3장 Win32 API

* 객체 - 이전과 지금
* 멀티태스킹과 멀티스레딩
* Win32의 메모리 관리
* Win32의 에러 처리

이번 장에서는 Win32 API와 Win32 운영체제에 대한 일반적인 내용을 다룬다. Win32 운영체제의 기능과 함께 16비트 환경과의 차이점에 대해서도 이야기하겠다. 이번 장의 목적은 Win32 환경이 어떻게 작동하는지에 대한 기본적인 지식을 제공하는 것이지, Win32 운영체제 전체를 세세하게 다루는 것이 아니라는 점을 기억하기 바란다. Win32가 어떻게 작동하는지 이해하고 있어야 델파이에서 Win32 API를 제대로 활용할 수 있다.

객체 - 이전과 지금

객체라는 말은 문맥에 따라 여러 가지 의미를 가진다. Win32 구조에 대해 이야기를 할 때 나오는 객체라는 개념은 객체 지향 프로그램이나 COM을 말할 때의 객체가 아니다. 심지어, 16비트 윈도우즈를 말할 때와 Win32에서의 객체도 약간 다른 의미를 가진다. 따라서, 우선 Win32의 객체란 어떤 것인지를 명확하게 이해하고 있어야 한다.

Win32 환경에는 기본적으로 두 종류의 객체, 즉 커널 객체들과 GDI/USER 객체들이 있다.

커널(Kernel) 객체들

커널 객체란, Win32 운영체제의 핵심을 이루며, 커널 객체에는 이벤트나 파일 맵핑, 파일, 메일슬롯, 뮤텍스(mutex), 파이프, 프로세스, 세마포어(semaphore), 스레드 등이 포함된다. Win32 API에는 각 커널 객체에 사용하는 여러 함수들이 있다. 커널 객체에 대한 일반적인 내용을 이야기하기 전에 Win32 환경에서 객체들이 어떻게 다루어지는지에 대한 기본적인 지식부터 갖추도록 하자.

프로세스

프로세스(process)란, 실행중인 프로그램 또는 프로그램 인스턴스라고 할 수 있다. Win32 환경은 멀티태스킹 환경이므로 동시에 여러 개의 프로세스들이 떠 있을 수

있다. 각각의 프로세스는 코드와 데이터를 저장할 4GB의 주소 공간을 할당받는다. 프로세스는 그 4GB 공간 안에서 메모리 할당이나 스레드, 파일 맵핑 등을 처리한다. 프로세스가 적재한 동적 연결 라이브러리(DLL)도 그 공간 안에 자리를 잡는다. Win32 운영체제의 메모리 관리에 대해서는 이번 장의 "Win32의 메모리 관리" 부분에서 다시 이야기하겠다.

프로세스는 자활력이 없다. 다른 말로 하자면, 프로세스는 아무 것도 수행하지 않는다. 코드를 수행하는 것은 스레드이다. 각각의 프로세스는 주 스레드(primary thread)를 하나 할당받으며, 주 스레드는 자신이 속한 프로세스의 문맥 안에서 코드를 수행한다. 하나의 프로세스는 여러 개의 스레드를 가질 수 있으나, 주 스레드는 단 하나만 존재할 수 있다.



스레드(thread)란, 어떤 프로세스 안의 특정한 코드 실행 단위를 의미하는 운영체제의 객체이다. 모든 Win32 애플리케이션은 적어도 하나의 스레드를 가지며, 그 스레드를 주 스레드(primary 또는 default thread)라고 한다. 물론, 애플리케이션은 하나 이상의 스레드를 가질 수 있다. 스레드에 대해서는 11장, "멀티스레드 애플리케이션의 작성"에서 자세히 이야기하겠다.

프로세스가 생성되면 Win32 운영체제는 그 프로세스에 주 스레드를 할당한다. 기본 스레드는 필요에 따라 다른 스레드를 생성하기도 한다. Win32 운영체제는 프로세스의 스레드에 CPU 수행 시간을 할당하는데, 그것을 타임 슬라이스(time slice)라고 부른다.

표 3.1은 프로세스에 관련된 Win32 API 함수들이다.

표 3.1 프로세스 함수들

함 수	용 도
CreateProcess()	새 프로세스와 기본 스레드를 만든다. 이 함수는 윈도우즈 3.1의 WinExec()를 대신하는 것이다.
ExitProcess()	현재 프로세스를 종료한다. 그 프로세스에 관련된 모든 스레드들도 종료된다.
GetCurrentProcess()	현재 프로세스에 대한 가상 핸들(pseudohandle)을 얻는다. 가상 핸들이란, 현재 프로세스 핸들로 해석할 수 있는 특별한 핸들이다. 실제 핸들을 얻으려면 DuplicateHandle() 함수를 사용한다.
DuplicateHandle()	커널 객체의 핸들을 복제한다.
GetExitCodeProcess()	현재 프로세스 ID를 얻는다. 프로세스 ID란, 운영체제가 프로세스에 붙인 고유한 식별자이다. 프로세스가 종료되면 그 프로세스의 ID도 사라진다.
GetPriorityClass()	인자로 지정된 프로세스의 우선순위 등급을 얻는다. 이 함수로 얻은 값과 그 프로세스의 각 스레드들의 우선순위 값이 그 프로세스의 각 스레드들의 기본 우선순위 등급을 결정한다.
GetStartupInfo()	TStartupInfo 구조체의 내용을 얻는다. TStartupInfo 구조체는 프로세스가 생성될 때 초기화된다.
OpenProcess()	인자로 지정한 프로세스 ID에 해당하는 프로세스의 핸들을 얻는다.
SetPriorityClass()	프로세스의 우선순위 등급을 설정한다.
TerminateProcess()	프로세스를 강제로 종료하고, 그 프로세스에 관련된 모든 스레드들을 제거한다.
WaitForInputIdle()	프로세스가 사용자로부터의 입력을 기다리는 상황이 될 때까지 실행을 지연한다.

몇몇 Win32 API 함수들은 애플리케이션의 인스턴스 핸들을 인자로 사용하며, 어떤 것들은 모듈 핸들을 사용한다. 16비트 윈도우즈에서는 인스턴스 핸들과 모듈 핸들이 다른 값을 가졌지만, Win32에서는 그렇지 않다. 모든 프로세스는 고유한 인스턴스 핸들을 가진다. 델파이 5에서 애플리케이션의 인스턴스 핸들은 전역 변수 HInstance이다. HInstance와 애플리케이션의 모듈 핸들이 동일하므로 지정된 모듈의 파일 이름을 돌려주는 함수인 GetModuleFileName() 함수처럼 모듈 핸들을 요구하는 Win32 API 함수에 HInstance를 넣어도 문제가 생기지 않는다. HInstance가 애플리케이션의 모듈 핸들을 가리키지 않는 경우에 대해서는 다음 주의를 참고하기 바란다.



패키지 안으로 컴파일된 코드의 경우, HInstance가 애플리케이션의 모듈 핸들을 의미하지 않는다. 호스트 애플리케이션의 모듈 핸들을 원한다면 항상 MainInstance를 사용하고, 코드가 들어 있는 모듈의 핸들을 원한다면 HInstance를 사용해야 한다.

Win32와 16비트 윈도우즈의 또 하나의 차이는 HPrevInst 전역 변수이다. 16비트 윈도우즈 환경에서 HPrevInst는 이전에 실행된 동일한 애플리케이션의 인스턴스를 의미한다. HPrevInst는 주로 중복 실행 방지, 즉 한 애플리케이션이 두 번 실행되지 않도록 하는 데 사용했었지만, Win32 환경에서는 더 이상 HPrevInst를 중복 실행 방지에 써먹을 수 없다. Win32에서 각각의 프로세스는 독립적인 4GB의 주소 공간을 할당받으며, 다른 프로세스의 상태는 알지 못한다. 따라서, HPrevInst의 값은 항상 0이 된다. Win32 환경에서 중복 실행 방지를 구현하는 방법은 13장, "고난도 기법들"에서 제시하겠다.

나머지 커널 객체들

그 외에도, 여러 가지 커널 객체들이 있다. 생성된 커널 객체는 프로세스의 주소 공간에 존재하며, 프로세스는 그 커널 객체에 대한 핸들을 얻는다. 그 핸들은 다른 프로세스에 넘겨줄 수도 없고, 다음에 생성되는 프로세스가 동일한 핸들을 다시 사용할 수도 없다. 그러나 적절한 Win32 API 함수를 이용하면 두 번째 프로세스가 이미 존재하고 있는 커널 객체의 핸들을 얻을 수 있다. 예를 들어서, CreatMutex() Win32 API 함수는 이름 붙은(named) 또는 이름 없는(unnamed) 뮤텍스 객체를 만들고, 그 객체의 핸들을 돌려준다. 그리고 OpenMutex() Win32 API 함수는 인자로 지정된 이름에 해당하는 뮤텍스 객체의 핸들을 돌려준다. 즉, 기존의 '이름 붙은' 뮤텍스 객체의 핸들을 얻을 수 있는 것이다.



이름 붙은 커널 객체들에는 문자 그대로 이름이 붙어 있는데, 이름은 널 종료 문자열이다. 이름 붙은 커널 객체는 해당 CreateXXX() 함수들로 만들며, 객체의 이름은 Win32 운영체제에 등록된다. 그 커널 객체를 만든 프로세스 이외의 프로세스에서도 OpenXXX() 함수의 인자에 객체의 이름을 넣어서 객체의 핸들을 얻을 수 있다.

하나의 뮤텍스 객체를 여러 프로세스들이 공유하게 하는 과정은 다음과 같다. 우선, 첫 번째 프로세스에서 CreateMutex() 함수로 뮤텍스 객체를 만든다. 이때 미리 정해진 이름을 제공해야 한다. 일단 뮤텍스가 만들어지면 다른 프로세스에서는 동일한 이름을 OpenMutex()의 인자에 넣어서 뮤텍스 객체의 핸들을 얻은 후 사용하면 된다. 그런데 프로세스들 사이에서 커널 객체를 공유할 때에는 보안 관련 설정이 필요할 수 있다. CreateMutex() 함수에는 보안 관련 인자가 있는데, 보안 관련 인자의 여러 설정에 대해서는 Win32 도움말이나 문서를 참고하기 바란다.

GDI와 USER 객체

16비트 윈도우즈의 객체들은 핸들을 이용해서 조작했었다. 그러나 16비트 윈도우즈에서는 커널 객체란 것이 존재하지 않았으므로 커널 객체에 대한 핸들도 존재하지 않았다.

16비트 윈도우즈에 존재하는 객체는 두 종류이다. 하나는 GDI와 USER 지역 힙(heap)에 저장되는 객체들이고, 또 하나는 전역 힙으로부터 할당되는 객체들이다. GDI 객체란, 브러시, 펜, 폰트, 팔레트, 비트맵, 영역(region) 등 그래픽 표시에 관련된 것들을 말하며, USER 객체란 윈도우, 윈도우 클래스, 아톰(atom), 메뉴 등 사용자 인터페이스에 관련된 것들을 말한다.

객체와 핸들은 직접적인 연관 관계를 가지고 있다. 객체의 핸들은 객체의 실제 모습을 담고 있는 구조체를 가리키는 포인터로 변환된다. GDI 객체의 구조체는 GDI의 기본 데이터 세그먼트에, 그리고 USER 객체의 구조체는 USER의 기본 데이터 세그먼트에 존재한다. 그리고 전역 힙에 저장된 객체의 핸들은 전역 메모리 세그먼트의 특정 메모리 블록을 가리키는 포인터로 변환된다.

이처럼 16비트 윈도우즈에서는 프로세스별로 메모리 공간이 분리되지 않으므로 객체들을 서로 공유할 수 있다. 16비트 윈도우즈에서 객체의 핸들은 LDT(Local Descriptor Table)에 저장되며, 이 테이블은 모든 프로세스가 참조할 수 있다. 그리고 GDI와 USER의 기본 데이터 세그먼트 역시 16비트 윈도우즈의 모든 프로세스, 즉 모든 애플리케이션과 DLL들이 접근할 수 있다. 결론적으로 말하면, 어떤 애플리케이션이나 DLL에 속한 객체를 다른 애플리케이션이나 DLL이 사용할 수 있다는 뜻이다. LDT 같은 객체들은 오직 윈도우즈 3.1에서만 공유가 가능하다는 점을 기억하기 바란다. 이러한 16비트 윈도우즈 구조는 여러 목적으로 써먹을 수 있는데, 대표적인 것이 애플리케이션들 사이에서 메모리를 공유하는 것이다.

Win32는 GDI와 USER 객체들을 다른 방식으로 다루기 때문에 16비트 윈도우즈 애플리케이션에서 사용했던 기법들이 Win32 환경에서 작동하지 않는 경우가 많다.

우선, Win32에는 커널 객체란 것이 도입되었는데, 이에 대해서는 앞에서 이야기했다. 그리고 GDI 객체와 USER 객체의 구현 방식도 다르다.

Win32에서 GDI 객체들은 공유할 수 없다. GDI 객체들은 더 이상 전역 메모리 공간에 존재하지 않으며, 프로세스마다 독립적으로 존재한다(각각의 프로세스는 분리된 4GB의 주소 공간을 가진다). 또한, 16비트의 핸들은 전역 LDT에 저장되는 반면, Win32에서는 프로세스마다 핸들 테이블이 따로 있다. 따라서, 같은 번호의 핸들이라도 프로세스가 다르면 다른 객체를 의미하며, 다른 프로세스에 GDI 객체 핸들을 넘겨주는 것은 아무 의미가 없다.

16비트 윈도우즈에서 LDT는 모든 애플리케이션이 사용할 수 있다고 했다. 그러나 Win32에서는 프로세스마다 자신만의 LDT가 있다. 즉, 프로세스마다 지역 LDT가 있는 것이다.



C++ 프로그래머라면 SelectObject()를 지겹게도 많이 사용했을 것이다. Win32 환경에서 SelectObject()의 인자로 다른 프로세스의 GDI 객체 핸들을 지정해도 호출이 실패하지 않을 수 있다. 그러나 그 핸들이 실제로 어떤 객체를 의미하는지는 프로세스마다 다르다는 점을 꼭 기억하기 바란다.

GDI 핸들은 Win32의 GDI 서브운영체제가 관리하는데, GDI 서브운영체제가 수행하는 관리 작업에는 GDI 객체의 유효성 검사나 핸들의 재활용 등이 포함된다.

USER 객체 역시 GDI 객체와 동일한 방법으로 작동하며, Win32 USER 서브운영체제가 관리한다. 그러나 GDI와는 달리, USER 객체의 핸들 테이블은 프로세스별로 관리되지 않고, USER 자신이 관리한다는 점이 중요하다. 따라서, 윈도우나 윈도우 클래스, 아톰 같은 USER 객체들은 프로세스들 사이에서 공유할 수 있다.

멀티태스킹과 멀티스레딩

멀티태스킹(multitasking)이란, 여러 개의 애플리케이션을 병렬적으로 실행시킬 수 있는 운영체제의 능력을 의미하는 말이다. 병렬적이라고는 하지만, 실제로는 CPU의 수행 시간을 아주 잘게 나누어서(타임 슬라이스) 각각의 애플리케이션을 번갈아서 수행하는 것이다. 따라서, 멀티태스킹이라는 용어보다는 태스크 스위칭(task switching), 즉 작업 전환이라고 하는 것이 더 정확하다. 상식적으로 생각해도 하나의 CPU가 동시에 두 개 이상의 프로그램을 돌릴 수는 없다. 짧은 시간 간격으로 여러 프로그램을 조금씩 나누어서 실행하는 것이 현재의 멀티태스킹이며, 사용자의 입장에서는 작업 전환의 시간 간격이 너무나 짧기 때문에 여러 애플리케이션이 동시에 병렬적으로 실행되는 것처럼 보인다.

멀티태스킹은 유닉스 이전 시절에서부터 존재한 기술이며, 16비트 윈도우즈들도 멀티태스킹을 지원했다. 그러나 이전 버전의 윈도우즈들과 Win32 운영체제는 중요한 차이가 있다. Win32는 선점형(preemptive) 멀티태스킹을 사용하는 반면, 이전 버전들은 비선점형(nonpreemptive) 멀티태스킹을 사용한다.

비선점형 멀티태스킹이란, 윈도우즈 운영체제가 애플리케이션의 수행 시간을 관리할 때 시스템 타이머를 사용하지 않는다는 뜻이다. 다시 말하면, 윈도우즈는 애플리케이션의 수행 권리를 강제로 빼앗을 수 없다. 애플리케이션이 프로그램의 제어권을 윈도우즈에게 돌려줘야만 윈도우즈는 다른 애플리케이션에게 CPU를 사용할 권한을 넘겨줄 수 있는 것이다. 윈도우즈 3.1에서 하나의 애플리케이션이 다운되면 윈도우즈 전체가 다운되는 일이 많았던 이유도 바로 비선점형 멀티태스킹 때문이었다.

Win32의 경우, 운영체제가 각 프로세스의 스레드들에 CPU 시간을 할당한다. 스레드에 할당되는 시간은 스레드의 우선순위에 따라 다르다. 스레드의 시간 할당에 대해서는 11장에서 좀더 자세히 이야기하겠다.



윈도우즈 9x와는 다른 Win32 운영체제인 윈도우즈 NT는 하나 이상의 CPU가 장착된 하드웨어 시스템을 지원한다. 즉, 진정한 의미의 멀티태스킹이 가능한 것이다. NT는 각 CPU들의 상태를 관리해서 적절히 CPU의 시간을 각각의 스레드에 할당한다.

멀티스레딩(multithreading)은 하나의 애플리케이션 안에서 멀티태스킹이 일어나는 것이라고 생각하면 된다. 다시 말해서, 하나의 애플리케이션이 병렬적으로 여러 개의 작업을 진행할 수 있다는 뜻이다. 하나의 프로세스는 여러 개의 스레드를 가질 수 있으며, 각각의 스레드는 다른 스레드와는 분리된 코드를 수행한다. 그러나 한 스레드의 작업이 다른 스레드의 작업 결과를 참조해야 할 경우가 있는데, 각 스레드의 동작을 서로 조절하는 것을 "동기화(synchronization)"라고 한다. 예를 들어서, 세금 계산이 끝나지 않으면 월급 총액을 계산할 수가 없다. 스레드 동기화에는 메모리나 보안 관련 문제까지 겹치기 때문에 스레드 동기화가 가장 어렵다고 말하는 개발자들도 많다. 스레드에 대해서는 11장에서 다시 자세히 이야기하겠다.

Win32의 메모리 관리

Win32 환경은 이전 버전들과는 달리, 32비트 플랫(flat) 메모리 모델을 사용한다. 도스 시절부터 파스칼을 써왔던 독자라면 다음 코드처럼 커다란 배열 선언이 컴파일 에러를 내지 않는다는 것이 얼마나 멋진 일인지 알 것이다.

```
BigArray = array[1..1000000] of Integer;
```

그럼, Win32의 메모리 모델은 어떤 것인지, 그리고 Win32 운영체제에서 메모리 관리는 어떻게 일어나는지 알아보자.

플랫 메모리 모델이란

16비트의 세계에는 세그먼트(segment)라는 것이 있었다. 세그먼트 메모리 모델에서는 주소를 세그먼트:오프셋(segment:offset) 형태로 표현했다. 어떤 메모리 주소를 표현할 때 그 메모리가 속한 기본 주소가 세그먼트이고, 오프셋은 기본 주소로부터 해당 메모리 주소까지의 바이트 수이다. 아파트의 동:호수나 학교의 학년:반을 연상하면 조금 도움이 될 것이다. 세그먼트:오프셋 방식의 가장 큰 문제는 커다란 메모리 공간을 다룰 때 상당히 불편하다는 점이다. 세그먼트:오프셋 방식에서 64KB 이상의 구조체를 다루는 것은 대단히 골치아픈 일이기 때문에 C나 어셈블리 개발자들은 웬만하면 64KB 이상의 자료 구조를 다루지 않고 넘어가는 방법을 찾았었다.

플랫 메모리 모델의 경우, 그러한 제한은 없다. 각각의 프로세스는 자신만의 4GB 주소 공간을 가지며, 그 안에서 자유롭게 커다란 데이터를 마음대로 다룰 수 있다. 그리고 4GB 공간은 그야말로 평평한(flat) 공간이므로 플랫 메모리 주소는 실질적인 메모리의 위치를 의미한다(물론, 물리적인 메모리 위치를 뜻하는 것은 아니다).

Win32 운영체제의 메모리 관리 방식

4GB의 메모리를 장착한 PC는 거의 없다. Win32 운영체제가 실제 램 크기보다 더 큰 메모리 공간을 다룰 수 있다는 점은 이미 알고 있을 것이다. 32비트의 주소 공간은 물리적 메모리 공간을 의미하지 않는다. 대신, Win32는 가상 메모리라는 개념을 사용한다.

가상 메모리를 사용하면 각각의 프로세스마다 분리된 4GB의 가상 주소 공간을 할당할 수 있다. 그 주소 공간의 상위 2MB 영역은 윈도우즈가 사용한다. 하위 2MB 공간은 애플리케이션이 존재하는 곳이며, 애플리케이션은 그 곳의 메모리를 할당해서 사용한다. 이런 구조이기 때문에 한 프로세스의 스레드는 다른 프로세스의 메모리를 사용할 수 없다는 장점이 생긴다. 한 프로세스의 $54545454라는 주소가 가리키는 메모리는 다른 프로세스의 $54545454가 가리키는 메모리와 전혀 별개의 위치이다.

프로세스가 4GB의 공간을 가진다는 말은 프로세스가 실제로 4GB의 메모리를 가진다는 뜻이 아니라, 4GB까지의 주소 범위를 다룰 수 있는 능력이 있다는 뜻이다. 프로세스에 실제로 할당되는 메모리 양은 PC의 물리적 램의 크기와 하드 디스크의 페이지 파일(가상 메모리로 사용하는 영역) 크기에 따라 결정된다. 한 페이지 크기는 하드웨어 시스템에 따라 다른데, 인텔 플랫폼의 경우는 4KB이고, 알파 플랫폼에서는 8KB이다. 이미 몰락해 가는 파워 PC나 MIPS 플랫폼도 4KB를 사용한다. 운영체제는 필요에 따라 페이지 파일(하드 디스크)로부터 페이지들을 읽어서 물리적 메모리에 복사하거나, 물리적 메모리의 페이지들을 페이지 파일에 복사한다. 프로세스의 가상

주소를 물리적 주소로 변환할 때에는 페이지 맵(page map)을 참조하는데, 더 이상 자세히 들어가는 것은 이 책의 성격과 맞지 않는 것 같으니 이 정도로 그치겠다. 개발자가 다룰 수 있는 Win32 환경의 메모리 종류는 크게 세 가지이다. 첫 번째는 가상 메모리이고, 두 번째는 파일 맵핑 객체, 그리고 세 번째는 힙이다.

가상 메모리

Win32는 프로세스의 가상 메모리를 다룰 수 있는 여러 가지 저수준 함수들을 제공한다. 가상 메모리는 다음 세 가지 중 하나의 상태를 가진다.

- Free(해제) : 메모리가 보존되거나 수행될 수 있는 상태
- Reserved(예약됨) : 이후 사용을 위해 예약된 상태. 다른 할당 요청이 있어도 이 상태의 메모리 공간은 할당되지 않는다. 그러나 이 상태의 메모리는 실제 물리적 메모리와 연결되지 않았기 때문에 아직 프로세스가 사용할 수는 없다. Commited가 되어야 실제 물리적 메모리 공간과 연결된다. 메모리 공간을 예약하는 함수는 VirtualAlloc()이다.
- Commited(허용됨) : 메모리가 할당되었으며, 물리적 메모리와 연결된 상태. 허용된 메모리는 프로세스가 사용할 수 있다. 메모리를 허용하는 것도 VirtualAlloc()이다.

2장에서도 나왔었지만, 표 3.2에 나열된 Win32의 VirtualXXXX() 함수들은 가상 메모리를 다루는 함수들이다. 각 함수에 대한 구체적인 정보는 온라인 도움말을 참고하기 바란다.

표 3.2 가상 메모리 함수들

함 수	용 도
VirtualAlloc()	프로세스의 가상 주소 공간에서 페이지(메모리의 단위)들을 예약/허용한다.
VirtualFree()	프로세스의 가상 주소 공간에서 할당된 페이지들을 해제하거나 허용된 페이지들을 예약 상태로 만든다.
VirtualLock()	프로세스의 가상 주소 공간 중 특정 영역을 잠근다. 잠겨진 영역은 페이지 파일로 대체(swapping)되지 않는다(즉, 물리적 메모리에 계속 남아 있게 된다). 페이지 오류를 막기 위한 함수이다.
VirtualUnLock()	잠겨진 영역을 다시 푼다. 풀려진 영역은 이후 필요에 따라서 페이지 파일로 대체될 수 있다.
VirtualQuery()	호출한 프로세스의 가상 주소 공간에 있는 페이지들의 영역에 관한 정보를 돌려준다.

다음 페이지에 계속

이전 페이지에 이어서

함 수	용 도
VirtualQueryEx()	인자로 지정한 프로세스의 가상 주소 공간에 있는 페이지들의 영역에 관한 정보를 돌려준다.
VirtualProtect()	호출한 프로세스의 가상 주소 공간에 있는 허용된 페이지 영역의 접근 보호 설정을 변경한다.
VirtualProtectEx()	인자로 지정한 프로세스의 가상 주소 공간에 있는 허용된 페이지 영역의 접근 보호 설정을 변경한다.



위의 표에 나온 xxxEX() 함수들은 다른 프로세스에 대한 디버깅 권한을 가진 프로세스만 호출할 수 있다. 디버거를 만들고자 하는 독자가 아니라면 자세히 알 필요는 없을 것이다.

메모리 맵 파일

메모리 맵 파일(memory-mappde file : 파일 형태를 가지는 메모리 영역, file-mapping object라고도 함)이란, 사용 방법은 디스크의 파일과 같지만 실제 데이터는 메모리에 저장되는 형태의 자료 구조를 말한다. 파일의 전부 또는 일부가 호출한 프로세스의 주소 공간에 저장되는 것이다. 메모리 맵 파일에 대해서는 12장, “파일 다루기”에서 좀더 자세히 이야기하겠다.

힙

힙(heap)이란, 애플리케이션이 사용하는 연속적인 메모리 블록들을 말한다. 힙을 만든 후에는 힙 안에서 더 작은 메모리 블록을 개별적으로 할당할 수 있다. 힙은 동적인 메모리 할당이나 조작을 효율적으로 하기 위한 메모리 구조이다. 힙 메모리는 Win32의 HeapXXXX() 함수들로 다루는데, 표 3.3에 나와 있다. 자세한 사항은 온라인 도움말을 참고하기 바란다.

표 3.3 힙 함수들

함 수	용 도
HeapCreate()	힙 객체를 생성한다. 구체적으로 말하면, 호출한 프로세스의 가상 주소 공간에서 연속적인 블록을 하나 예약하고, 인자로 지정된 만큼의 영역을 물리적 메모리로 할당한다.
HeapAlloc()	힙에서 이동되지 않는 메모리 블록을 할당한다.
HeapReAlloc()	힙에서 메모리 블록을 다시 할당한다. 힙의 크기나 속성을 변경할 때 사용한다.
HeapFree()	HeapAlloc()으로 할당한 메모리 블록을 해제한다.
HeapDestroy()	HeapCreate()로 생성한 힙 객체를 파괴한다.

참고

같은 Win32라도 윈도우즈 NT와 윈도우즈 95/98에는 차이가 있다는 점을 기억하기 바란다. 특히, 보안과 속도면에서 차이가 있다. 예를 들어, 윈도우즈 95의 메모리 관리자는 윈도우즈 NT의 메모리 관리자보다 더 간략하다(윈도우즈 NT는 힙 블록들에 대해 좀더 복잡한 관리를 수행한다). 그러나 윈도우즈 NT의 메모리 관리자가 윈도우즈 98보다 훨씬 더 빠르다고 알려져 있다.

따라서, 메모리 관련 객체들을 다룰 때에는 윈도우즈 NT와 98의 차이를 염두에 두고 있어야 한다. 델파이의 온라인 도움말에는 플랫폼에 따른 함수의 용도 차이가 나온다. 메모리 관련 함수들을 사용할 때에는 항상 도움말을 자세히 살펴보기 바란다.

Win32의 에러 처리

대부분의 Win32 API 함수들은 함수의 수행 성공 여부를 True나 False로 돌려준다. 만일 수행이 성공적이지 못하면(리턴값이 False이면), GetLastError() Win32 API 함수를 이용해서 에러 코드를 확인하고 적절한 처리를 해줘야 한다.



GetLastError()로 에러 코드를 얻을 수 없는 Win32 API 함수들도 있다. 예를 들어서, GDI 함수들 중에는 에러 코드를 설정하지 않는 함수들도 많다.

에러 코드는 스레드별로 적용되므로 GetLastError()는 에러를 발생시킨 스레드의 문맥 안에서 호출해야 한다. 다음은 GetLastError()를 사용하는 예이다.

```
if not CreateProcess(CommandLine, nil, nil, False,
  NOMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo) then
      raise Exception.Create('프로세스 생성중 오류 발생: '
          + IntStr(GetLastError));
```



델파이 5의 SysUtils 유닛에는 시스템 오류를 예외로 변환하는 표준 예외 클래스 유틸리티 함수들이 있다. 예외 클래스 EWin32Error와 Win32Check(), RaiseLastWin32Error가 바로 그것인데, 일일이 API 함수의 리턴값 점검을 하는 것보다는 이들을 이용하는 것이 좋다. 자세한 사항은 델파이의 온라인 도움말을 참고하기 바란다.

앞의 코드는 널 종료 문자열이 담긴 커맨드 라인으로 지정된 프로세스를 생성하는데, 함수 호출이 실패하면 예외를 발생시킨다. 예외 메시지에는 GetLastError()로 얻은 Win32의 에러 코드가 들어 있다. 아주 표준적인 Win32 API 점검 기법이므로 독자의 코드에도 적용시켜 보기 바란다.



일반적으로, GetLastError()가 돌려주는 에러 코드는 에러를 발생시킨 함수의 도움말에 나온다. 예를 들어서, CreateMutex()를 호출할 때 생길 수 있는 에러 코드들은 도움말의 CreateMutex() 항목에서 찾을 수 있다.

요약

이번 장은 Win32 API에 대해서 이야기했다. Win32에서 새로 도입된 커널 객체가 무엇인지, 또 Win32가 어떻게 메모리를 관리하는지 알게 되었을 것이다. 또한, 개발자가 사용할 수 있는 여러 가지 메모리 관리 기법들에 대해서도 배웠다. 일반적인 델파이 애플리케이션을 만들 때에는 이번 장에 나온 내용들이 그리 크게 필요하지는 않다. 그러나 Win32 운영체제와 기능에 대한 기본적인 이해를 가지고 있으면 애플리케이션의 기능을 좀더 다양하고 광범위하게 확장시킬 수 있다. 이번 장을 계기로 좀더 폭넓은 능력을 가진 개발자로 발돋움하기 바란다.

쉬어가는 페이지