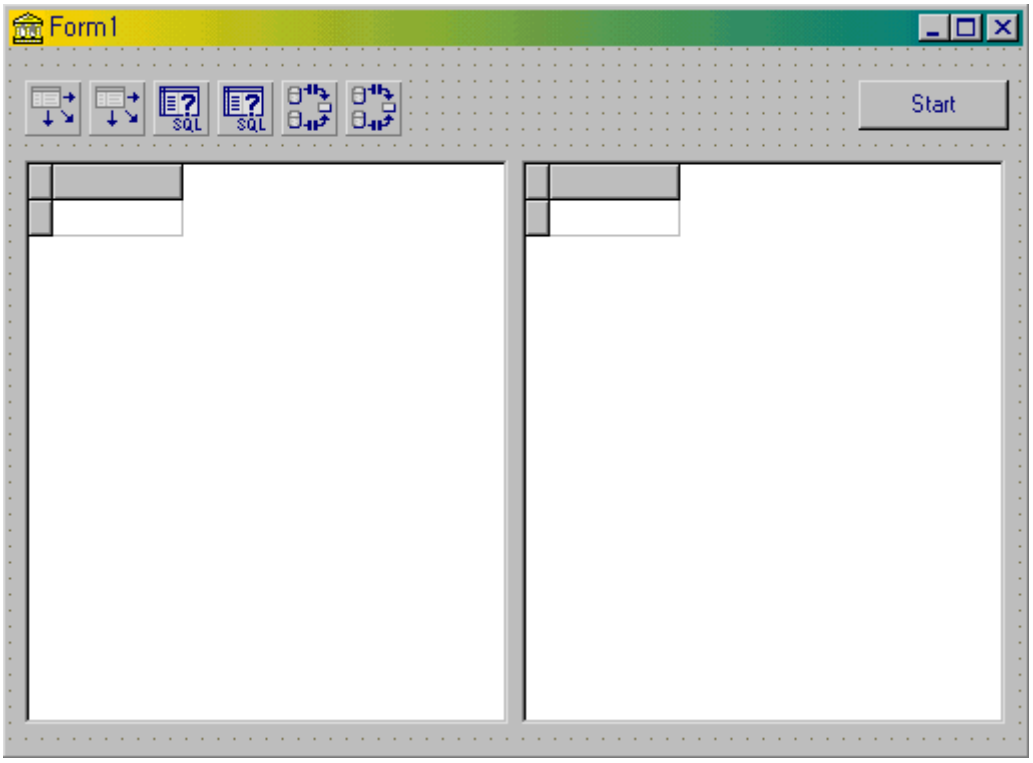
앞에서 언급했지만 보통 그래픽을 사용하는 VCL 객체를 이용할 때에는 Synchronize 메소드를 사용한다. 그런데, TQuery 객체가 Synchronize 메소드를 통해서 접근될 때에는 쿼리가 실행될 동안 델파이의 메인 쓰레드가 실행을 멈추게 된다. 이렇게 되면 멀티 쓰레드 어플리케이션을 제작한 의미가 거의 없게 되어 버린다.
BDE 는 기본적으로 멀티 쓰레딩을 지원하므로 Synchronize 를 사용하지 않고 TQuery, TTable 에 접근해도 된다.

3. TQuery 와 TTable 객체가 Open 되어 있을 때에는 TDataSource 와 연결하지 않는다.

TDataSet 가 Open 되어 있을 때 TDataSource 와 연결되면 델파이가 연결되어 있는 데이터 어웨어 컨트롤에 결과를 보여주려고 시도하게 된다. 그러므로, 이럴 때에는 TDataSet 를 Open 하기 전에 Synchronize 메소드를 사용해서 TDataSet 와 TDataSource 의 연결을 일단 끊어야 한다. 일단 TDataSet 가 Open 되면 TDataSource 와 Synchronize 메소드를 사용해서 연결한다.

백그라운드 쿼리 실행 예제

멀티 쓰레드를 이용한 데이터베이스 어플리케이션을 구현하는 아주 간단한 예제를 제작해 보자. 이 예제는 사실 그다지 바람직하지 않은 구현 부분도 있지만, 동시에 두 개의 데이터베이스에 접근해서 쿼리를 실행하는 것을 직접 눈으로 확인할 수 있어서 멀티 쓰레드의 장점을 음미하는데에는 도움이 될 것이다.
먼저, 폼 위에 다음 그림과 같이 TDataSource, TQuery, TSession, TDBGrid 컴포넌트를 각각 두 개씩과 버튼을 하나 올려 놓는다.



그리고, DataSource1, DataSource2 객체의 DataSet 프로퍼티를 각각 Query1, Query2 로 설정하고 DBGird1, DBGrid2 객체의 DataSource 프로퍼티를 각각 DataSource1, DataSource2 로 설정한다. 또한, Query1 과 Query2 의 SessionName 프로퍼티를 각각 Session1 과 Session2 의 SessionName 프로퍼티 값으로 설정한다. 여기서는 각각 'Sample1', 'Sample2'로 설정하였다. 앞에서도 잠시 설명했지만 BDE 가 멀티 쓰레드로 실행되려면 이렇게 각각의 쓰레드에 대해 다른 TSession 객체가 정의되어 있어야 한다.
마지막으로 실행할 쿼리를 입력할 차례인데, 일단 DatabaseName 프로퍼티를 DBDEMOS 로 설정한다. Query1 의 SQL 문장으로는 Employee.db 에서 레코드를 가져오도록 'SELECT * FROM "emplyee.db".', Query2 는 Customer.db 에서 레코드를 가져오도록 'SELECT * FROM "customer.db".'로 설정한다.
이제 쓰레드 클래스를 선언하도록 하자. type 선언문에 다음과 같이 선언한다.

```
TQThread = class(TThread)
private
  FQuery: TQuery;
protected
  procedure Execute; override;
public
  constructor Create(Query: TQuery);
end;
```

이 쓰레드 클래스의 구현은 아주 간단하게 한다. Create 메소드에서 파라미터로 넘어온 Query 파라미터를 FQuery 필드에 저장하는 정도로 구현하고, Execute 메소드에서 FQuery 필드를 Open 하는 것으로 충분한다.

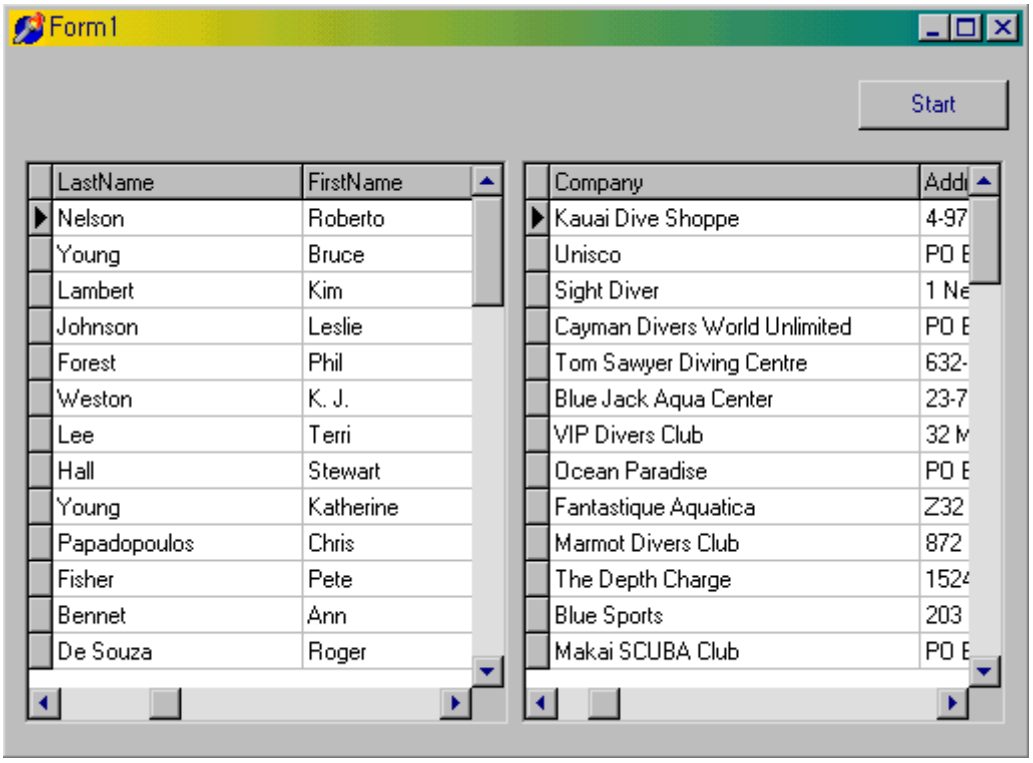
```
constructor TQThread.Create(Query: TQuery);
begin
  inherited Create(True);
  FQuery := Query;
  FreeOnTerminate := True;
  Resume;
end;

procedure TQThread.Execute;
begin
  FQuery.Open;
end;
```

마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TQThread.Create(Query1);
  TQThread.Create(Query2);
end;
```

이제 프로젝트를 실행시켜 보자. 워낙 빨리 실행되기 때문에 잘 관찰하기는 어렵겠지만 다음 그림과 같이 두 개의 데이터베이스의 레코드가 DBGrid 에 동시에 뿌려진 것을 관찰할

수 있을 것이다.



Win32 API 를 이용한 멀티 쓰레드 구현

멀티 쓰레드 프로그래밍을 하면서도 델파이의 TThread 객체를 사용하기가 어려운 경우가 있다. 예를 들어, 아주 간단한 프로시저나 함수를 루프에 넣어서 돌릴 경우에 TThread 객체를 생성해서 쓰기가 좀 아깝다고 생각될 수가 있다. 이럴 때에는 Win32 API 를 직접 호출해서 멀티 쓰레드 프로그래밍을 하는 것이 낫다.

API 를 이용한 멀티 쓰레드 프로그래밍을 할 때에는 먼저 쓰레드 자체를 생성하고, 쓰레드의 엔트리 포인트가 되는 함수를 작성한 후, 이를 쓰레드에게 알려주어 실행하게 하는 절차를 밟는다.

일반적인 함수의 구현과는 달리 멀티 쓰레드를 구현하는 함수에는 다음과 같은 제한점을 가지고 있다. 기본적으로 파라미터를 Pointer 형으로 하나만 가져야 하고, 반드시 LongInt 형의 정수값을 반환해야 한다. 그리고, 표준 윈도우에 의해서 파라미터가 전달되므로 stdcall 로 선언되어야 한다. 전형적인 쓰레드 함수의 선언문은 다음과 같다.

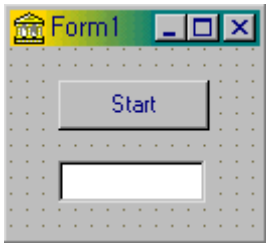
```
function MyThreadFunc(Ptr: Pointer): LongInt; stdcall;
```

그러면, 쓰레드를 생성하는 API 함수인 CreateThread 함수에 대해서 알아보자. 이 함수의

선언문은 다음과 같다.

```
function CreateThread(lpThreadAttributes: Pointer; dwStackSize: DWORD;
  lpStartAddress: TFNThreadStartRoutine; lpParameter: Pointer; dwCreationFlags:
  DWORD; var lpThreaded: DWORD): THandle; stdcall;
```

lpThreadAttributes 파라미터는 쓰레드의 보안 특성을 설정하는 주소를 지정하는 것으로 대개 nil 로 설정하게 되며, dwStackSize 파라미터는 쓰레드에 대한 스택의 크기를 지정하는 것으로 보통 0 으로 설정한다. lpStartAddress 파라미터가 가장 중요한 것으로 멀티 쓰레드로 실행될 함수의 주소를 넘겨 준다. lpParameter 파라미터에는 쓰레드에서 사용할 변수의 주소를 넘겨주도록 되어 있는데, 이를 이용해도 되지만 전역 변수를 이용하는 방법이 더 편리하므로 대개는 nil 로 설정한다. dwCreationFlags 파라미터는 대개 0 으로 설정하며, 마지막으로 lpThreaded 파라미터에는 생성한 쓰레드의 ThreadID 의 참조값이 넘어오게 된다.
그러면, 이를 이용해서 실제로 멀티 쓰레드를 구현하는 예제를 하나 제작해 보도록 하자.
예제는 아주 단순하게 'Start'라는 버튼을 클릭하면 1 부터 1000 까지의 정수를 하나씩 에디트 컨트롤에 표시하는 예제이다. 먼저 다음 그림과 같이 폼을 디자인한다.



그리고, 앞에서 간단히 설명한 데로 쓰레드 함수로 적당한 형태의 함수를 다음과 같이 구현한다.

```
function MyThreadFunc(P: Pointer): LongInt; stdcall;
var
  Temp: String;
  i: Integer;
begin
  EnableWindow(Form1.Button1.Handle, False);
  for i := 1 to 1000 do
  begin
    Temp := IntToStr(i);
```

```
    SendMessage(Form1.Edit1.Handle, WM_SETTEXT, 0, LongInt(PChar(Temp)));
  end;
  EnableWindow(Form1.Button1.Handle, True);
end;
```

아주 간단한 함수인데, 다시 한번 강조하지만 Pointer 형인 파라미터 하나만 가진 함수로 LongInt 형을 반환하며 stdcall 로 선언한다.
EnableWindow 와 SendMessage 를 쓴 이유는 VCL 코드를 직접 사용하면 VCL 클래스는 멀티 쓰레드와 충돌을 일으킬 수 있기 때문에, Button1 과 Edit1 컨트롤의 Handle 을 직접 이용해서 API 로 처리한 것이다.
EnableWindow 함수는 핸들을 넘겨준 컨트롤을 사용할 수 있게 한 것으로 VCL 컨트롤의 Enable 프로퍼티를 설정하는 것과 같은 역할을 하게 된다. 1 부터 1000 까지 숫자를 에디트 컨트롤에 표시하고, 그동안 Button1 을 사용할 수 없게 하고 이 작업이 끝나면 Button1 을 사용할 수 있게 하는 것이다.
WM_SEXTEXT 메시지는 컨트롤에 텍스트를 설정하는 것으로, VCL 컴포넌트의 Text 프로퍼티를 사용하는 것과 동일한 역할을 한다. 이때 SendMessage 함수의 파라미터로는 LongInt 데이터 형을 사용하기 때문에 형변환이 필요하다.
이 함수를 작동시키기 위한 Button1 의 OnClick 이벤트 핸들러는 다음과 같이 작성하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Thread: THandle;
  ThrID: DWORD;
begin
  Thread := CreateThread(nil, 0, @MyThreadFunc, nil, 0, ThrID);
  if Thread = 0 then ShowMessage('쓰레드가 생성되지 않았습니다 !');
end;
```

어렵지 않은 코드 이므로, 금방 이해할 수 있을 것이다.
프로젝트를 실행해 보면 단순하게 멀티 쓰레드를 구현할 수 있다는 것을 알 수 있을 것이다.

쓰레드 대기 처리

멀티 쓰레드를 이용해서 어플리케이션을 제작하다 보면 쓰레드가 모두 끝난 후에 특정 작업을 해야 할 경우가 생긴다.

이를 해결하기 위한 방법으로 가장 손쉬운 것은 Boolean 형의 전역 변수를 하나 이용해서 쓰레드가 끝날 때 이를 True 로 설정하여 쓰레드가 종료했는지 여부를 알아보고, 종료되지 않았다면 Application.ProcessMessages 메소드를 호출하여 다른 작업을 계속할 수 있도록 하는 것이다.
Application.ProcessMessages 메소드를 이용하면 프로그램이 메시지를 받고 처리할 수 있게 되기는 하지만, 상당한 CPU 시간의 소모가 있게 된다. 특히, 여러 개의 쓰레드를 동시에 실행하였는데 이들이 모두 끝났는지를 알아보아야 한다면 문제는 더 심각해진다.
이를 해결하기 위한 가장 좋은 방법은 대기를 수행하는 대기 쓰레드(wait thread)를 따로 작성하는 것이다. 이러한 대기 쓰레드에는 CPU 시간과 시스템 리소스를 거의 소모하지 않는 효과적인 Win32 API 함수를 호출하여 소기의 목적을 달성할 수 있다.
WaitForSingleObject 와 WaitForMultipleObjects API 함수가 이러한 역할을 수행하는 Win32 API 함수이다. 기본적으로 윈도우에서 모든 객체가 생성될 때 이 객체가 현재 사용중이거나 활성화 되어 있을 때에는 이를 non-signaled 상태에 있다고 하며, 사용 중이 아닐 때에는 signaled 상태에 있다고 한다. 앞의 두가지 API 함수는 지정된 객체 들이 signaled 상태에 들어갈 때까지 대기하는 역할을 해주는 함수이다. 이들 함수의 장점은 CPU 시간을 거의 소모하지 않는 아주 효과적인 대기 상태로 들어간다는 것이다.
WaitForSingleObject 함수는 2 개의 파라미터를 가진다. 대기할 쓰레드의 핸들을 지정하는 THandle 형의 파라미터와 쓰레드가 종료되지 않을 경우에 최고 한도로 몇 millisecond 까지 기다릴 것인지를 설정하게 되어 있다.
보통은 지정한 쓰레드가 종료될 때까지 무조건 기다리는 INFINITE 상수를 설정하게 된다.
간단한 사용 예제 코드는 다음과 같다.

```
WaitForSingleObject(MyThread.Handle, INFINITE);
```

WaitForMultipleObjects 함수는 이보다 조금 더 많은 파라미터를 가진다. 선언문은 다음과 같다.

```
function WaitForMultipleObjects(nCount: DWORD; lpHandles: PWOHandleArray;
  bWaitAll: BOOL; dwMilliseconds: DWORD): DWORD; stdcall;
```

첫번째 파라미터인 nCount 는 DWORD 형으로 쓰레드의 핸들 배열에 있는 핸들의 수를 지정하며, lpHandles 파라미터는 쓰레드 객체의 핸들 배열의 주소를 설정하도록 되어 있다. bWaitAll 은 모든 객체가 signaled 상태에 들어갈 때까지 대기할 것인지 여부를 설정하는 파라미터이며 마지막으로 dwMilliseconds 파라미터는 대기할 시간을 지정하는 것으로 보통 INFINITE 로 설정한다.
이 함수를 사용할 때 가장 특기할 만한 것은 핸들 배열을 사용하는 것이다. 그러면, 실제

로 간단한 예제를 하나 만들어 보자.
폼위에 TLabel 컴포넌트와 TButton 컴포넌트를 하나씩 올려 놓고, 버튼을 클릭하면 '대기' 라는 글을 TLabel에 보여주다가 모든 쓰레드가 종료되면 '종료'를 표시하게 한다.
먼저 사용할 쓰레드를 TTestThread, 대기 쓰레드를 TWaitThread 라고 명명하고, type 선언부분에 다음과 같이 쓰레드를 정의한다.

```
  TTestThread = class(TThread)
  private
  protected
    procedure Execute; override;
  end;

  TWaitThread = class(TThread)
  private
    procedure UpdateLabel;
  protected
    procedure Execute; override;
  end;
```

TTestThread 를 구현해 보자. 단지 Sleep 기능을 이용하는 것으로 실행을 대신하도록 한다.

```
procedure TTestThread.Execute;
begin
  FreeOnTerminate := True;
  Sleep(10000);
end;
```

이제 핸들 배열과 WaitForMultipleObjects API 함수를 사용해서 대기 쓰레드를 구현한다. 이 부분이 이번 섹션의 핵심 부분이라고 할 수 있다.

```
procedure TWaitThread.UpdateLabel;
begin
  Form1.Label1.Caption := '종료';
end;
```

```
procedure TWaitThread.Execute;
var
  HandleArray: array[0..4] of THandle;
  ThreadArray: array[0..4] of TTestThread;
  i: Integer;
begin
  FreeOnTerminate := True;
  for i := 0 to 4 do
  begin
    ThreadArray[i] := TTestThread.Create(False);
    HandleArray[i] := ThreadArray[i].Handle;
    Sleep(1000);   //쓰레드가 동시에 생성되지 않도록 ...
  end;
  WaitForMultipleObjects(5, @HandleArray, True, INFINITE);
  Synchronize(UpdateLabel);
end;
```

그러면 이제 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := '대기';
  TWaitThread.Create(False);
end;
```

비교적 간단한 예제 이지만, 사용법을 익히는 데에는 큰 무리가 없었을 것으로 생각한다.

임계 섹션(Critical Sections)의 활용

멀티 쓰레드 어플리케이션을 제작하다 보면 여러 개의 쓰레드가 동시에 파일과 같은 리소스에 접근하는 경우가 있게 된다. 이런 경우에 데이터가 깨지거나, 심할 경우 시스템이 멈추는 등의 부작용이 있을 수가 있으므로 이런 상황을 피하도록 해야 한다. 이를 위해서는 하나의 쓰레드가 점령하고 있는 리소스에는 다른 쓰레드가 접근할 수 없도록 할 필요가 있는데, 이럴 때 사용하게 되는 것이 임계 섹션(Critical Section)이다.
임계 섹션을 간단하게 설명하면 여러 개의 쓰레드가 자신의 차선을 지키면서 동시에 도로를 달려오다가, 1 차선으로 좁아지는 병목 구간이 생긴 것으로 생각하면 쉽게 이해할 수 있다.

즉, 리소스를 사용하는 곳에서는 하나의 쓰레드 만이 통과가 가능한 것이다. 이러한 구간을 Win32 API 인 EnterCriticalSection 과 LeaveCriticalSection 함수의 블록으로 설정해서 사용하면 된다. 이렇게 임계 섹션에 들어가려고 대기하고 있는 각각의 쓰레드들은 CPU 시간을 전혀 소모하지 않으므로 효과적이다.
델파이 3 에서는 이들 API 함수를 SyncObjs.pas 유닛에 TCriticalSection 클래스로 구현해 놓았는데, 실제로 구현한 부분을 살펴 보면 Win32 API 를 직접 사용하는 것과 크게 다르지 않으므로 여기에서는 Win32 API 를 직접 사용하는 방법에 대해서 먼저 알아보고, TCriticalSection 클래스를 사용할 때에는 어떻게 변경 되는지 공부하도록 하자.
임계 섹션을 구현하기 위해서는 먼저 메모리에 임계 섹션을 설정해야 한다. 이를 위해서 사용하는 API 함수가 InitializeCriticalSection 이라는 프로시저로 이 프로시저의 선언부는 다음과 같다.

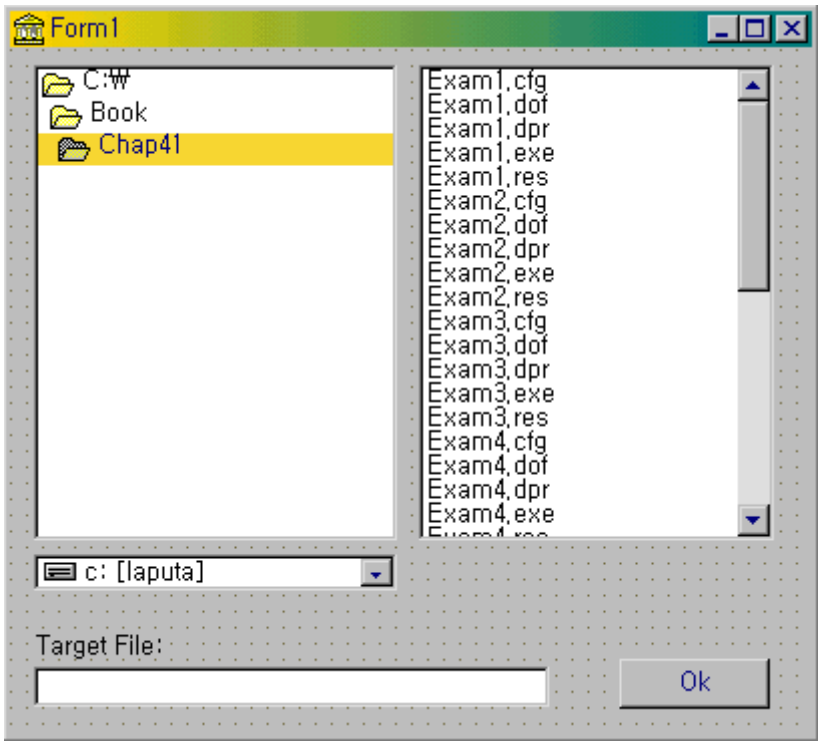
```
procedure InitializeCriticalSection(var lpCriticalSection: TRTLCriticalSection); stdcall;
```

파라미터로 TRTLCriticalSection 형의 변수를 가지게 되어 있는데, 이 데이터 형은 정의된 임계 섹션에 대한 정보를 가지고 있는 레코드 형이다. 그러나, 실제로 이 레코드에 직접 접근하는 경우는 거의 없다.
이렇게 InitializeCriticalSection 프로시저를 이용해서 임계 섹션을 초기화 했으면 나중에는 결국 DeleteCriticalSection 프로시저를 이용해서 사용한 리소스를 해제해 주어야 한다. 이 프로시저 역시 InitializeCriticalSection 과 마찬가지로 TRTLCriticalSection 데이터 형의 변수를 파라미터로 가진다.
보통 이들 프로시저는 각 유닛의 initialization 과 finalization 섹션에 설정해서 사용하게 된다. TRTLCriticalSection 형의 변수 CritSect 를 사용한다고 가정하면 다음과 같이 각 유닛에 설정해서 사용한다.

```
initialization
  InitializeCriticalSection(CritSect);

finalization
  DeleteCriticalSection(CritSect);
```

임계 섹션과 쓰레드 대기 처리 방법을 동시에 익히기 위해서 예제를 하나 만들어 보자. 이번 예제는 두개의 파일을 붙여서 하나의 파일로 합쳐주는 프로그램이다. 예를 들어, 텍스트 파일 같은 경우에 파일을 붙여서 하나로 만들어야할 경우가 생기는데 이를 멀티 쓰레드 기법과 임계 섹션을 이용해서 구현하는 것이다.
이를 위해서 폼에 드라이브와 디렉토리, 파일을 선택할 수 있도록 TDriveComboBox,

TDirectoryListBox, TFileListBox 컴포넌트를 올려 놓고, TEdit 컨트롤에 목적 파일을 적어넣을 수 있도록 한다. 그리고, 작업을 실행시킬 버튼을 추가한다.
폼의 모습은 다음과 같다.



그러면, 실제로 쓰레드가 어떻게 작동할 것인지 생각해보자. 두개의 쓰레드가 소스 파일에 접근하는 것이 허용된다. 그렇지만 하나의 목적 파일에 쓸 때에는 오직 하나의 쓰레드만 접근할 수 있어야 한다. 그리고, 이렇게 파일에 쓰고 있는 동안에는 그 작업이 끝날 때까지 다른 쓰레드는 대기하도록 해야 한다. 그러므로, 파일에 쓰는 작업을 임계 섹션 블록으로 지정하면 동시에 접근하는 일은 없어진다. 이렇게 멀티 쓰레드를 사용하면 하나의 쓰레드가 파일에 쓰고 있을 때, 다른 쓰레드는 파일의 내용을 읽어오는 등의 작업을 동시에 할 수 있다.
실제로 이 예제에서 사용할 쓰레드는 파일을 복사하는데 사용할 TCopyThread 와 두 개의 TCopyThread 가 실행될 때 이들이 하나의 파일로 복사될 때 사용자 인터페이스가 멈추기 않기 위해서 이들 쓰레드가 완료될 때까지 대기하는 TMasterThread 로 구성된다.
TCopyThread 에서는 임계 섹션을 사용해서 파일에 쓰기 작업을 할 때 하나의 쓰레드만 접근하도록 할 것이며, TMasterThread 에서는 두 개의 TCopyThread 핸들 배열을 이용해서 쓰레드 대기 처리를 할 것이다.
먼저 type 선언문에 TMasterThread 와 TCopyThread 를 다음과 같이 선언하고, 전역변수로 사용할 CritSect, Files, NoFile 변수를 var 절에 추가한다.

```
  TMasterThread = class(TThread)
  private
    FTarget : String;
  protected
    procedure Execute; override;
  public
    constructor Create(TargetFile: String);
    function WaitForMultipleThreads(const ThreadArray: array of TThread;
      TimeOutVal: DWORD): Word;
  end;

  TCopyThread = class(TThread)
  private
    FTarget: String;
  protected
    procedure Execute; override;
  public
    constructor Create(Target: String);
end;

var
  Form1: TForm1;
  CritSect: TRTLCriticalSection;
  Files: TStrings;
  NoFile: Boolean;
```

앞에서 NoFile 전역 변수는 파일 리스트 박스에서 파일이 선택되었는지 여부와 쓰레드에서 작업을 수행한 후 이 작업이 완료되었는지 등을 검사할 때 사용하게 된다. 이 변수와 임계 섹션을 초기화하기 위해서 다음과 같이 initialization, finalization 섹션의 코드를 작성한다.

```
initialization
  InitializeCriticalSection(CritSect);
  NoFile := False;

finalization
  DeleteCriticalSection(CritSect);
```

TMasterThread 를 구현해보자. 이 쓰레드의 Create 메소드를 다음과 같이 구현해서, 두 개의 파일을 합칠 파일의 이름이 TargetFile 파라미터로 넘어오면 이를 FTarget 필드에 저장한다.

```
constructor TMasterThread.Create(TargetFile: String);
begin
  FTarget := TargetFile;
  FreeOnTerminate := True;
  inherited Create(False);
end;
```

그리고, TCopyThread 를 이용해서 작업을 수행하고 이들 작업이 끝날 때까지 기다리는 Execute 메소드는 다음과 같이 구현한다.

```
procedure TMasterThread.Execute;
var
  Thread1, Thread2: TCopyThread;
  i: Integer;
begin
  Thread1 := TCopyThread.Create(FTarget);
  Thread2 := TCopyThread.Create(FTarget);
  WaitForMultipleThreads([Thread1, Thread2], INFINITE);
end;
```

즉, 쓰레드를 생성하고 이들을 직접 내부적으로 정의한 WaitForMultipleThrads 메소드를 이용해서 파라미터로 넘겨주면 이들 쓰레드의 실행이 모두 완료될 때까지 대기한다. 이를 적당하게 변경하면 두 개의 파일 뿐만 아니라 여러 개의 파일을 붙일 수 있도록 확장할 수 있을 것이다.
TMasterThread 의 핵심적인 부분인 WaitForMultipleThreads 메소드는 다음과 같이 구현된다.

```
function TMasterThread.WaitForMultipleThreads(const ThreadArray: array of TThread;
  TimeOutVal: DWORD): Word;
var
  Handles: TWOHandleArray;
```

```
  i: Integer;
begin
  for i := 0 to High(ThreadArray) do
    Handles[i] := ThreadArray[i].Handle;
  Result := WaitForMultipleObjects(High(ThreadArray) + 1, @Handles, True,
    TimeOutVal)
end;
```

이 메소드는 WaitForMultipleObjects API 함수를 사용하기 쉽도록 변경한 것으로, TThread 클래스의 배열을 직접 파라미터로 받아서, 이를 핸들 배열의 형태로 변환한 후 API 함수를 호출하게 된다.
이번에는 임계 섹션을 사용해서 파일을 읽고, 쓰는 TCopyThread 클래스를 구현해 보자. 먼저 Create 메소드를 다음과 같이 구현한다.

```
constructor TCopyThread.Create(Target: String);
begin
  FTarget := Target;
  FreeOnTerminate := True;
  inherited Create(False);
end;
```

파라미터로 넘어온 문자열을 필드에 저장하는 역할을 하게 된다.
실제로 파일을 읽고, 쓰는 Execute 메소드는 다음과 같이 구현한다. 여기에서는 TFileStream 을 이용해서 파일 조작을 했는데, 자세한 설명은 다른 장에서 다루게 되므로 생략하고 임계 섹션을 다룬 부분을 유심히 살펴보도록 한다. 이 쓰레드에서는 두 개의 임계 섹션 블록이 사용된다.

```
procedure TCopyThread.Execute;
var
  FileName: String;
  sStream, dStream: TFileStream;
  pBuf: Pointer;
  cnt, bufSize : LongInt;
  FName: String;
  i: Integer;
begin
```

```
  FileName := '';
  if not NoFile then
  repeat
    try
      if Assigned(Files) and (Files.Count > 0) then
      begin
        EnterCriticalSection(CritSect);
        FileName := Files[0];
        Files.Delete(0);
        LeaveCriticalSection(CritSect);
      end
      else Break;
      if (FileName <> '') then
        sStream := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite)
      else Break;
      bufSize := sStream.Size;
      try
        GetMem(pBuf, bufSize);
        cnt := sStream.Read(pBuf^, bufSize);
        EnterCriticalSection(CritSect);
        if FileExists(FTarget) then
          dStream := TFileStream.Create(FDest, fmOpenReadWrite)
        else dStream := TFileStream.Create(FDest, fmCreate);
        dStream.Seek(0, soFromEnd);
        cnt := dStream.Write(pBuf^, cnt);
        LeaveCriticalSection(CritSect);
      finally
        FreeMem(pBuf, bufSize);
        dStream.Free;
      end;
    finally
      sStream.Free;
    end;
  until ((not Assigned(Files)) or (Files.Count = 0));
  NoFile := True;
end;
```

다소 길지만, 대부분은 파일을 다루기 위한 코드이다. 먼저 앞에서 사용하는 NoFile 전역 변수는 선택된 파일이 있을 경우에 이 쓰레드를 실행하기 위해서 사용되는 Boolean 형의 변수이다. 이 변수가 False 이면 파일이 선택되어 있는 것이다. Files 변수는 합치는 대상이 될 파일 들을 저장하는 TStrings 형의 변수로 이 변수에 있는 모든 파일을 읽고, 쓰면 이 쓰레드가 종료되고 NoFile 변수는 True, Files 변수의 아이템은 모두 삭제 된다.
파일 이름을 설정하고, Files 변수의 아이템을 삭제하는 부분에 첫번째 임계 섹션 블록이 설정되었다. 여기에 임계 섹션이 설정된 이유는 이렇게 하지 않을 경우, 다른 쓰레드가 Files 전역 변수에 접근해서 아이템을 삭제하거나 하면 예기치 않은 결과가 나타나기 때문이다. 이와 같이 파일과 같은 입출력 작업 이외에, 전역 변수에 대한 작업을 할 때에도 임계 섹션을 잘 활용해야 한다.
그 뒤로 나오는 여러 줄을 파일을 읽어와서 목적 파일에 쓰는 작업을 하는 코드로 자세한 설명은 생략하겠다. 두번째 임계 섹션 블록이 설정되어 있는 부분을 주목하자. 이 부분의 코드는 다음과 같다.

```
EnterCriticalSection(CritSect);
  if FileExists(FTarget) then
    dStream := TFileStream.Create(FTarget, fmOpenReadWrite)
  else dStream := TFileStream.Create(FTarget, fmCreate);
  dStream.Seek(0, soFromEnd);
  cnt := dStream.Write(pBuf^, cnt);
LeaveCriticalSection(CritSect);
```

즉, 읽어온 파일을 쓰게 될 파일에 대한 부분을 임계 섹션 블록으로 설정해서 다른 쓰레드의 접근을 막는 것이다.
마지막으로, 버튼을 클릭했을 때 파일 리스트 박스에 선택된 파일 이름 들을 이용해서 TMasterThread 를 호출하는 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender : TObject);
var
  i : Integer;
  Thread : TMasterThread;
begin
  Files := TStringList.Create;
  with FileListBox1 do
    for i := 0 to Items.Count - 1 do
```

```
      if Selected[i] then
        Files.Add(Items[i]);
  Thread := TMasterThread.Create(Edit1.Text);
end;
```

정 리 (Summary)

멀티 쓰레드 환경은 이미 대부분의 어플리케이션이 지원하고 있는 기본 사양이 되어 가고 있는 추세이다. 그럼에도 불구하고 의외로 멀티 쓰레드를 지원하게 할 때 만날 수 있는 어려움이 꽤 많다.

이번 장에서 다룬 임계 섹션 이외에도 뮤텍이나 세마포어 같이 멀티 쓰레드 프로그래밍을 할 때의 동기화 문제가 특히 중요한데, 여기에 대해서는 MSDN 등의 내용을 통해 더욱 깊은 내용을 알아두기를 권하고 싶다.