델파이 4의 메모리 관리 기법과 메모리 맵 파일 (Memory Management Techniques in Delphi 4 and Memory Mapped File)

마이크로소프트 Win32 API 에서는 각각의 프로세스는 4GB 까지 메모리를 가질 수 있는 32 비트 가상 주소 공간을 가진다. 하위 2GB 메모리($0~$7FFFFFFF)는 사용자가 사용할 수 있으며, 상위 2GB 메모리($80000000~$FFFFFFFF)는 커널에 의해 예약되어 있다. 프로세스에 의해 사용되는 가상 주소는 메모리에서의 객체의 실제 물리적인 위치를 나타내는 것은 아니다. 커널이 관리하는 각 프로세스는 가상 주소를 물리적 주소로 변환하는 페이지 맵을 관리한다.

가상 주소 공간과 물리적 저장 공간

각 프로세스의 가상 주소는 물리적 메모리인 RAM 의 전체 크기보다 크다. 물리적 저장 공간을 증가 시키기 위해 커널은 디스크 공간을 사용한다. 그러므로, 실행 중인 프로세스 들이 사용할 수 있는 저장 공간의 총량은 RAM 의 크기에 페이지 파일로 쓸 수 있는 디스크 공간의 크기를 합한 것이 된다.
메모리 관리의 유연성을 극대화하기 위해서 커널은 물리적 메모리의 페이지를 디스크의 페이지 파일로 옮길 수 있도록 허용한다. 물리적 메모리가 옮겨지면 커널은 페이지 맵을 재구성하게 된다. 커널이 물리적 메모리에 공간을 필요로 하면, 가장 오래 전에 사용했던 물리적 메모리 공간을 페이지 파일로 옮기게 된다.
프로세스의 가상 주소 공간의 페이지 들은 다음에 설명되는 상태 중 하나에 있게 된다.

1. Free

현재 접근이 가능하지 않지만, 언제든 예약 또는 사용 가능한 페이지이다.

2. Reserved

프로세스의 가상 주소 공간의 블록으로 앞으로 사용하기 위해 예약한 페이지이다. 다른 프로세스가 이 페이지에 접근할 수 없으며, 이 페이지와 연결된 물리적 메모리도 없다. 예약된 페이지는 다른 메모리 할당 작업에 의해 접근할 수 없는 가상 주소의 범위를 정하게 된다. 이런 주소 공간을 예약하기 위해서 프로세스는 VirtualAlloc API 함수를 사용하며, 이

를 해제하기위해서는 VirtualFree 함수를 사용한다.

3. Committed

Committed 페이지는 실제 할당이된 물리적 저장 공간(RAM 또는 디스크)에 해당되는 페이지이다. 이 페이지는 접근이 불가능하게 하거나, 읽기 전용으로 보호할 수 있으며 경우에 따라서는 읽고 쓰기를 자유롭게 허용할 수도 있다. 이렇게 실제 메모리를 할당하기 위해서 역시 VirtualAlloc 함수를 사용할 수 있다. 동시에 읽고 쓰기 접근을 가능하게 할 경우에는 GlobalAlloc, LocalAlloc API 함수를 사용할 수도 있다. 일단 VirtualAlloc 함수로 할당된 페이지는 VirtualFree 함수로 해제할 수 있으며, 이렇게 되면 페이지의 저장 공간은 해제되지만 그 페이지의 상태는 'reserverd'로 바뀌게 된다.

전역과 지역 메모리 관리 함수

프로세스가 메모리를 할당할 때 GlobalAlloc, LocalAlloc 함수를 사용할 수 있다. Win32와 같은 32 비트 선형 메모리 시스템에서는 로컬 힙(heap)과 글로벌 힙이 사실상 구별되지 않기 때문에, 이들 함수에 의해 할당받는 메모리 객체에는 차이가 없다.
이들 함수에 의해 할당받는 메모리 객체는 commited 페이지로, 읽기 쓰기가 모두 가능하다. 이들은 또한 private 메모리로 사용되기 때문에, 다른 프로세스가 접근할 수 없게 된다. GlobalAlloc 함수를 호출할 때에는 GMEM_DDESHARE 라는 플래그를 설정할 수도 있는데, 실제로 메모리 공유가 되지는 않는다. 다만, 이 플래그는 Win16 API 와의 호환성을 위해서 남겨둔 것으로, 일부 어플리케이션에서 DDE 작업의 효율성을 높이기 위해서 사용될 수 있다. 만약에 프로세스 간에 메모리를 공유하고 싶을 경우에는 반드시 파일-매핑 객체를 사용해야 한다.
GlobalAlloc, LocalAlloc 함수를 사용함으로써 32 비트로 표현할 수 있는 어떤 크기의 메모리 블록도 할당받을 수 있다. 다만, 이들이 따로 존재하는 이유는 Win16 과의 호환성을 위해서 이다. 어쨌든 16 비트의 세그먼트로 나뉜 메모리 모델에서 32 비트 가상 메모리 모델로의 변화는 일부의 함수의 기능과 옵션을 의미 없는 것으로 바꾸어 놓았다. 예를 들어, 이제 더 이상 'far'와 같은 예약어를 사용할 필요가 없어졌다.
GlobalAlloc, LocalAlloc 함수 모두 고정되(fixed)거나 이동가능(movable)한 메모리 객체를 할당할 수 있다. 이동가능(movable)한 객체들은 'discardable'로 간주된다. 과거 윈도우 3.1 시절에는 이러한 이동가능한 메모리 객체를 사용하는 것이 메모리 관리 기법으로 중요시 되었다. 이를 이용하면 시스템이 힙의 크기를 절약할 수 있어서, 다른 메모리에 대한 할당 작업을 원활하게 할 수 있었다. 가상 메모리를 이용하면 시스템은 물리적 메모리를 디스크에 있는 페이지 파일로 옮길 수 있는데, 이렇게 옮겨진 부분의 메모리에 대한 가상 메모리 페이지 맵을 수정하는 것으로 쉽게 구현된다. 대신 이동가능한 메모리로 할당하면

시스템이 추가적인 물리적 저장 공간을 필요로 할 때, 가장 오래 전에 사용한, 락(lock)이 걸리지 않은 이동가능한 메모리를 이용하게 할 수 있다. 그러므로, 이동가능한 메모리는 자주 사용되지 않고, 쉽게 다시 생성할 수 있는 메모리의 경우에 사용해야 한다.
고정된 메모리 객체를 할당할 때에는 GlobalAlloc, LocalAlloc 함수가 직접 메모리에 접근할 수 있도록 32 비트 포인터를 반환한다. 그에 비해, 이동가능한 메모리의 경우에는 메모리에 대한 핸들을 반환한다. 만약에 이동가능한 메모리 객체에 대한 포인터를 얻고 싶을 때에는 반드시 GlobalLock, LocalLock 함수를 이용해야 한다. 이들 함수는 메모리를 이동하거나 버릴 수 없도록 고정시키는 역할을 하게 된다. 각 메모리 객체의 내부 데이터 구조에는 0 부터 시작하는 잠금 계수(lock count)가 존재한다. 이동가능한 메모리에서 GlobalLock, LocalLock 함수를 호출하면 이 계수가 하나 증가하며 GlobalUnlock, LocalUnlock 함수를 호출하면 하나 감소한다. 잠긴 메모리는 GlobalReAlloc 이나 LocalReAlloc 함수에 의해 재할당 받지 않는 한 이동과 버림이 불가능하다. 만약 메모리에 대한 UnLock 함수에 의해 잠금 계수가 0 으로 감소하면 그 때부터 이 메모리 객체는 이동과 버림이 가능해진다.
GlobalAlloc 또는 LocalAlloc 함수에 의해 할당받게 되는 메모리의 실제 크기는 요구한 크기보다 클 수도 있다. 이때 실제로 할당받은 메모리의 정확한 바이트 크기를 정할 때 사용되는 API 함수가 GlobalSize, LocalSize 함수이다. 만약에 할당된 크기가 요구한 크기보다 큰 경우에는 프로세스가 이를 모두 사용할 수 있다. GlobalAlloc, LocalAlloc 함수에 의해 할당받은 메모리의 크기와 속성을 변경할 때에는 GlobalReAlloc, LocalReAlloc 함수를 사용할 수 있다. 메모리 객체를 해제할 때에는 공히 GlobalFree, LocalFree 함수가 사용된다. 그밖에 지정된 메모리 객체에 대한 정보를 얻을 때에는 GlobalFlags, LocalFlags 함수가 사용될 수 있으며 지정된 포인터와 연관된 메모리 객체를 알아낼 때에는 GlobalHandle, LocalHandle 함수가 사용된다.

가상 메모리 함수

Win32 API 는 가상 주소 공간에 존재하는 메모리 페이지의 상태를 결정하거나, 관리하는 여러가지 가상 메모리 함수를 제공한다. 많은 수의 어플리케이션이 앞에서 설명한 GlobalAlloc, LocalAlloc 함수를 사용해서 Win16 과의 호환성을 제공하지만 가상 메모리 함수는 앞의 함수 들이 제공할 수 없는 기능을 제공한다. 이 함수들을 이용해서 할 수 있는 작업에는 다음과 같은 것들이 있다.

1. 프로세스의 가상 주소 공간의 범위를 예약한다. 이러한 예약 행위는 물리적 저장 공간을 할당하지 않지만, 지정된 주소 공간 범위에 대해 다른 메모리 할당 작업이 접근할 수 없도록 한다. 이 작업은 다른 프로세스의 가상 주소 공간에 영향을 미치지 않는다. 이렇게 사용할 페이지를 예약함으로써 물리적 저장 공간의 불필요한 소모를 막을 수 있

고, 동적인 데이터 구조가 커질 때에 효과적으로 이용할 수 있게 한다.

2. 예약된 프로세스의 가상 주소 공간의 범위를 실제 물리적 저장 공간에서 사용할 수 있도록 할 수 있다. (Commit)
3. 사용하는 페이지의 접근 속성을 읽기-쓰기, 읽기 전용, 접근 불가로 설정할 수 있다. 이는 앞에서 설명한 GlobalAlloc, LocalAlloc 등의 표준 함수 들이 읽기-쓰기로만 설정되는 것에 비해 다른 점이다.
4. 예약된 페이지의 범위를 해제할 수 있다.
5. Commit 된 메모리 페이지를 decommit 할 수 있다. 이로 인해 물리적 저장 공간이 해제되며, 다른 프로세스에 의해 이들이 사용될 수 있다
6. 하나 이상의 commited 메모리를 물리적 메모리(RAM)로 lock 해서 시스템이 페이지 파일로 스왑하지 않도록 할 수 있다.
7. 페이지에 대한 정보를 얻을 수 있다.

가상 메모리 함수들은 메모리 페이지를 관리한다. 이 함수들은 현재 컴퓨터에 설정된 페이지 파일의 크기를 이용해서 메모리 페이지의 크기와 주소를 결정하는데, 이를 알아보기 위해서는 GetSystemInfo 함수를 사용한다.
이러한 가상 메모리 함수들에는 VirtualAlloc, VirtualFree, VirtualLock, VirtualUnlock 등의 가장 기본적인 함수들과 메모리 페이지의 정보를 얻을 때 사용하는 VirtualQuery, VirtualQueryEx 함수가 있다.
그 밖에 VirtualProtect, VirtualProtectEx 함수를 이용하면 프로세스의 접근 권한 등을 변경할 수 있다.

힙(Heap) 함수

힙 함수는 호출하는 프로세스의 주소 공간에서 하나 이상의 페이지로 된 메모리 블록인 private 힙을 관리할 때 사용된다. 이렇게 할당 받은 private 힙과 일반적인 메모리 할당 함수에 의해 할당 받은 메모리는 근본적으로 같다.
HeapCreate 함수는 HeapAlloc 함수에 의해 할당 받은 private 힙 객체를 생성한다. 이 함수는 힙의 초기 크기와 최대 크기를 지정할 수 있다. 초기 크기는 힙을 처음 할당 받을 때 committed 된 읽기-쓰기가 가능한 페이지를 결정하며, 최대 크기는 예약된 페이지의 총 크기를 결정한다. 이들 페이지 들은 프로세스의 가상 메모리 공간에서 연속된 블록으로 구성되므로 힙이 동적으로 커져갈 수 있다. 만약에 HeapAlloc 함수에 의해 요구된 메모리의 크기가 현재의 committed 페이지의 크기보다 클 경우에는 자동으로 추가적인 페이지가 commint 된다. 일단 이렇게 페이지가 commit 되면, 프로세스가 종료되거나 HeapDestroy 함수에 의해 힙이 파괴되기 전에는 decommit 할 수 없다.
Private 힙 객체의 메모리는 이를 생성한 프로세스에 의해서만 접근이 가능하다. 그 밖에

HeapFree, HeapSize 함수를 사용할 수 있다.
HeapAlloc 함수에 의해 할당된 메모리는 이동할 수 없으며, 분절로 나뉘어 질 수도 있다.
이러한 힙 함수를 사용하는 것은 프로세스가 처음 시작할 때나, 프로세스에 필요한 메모리의 충분한 크기를 지정할 때이며, 만약 HeapCreate 함수 호출이 실패하면 프로세스는 사용자에게 메모리 부족을 알릴 수 있다.

공유 메모리

Win32 API 에서 공유 메모리는 파일 매핑에 의해 구현될 수 있다. 다른 메모리 할당 메소드에 의해서 할당된 메모리는 단지 호출한 프로세스에서만 접근할 수 있다.
파일 매핑을 이용하면 쉽게 공유 메모리 블록을 생성할 수 있다. 프로세스는 CreateFileMapping 함수를 사용할 때 파일 매핑 객체를 생성하기 위해 이름을 지정할 수 있으며, 다른 프로세스에서 이 이름을 이용해서 CreateFileMapping, OpenFileMapping 함수를 호출하면 매핑 객체의 핸들을 얻을 수 있게 된다. 이벤트 객체, 세마포어(semaphore) 객체, 뮤텍(mutex) 객체와 파일 매핑 객체는 같은 이름 공간(name space)을 공유한다. 만약 지정된 이름이 이미 존재하고 있는 다른 종류의 객체와 같으면 에러가 발생한다. 그러므로 이런 객체를 생성할 때에는 이름을 유일하게 지정하도록 해서, 중복을 피해야 한다.
각각의 프로세스는 파일 매핑 객체의 핸들을 MapViewOfFile 함수에서 지정하여 자신의 주소 공간에 매핑한다. 이때 모든 프로세스 들은 하나의 파일 매핑 객체를 공유가 가능한 물리적 저장 공간의 같은 위치를 매핑하게 된다. 그렇지만, 이들의 가상 주소는 프로세스 들마다 다르다.
매핑된 뷰가 물리적 메모리에서 디스크로 스왑되었을 때에는 파일 매핑 객체가 시스템이 사용하는 디스크 파일과 연관되며, 파일 매핑 객체가 생성될 때 지정된 다른 파일이 된다. 이런 경우에 메모리는 파일의 내용에 따라 초기화될 수 있다. 파일 시스템에서 지정된 파일을 매핑하는 것은 이미 존재하는 파일의 데이터를 공유하거나, 공유 프로세스에 의해 생성되는 데이터를 파일에 저장하고자 할 때 매우 유용하다. 만약 지정된 파일을 매핑하면, 이를 exclusive 접근 권한으로 파일을 열어야 하며, 공유 메모리에 대한 작업이 끝날 때까지 이 파일을 계속 열어 두어야 한다. 이렇게 파일을 열어두면 다른 프로세스가 이 파일에 대해 접근할 수 없게 된다.
이렇게 매핑된 파일은 매핑 객체를 이용하는 마지막 프로세스가 종료되거나, UnmapViewOfFile 함수에 의해 매핑이 제거될 때 업데이트 된다. 이러한 업데이트 작업은 FlushViewOfFile 함수에 의해 강요될 수도 있다. 공유 메모리에 대해서는 더 자세하게 다룰 것이다.

델파이의 메모리 관리

델파이에서 사용하는 메모리 관리자(Memory Manager)에 대해 알아보도록 하자. 델파이에서 표준으로 쓰이는 프로시저인 New, Dispose, GetMem, ReallocMem, FreeMem 등이 모두 메모리 관리자를 사용한다. 델파이의 System 유닛에 선언되어 있는 메모리 관리자 레코드의 선언부분을 살펴보자.

```
PMemoryManager = ^TMemoryManager;
TMemoryManager = record
  GetMem: function(Size: Integer): Pointer;
  FreeMem: function(P: Pointer): Integer;
  ReallocMem: function(P: Pointer; Size: Integer): Pointer;
end;
```

사실 이들의 구현 부분을 살펴 보면 위에서 선언된 프로시저 형을 이용해서 실제 구현되는 부분은 인라인 어셈블러로 모두 구현되어 있다. 여기에 대한 자세한 설명은 필자도 정확히 이해하고 있지 못하거니와 이 책에서 다룰 수 있는 범위가 아니라고 생각되므로 생략하도록 한다. 다만 다음 정도를 이해하도록 하자.

델파이가 사용하는 모든 객체와 문자열은 이들 루틴에 의해 내부적으로 관리되고 있다. 이들 루틴의 장점은 small~medium size 의 많은 수의 블록을 관리하는데 최적화 되어 있다는 것이다. 이런 형태의 메모리 관리는 객체를 많이 사용하거나, 문자열을 처리하는 어플리케이션에 적합하다. 사실 이들 외에도 GlobalAlloc, LocalAlloc 등을 이용하여, 윈도에서 private heap 을 사용하도록 할 수도 있지만, 이들을 사용하면 일반적으로 어플리케이션의 속도가 느려진다.

델파이의 메모리 관리자는 Win32 가상 메모리 API 인 VirtualAlloc, VirtualFree 함수를 직접 이용하기 때문에 매우 효율적이다.

각 메모리 블록은 약 4 바이트 정도의 헤더를 가지고 있는데 여기에 각 블록의 크기를 담고 있다. 그리고, 메모리 관리를 위한 상태 변수로 AllocMemCount, AllocMemSize 가 이용된다. 이들은 각각 현재 할당된 메모리 블록의 수와 할당된 메모리 크기를 담고 있다. 이들을 적절히 이용하면 디버깅을 할 때 현재 어플리케이션에서의 메모리 사용의 문제점을 파악하기 쉬워진다.

System 유닛에서는 이 밖에도 GetMemoryManager, SetMemoryManager 라는 프로시저를 제공하는데, 이들을 이용하면 어플리케이션에서 메모리 관리자를 사용하는 호출을 중간에서 가로챌 수 있다. 또한, GetHeapStatus 라는 함수를 이용하면 메모리 관리자의 상태 정보를 담은 레코드를 넘겨받을 수 있다

GetMemoryManager 는 현재의 메모리 관리 루틴을 반환하며, SetMemoryManager 프로시

저를 호출하여 TMemoryManager 레코드에 새로운 함수 포인터를 설정할 수 있다.
메모리 관리자 레코드의 선언부에서 볼 수 있듯이, 핵심이 되는 메모리 관리자 함수는 GetMem, FreeMem 과 ReallocMem 이다. 그런데, 주의할 점은 FreeMem 의 경우 메모리를 해제할 때 델파이에게 해제될 메모리의 크기를 알려주도록 되어 있는데, 메모리 관리자의 선언부를 보면 크기에 대한 파라미터가 없다. 이는 스스로 할당된 메모리의 크기를 결정할 수 있어야 한다는 것이다. GetMem 함수는 블록 반위로 메모리를 할당하고 포인터를 반환하는데, 더 이상 할당할 메모리가 없다면 nil 을 반환하게 되며 nil 이 돌아오면 델파이가 예외를 발생시킨다. ReallocMem 함수는 주어진 크기의 메모리를 블록에 재할당하고 할당된 메모리의 포인터를 반환하는 역할을 한다. 만약 수행할 메모리가 충분하지 않은 경우에는 nil 을 반환하며 델파이는 예외를 발생시킨다.
그렇다면, 실제로 메모리 관리자를 변경하여 사용하는 예를 알아보도록 하자. 가장 전형적인 예는 델파이가 DLL 을 사용할 때 VCL 에 접근하게 될 경우 ShareMem.pas 유닛을 가장 처음에 선언해야 한다는 메시지를 접한 적이 있을 것이다. 이 ShareMem.pas 유닛의 내용이 바로 메모리 관리자를 DLL 에서와 같이 메모리를 공유할 때 적합한 함수들로 변경하는 작업을 하는 것이다. 소스를 잠시 살펴보면 다음과 같다.

```
unit ShareMem;

... (중략)

const
  DelphiMM = 'borlndmm.dll';

function SysGetMem(Size: Integer): Pointer;
  external DelphiMM name '@Borlndmm@SysGetMem$qqri';
function SysFreeMem(P: Pointer): Integer;
  external DelphiMM name '@Borlndmm@SysFreeMem$qqrpv';
function SysReallocMem(P: Pointer; Size: Integer): Pointer;
  external DelphiMM name '@Borlndmm@SysReallocMem$qqrpvi';
function GetHeapStatus: THeapStatus; external DelphiMM;
function GetAllocMemCount: Integer; external DelphiMM;
function GetAllocMemSize: Integer; external DelphiMM;
procedure DumpBlocks; external DelphiMM;

const
  SharedMemoryManager: TMemoryManager = (
```

```
    GetMem: SysGetMem;
    FreeMem: SysFreeMem;
    ReallocMem: SysReallocMem);

initialization
  if not IsMemoryManagerSet then
    SetMemoryManager(SharedMemoryManager);
end.
```

즉 SysGetMem, SysFreeMem, SysReallocMem 을 비롯한 핵심 메모리 관리 함수들을 borlndmm.dll 파일의 함수로 선언하고, SetMemoryManager 함수를 이용하여 메모리 관리자로 설정하는 것이다.
마찬가지로 경우에 따라서는 개발자가 자신의 메모리 관리자를 작성하여 이를 이용하도록 할 수도 있다. 그러면 간단한 사용자 정의 메모리 관리자를 다음과 같이 작성해보자. 여기서는 HeapAlloc, HeapFree, HeapRealloc API 함수를 이용하도록 한다.

```
unit SampleMgr;

interface

implementation
uses Windows;

var
  Heap: THandle;
  MemMgr: TMemoryManager;

function ExamGetMem(Size: Integer): Pointer;
begin
  Result := HeapAlloc(Heap, 0, Size);
end;

function ExamFreeMem(P: Pointer): Integer;
begin
  if HeapFree(Heap, 0, P) then Result := 0 else Result := 1;
end;
```

```
function ExamReallocMem(P: Pointer; Size: Integer): Pointer;
begin
  Result := HeapRealloc(Heap, 0, P, Size);
end;

initialization
begin
  Heap := GetProcessHeap;
  MemMgr.GetMem := ExamGetMem;
  MemMgr.FreeMem := ExamFreeMem;
  MemMgr.ReallocMem := ExamReallocMem;
  SetMemoryManager(MemMgr);
end;

end.
```

그러면, 메모리 관리자를 테스트해 보자. 테스트 폼에는 다음과 같이 TSpinEdit 컴포넌트와 버튼을 하나씩 얹고 SpinEdit1 컴포넌트의 MinValue, MaxValue 프로퍼티를 각각 1, 65536 으로 설정한다. 그리고 Value 프로퍼티를 4 로 설정하여 초기 할당 메모리의 크기를 4KB 로 설정한다.

그리고, Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

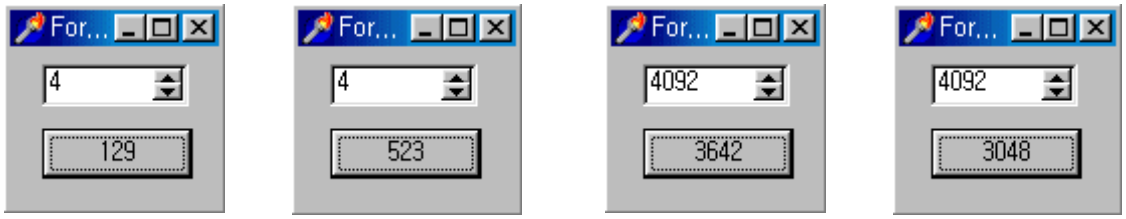
```
procedure TForm1.Button1Click(Sender: TObject);
var
  P: Pointer;
  i, StartTick, StopTick: Integer;
begin
  StartTick := GetTickCount;
  for i := 1 to 1000 do
```

```
  begin
    GetMem(P, SpinEdit1.Value * 1024);
    FreeMem(P);
  end;
  StopTick := GetTickCount;
  Button1.Caption := IntToStr(StopTick - StartTick);
end;
```

그러면, 델파이의 디폴트 메모리 관리자와 방금 작성한 메모리 관리자의 성능을 비교해 보도록 하자. 방금 작성한 프로젝트를 바로 컴파일해서 실행하면 델파이의 디폴트 메모리 관리자를 사용하게 된다. 만약 조금 전에 작성한 SampleMgr.pas 유닛의 메모리 관리 함수를 이용하고자 한다면, 다음과 같이 프로젝트 파일의 uses 절에 제일 앞에 SampleMgr.pas 를 추가해야 한다.

```
program ExamTst2;

uses
  SampleMgr,
  Forms,
  U_ExamTst2 in 'U_ExamTst2.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

다음 결과는 델파이 디폴트 핸들러를 사용한 경우와 SampleMgr 의 메모리 관리 함수를 사용한 경우의 차이를 보여준 경우이다. 여기서는 4KB 의 메모리의 할당, 해제를 반복한 것과 4MB 의 메모리의 할당, 해제를 반복한 경우의 결과를 보여주고 있다.
결과를 보면 작은 메모리를 이용할 때 델파이의 메모리 관리자가 대단히 효율적으로 동작하지만 커다란 메모리에 대해서는 Win32 API 의 HeapAlloc, HeapFree API 함수를 사용하는 것이 더 효율적인 것을 알 수가 있다.



파일 매핑

파일 매핑은 파일의 내용과 프로세스의 가상 주소 공간의 일부를 연관시키는 작업이라고 이해하면 된다. 이때 운영체제는 이를 위한 파일 매핑 객체를 생성한다. 파일 뷰(file view)는 파일의 내용에 접근하는데 사용되는 가상 주소 공간의 일부라고 생각하면 된다. 프로세스는 파일 뷰에 데이터를 읽고 쓸 때 일반적으로 동적으로 할당한 메모리를 사용할 때와 마찬가지로 포인터를 이용하면 된다. 또한, VirtualProtect API 함수를 이용하면 다른 가상 메모리 처럼 파일 뷰를 이용할 수 있다.
윈도우 NT 에서는 이러한 개념이 비슷하지만, 다소 다르게 사용되는데 이를 간단히 정리하면 다음과 같다.
파일 매핑은 파일의 내용을 프로세스의 가상 주소 공간으로 복사하는 것이다. 파일의 내용에 대한 복사본을 파일 뷰라고 하며, 운영체제가 사용하는 복사본에 대한 내부 구조체를 파일 매핑 객체라고 한다.
윈도우는 데이터를 파일에서 읽어와서 파일 뷰에 기록한, 나중에 이를 다시 파일에 기록한다. 사실 상 파일 매핑은 시스템에게 지정된 프로세스의 가상 주소 공간을 지정된 파일이나 시스템 페이지 파일을 이용하게 하는 방법이라고 이해하면 된다.
파일을 매핑하면 프로세스가 디스크 상에 있는 파일에 동적으로 할당받은 메모리에 접근하듯이 접근이 가능하며, 더 나아가서는 하나 이상의 프로세스가 메모리를 공유할 수 있는 수단이 제공된다.

- 파일 매핑의 구현 단계

파일 매핑의 첫번째 단계는 CreateFile 함수를 호출해서 사용할 파일을 여는 것이다. 이 파일은 반드시 다른 프로세스가 접근할 수 없는 상태(exclusive)로 열어야 문제가 생기지 않는다. 가장 쉬운 방법은 CreateFile 함수의 fdwShareMode 파라미터를 0 으로 설정하는 것이다.
CreateFileMapping 함수는 CreateFile 에 의해서 얻은 핸들을 이용해서 파일 매핑 객체를 생성하는 함수이다. 이 함수를 이용해서 파일 매핑 객체의 이름과 파일에서 매핑할 데이터의 크기 (바이트 수), 매핑된 메모리에 접근할 권한 등을 설정하게 된다. 이 함수가 실패하는 경우는 CreateFile 에 의해 열린 파일과 지정된 파일이 일치하되, 접근 권한 플래그에서 충돌이 있게 설정된 경우가 많다. 예를 들어, 파일에 데이터를 읽고 쓰기 위해서는

CreateFile 함수의 fdwAccess 파라미터를 GENERIC_READ 와 GENERIC_WRITE 로 설정하고, CreateFileMapping 함수의 fdwProtect 파라미터를 PAGE_READWRITE 로 설정해야 한다.

파일 매핑은 파일 시스템이 허용하는 한도의 커다란 파일까지 매핑할 수 있다. CreateFileMapping 함수의 dwMaximumSizeHigh, dwMaximumSizeLow 파라미터는 파일에서 매핑될 바이트 수를 지정하는 것으로 파일 매핑의 크기는 실제로 매핑될 파일의 크기와는 독립적이다.

어쨌든 매핑이 파일보다 큰 경우에는 시스템이 CreateFileMapping 함수가 리턴값을 반환하기 전에 파일의 크기를 증가시킨다. 반대로 파일 매핑의 크기가 파일의 크기보다 작으면 파일에서 지정된 크기만큼의 부분만 매핑된다.

어떤 경우에는 파일의 크기를 변경시키고 싶지 않은 경우가 있다 (예를 들어, 읽기 전용 파일을 매핑할 경우). 이럴 때에는 CreateFileMapping 함수의 dwMaximumSizeHigh, dwMaximumSizeLow 파라미터의 값을 모두 0 으로 지정한다. 이렇게 하면 파일 매핑 객체는 파일의 크기와 동일한 크기로 설정된다. 그렇지 않으면, 일단 파일 매핑 객체가 생성되면 매핑 크기가 더 커지거나 작아지지 않으므로, 파일의 크기를 정확하게 계산해서 매핑을 해야 한다.

이제는 파일의 데이터를 실제 메모리로 매핑할 차례이다. 이를 위해서는 앞에서도 잠시 언급한 파일의 뷰를 생성해야 한다. 이때 MapViewOfFile 과 MapViewOfFileEx API 함수를 사용한다. CreateFileMapping 함수를 호출하면 파일 매핑 객체의 핸들을 얻을 수 있는데, 파일의 뷰를 생성하거나 프로세스의 가상 주소 공간 내의 파일의 일부분에 대한 뷰를 생성할 때 이 핸들을 이용한다. MapViewOfFileEx 함수는 기본적으로 MapViewOfFile 함수와 같은 역할을 하는데, 프로세스가 파일의 뷰에 대한 base 주소를 지정할 수 있다는 점이 다르다.

MapViewOfFile 함수는 파일 뷰에 대한 포인터를 반환한다. 어플리케이션은 단순히 이 포인터를 이용해서 동적 메모리에 접근하듯이 데이터를 읽고 쓰면 된다. 실제로 읽고 쓰는 작업은 시스템에 의해 디스크 상의 파일에 반영된다. 그런데, 데이터가 파일 매핑 객체에 기록되면, 이것이 바로 파일로 전달되지는 않는다. 이런 데이터는 보통 캐쉬 메모리에 저장되었다가 마지막에 파일에 기록된다. 어플리케이션에서 만약에 그때 그때 데이터를 디스크에 기록하고 싶은 경우에는 FlushViewOfFile 함수를 이용한다.

어플리케이션은 같은 파일 매핑 객체에서 여러 개의 파일 뷰를 생성할 수 있다. 이들 각각의 파일 뷰는 크기가 다르며, 이들의 옵셋은 MapViewOfFile 함수의 dwOffsetHigh, dwOffsetLow 파라미터에 의해 지정된다. 이때 시스템의 메모리 할당에 대한 정보를 알고 싶은 경우에는 GetSystemInfo 함수를 호출하여 SYSTEM_INFO 구조체에 정보를 얻어오면 된다.

파일 매핑 객체의 사용이 끝나면, 프로세스는 먼저 모든 파일 뷰를 파괴해야 하는데, 이때 사용하는 함수가 UnmapViewOfFile 함수이다. 참고로 FlushViewOfFile 함수를 이용해서

데이터를 디스크에 기록할 때에도 먼저 UnmapViewOfFile 함수를 호출해야 한다.
파일 매핑 객체와 파일은 모두 CloseHandle 함수를 호출해서 닫을 수 있다. 먼저 파일 매핑 객체를 닫고, 나중에 파일을 닫는다. 파일 매핑 객체를 닫은 뒤에 필요하면 프로세스가 파일의 크기를 지정할 수 있는데, 이때에는 SetFilePointer 와 SetEndOfFile 함수를 이용한다. 예를 들어, 매핑이 파일의 크기보다 큰 경우에는 프로세스가 파일 포인터를 요구되는 파일의 크기로 설정할 수 있다.

- 파일과 메모리의 공유

파일 매핑은 두 개 이상의 프로세스에 의해 파일이나 메모리가 공유될 때 유용하게 사용될 수 있는 기법이다. 이렇게 하려면, 관련된 모든 프로세스가 같은 파일 매핑 객체와 파일 뷰를 사용하면 된다. 각각의 프로세스의 뷰는 프로세스의 가상 주소 공간에 위치한다. 만약에 하나의 프로세스가 뷰를 업데이트 하면, 다른 프로세스 들도 이러한 업데이트를 즉시 알 수 있다. 파일을 공유하려면 첫번째 프로세스가 CreateFile 함수를 이용해서 파일을 생성하거나 연다. 그리고, CreateFileMapping 함수에서 파일 매핑 객체에 대한 이름을 지정하여 파일 매핑 객체를 생성한다. 파일과 연관되지 않는 메모리를 공유하기 위해서는 프로세스는 반드시 CreateFileMapping 함수를 이용하되 이미 존재하는 파일의 핸들을 지정하는 대신, 파라미터로 $FFFFFFFF 를 지정한다. 이렇게 하면 파일 매핑 객체는 시스템 페이지 파일을 이용해서 메모리에 접근하게 된다. 그리고, 이 경우에는 반드시 매핑의 크기를 0 보다 크게 설정해야 한다.
다른 프로세스들이 생성된 파일 매핑 객체의 핸들을 얻는 가장 좋은 방법은 OpenFileMapping 함수에 파일 매핑 객체의 이름을 지정해서 호출하는 것이다. 만약 파일 매핑 객체가 이름이 없을 경우에는 프로세스는 상속(inheritance)나 복제(duplication)을 이용해서 핸들을 얻어와야 한다.
파일이나 메모리를 공유하는 프로세스 들은 반드시 MapViewOfFile 을 이용해서 파일 뷰를 생성해야 한다. 그리고, 파일 뷰들은 서로의 데이터를 보호하기 위해서 세마포어(semaphore), 뮤텍(mutex), 이벤트 등의 동기화 테크닉을 사용해야 한다.
이렇게 공유된 파일 매핑 객체는 모든 프로세스 들이 CloseHandle, UnMapViewOfFile 을 호출해서 핸들을 닫을 때까지 파괴되지 않는다.

- 일치성 (Coherence)

일치성은 하나의 파일 뷰에서 접근이 가능한 데이터는 디스크에 있는 파일의 내용과 동일한 복사본이라는 것을 보장하는 것이다. CreateFileMapping 함수는 원격지 파일 (remote file)에도 접근할 수 있는데, 이 경우에는 이러한 일치성이 보장되지 않는다. 예를 들어, 두 개의 컴퓨터가 하나의 파일을 쓰기 가능으로 설정하고 파일 매핑 객체를 생성했다고 하자.

이 때 두 컴퓨터에서 같은 페이지의 내용을 변경한 경우 각각의 컴퓨터는 자신이 페이지에 수행한 직업만을 반영해서 볼 수 있다. 데이터가 실제로 디스크에 업데이트될 때에는 이들이 서로 합쳐지지 않는다.

- 파일 매핑의 이용

처음에 파일 매핑 객체를 생성할 때에는 다음과 같이 한다.

```
hMapFile = CreateFileMapping(hFile,     //현재 파일의 핸들,
    NULL,                               //디폴트 보안값
    PAGE_READWRITE,                     //읽고, 쓰기가 모두 가능하게 설정
    0,                                  //최대 객체의 크기
    0,                                  //hFile 의 크기
    'MyFileMappingObject');             //매핑 객체의 이름
if (hMapFile = NULL) then ErrorHandler('파일 매핑 객체를 생성할 수 없다네 ...');
```

다음의 코드는 파일 매핑 객체의 핸들을 이용해서 파일의 뷰를 생성하는 부분이다.

```
lpMapAddress = MapViewOfFile(hMapFile,          //매핑 객체의 핸들
    FILE_MAP_ALL_ACCESS,                        //읽고, 쓰기가 모두 가능하게
    0,                                          //최대 객체의 크기
    0,                                          //hFile 의 크기
    0);                                         //전체 파일을 매핑한다.
if (lpMapAddress = NULL) then ErrorHandler('파일의 뷰를 매핑 못한다네 ...');
```

MapViewOfFile 함수는 파일 뷰에 대한 포인터를 반환한다. 프로세스는 메모리에 접근할 때 이 포인터를 사용한다. 다음의 코드에 의해 두번째 프로세스는 OpenFileMapping 함수를 이용해서 같은 메모리를 공유하게 된다

```
hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,     //읽고-쓰기 권한
    FALSE,                                          //상속받지 않음
    'MyFileMappingObject');                         //매핑 객체의 이름
if (hMapFile = NULL) then ErrorHandler('파일 매핑 객체를 열수 없다네 ...');

lpMapAddress = MapViewOfFile(hMapFile,          //매핑 객체의 핸들
    FILE_MAP_ALL_ACCESS,                        //읽고-쓰기 접근 권한
```

```
    0,                                          //객체의 최대 크기
    0,                                          //hFile 의 크기
    0);                                         //전체 파일을 매핑
if (lpMapAddress = NULL) then ErrorHandler('파일의 뷰를 매핑할 수 없다네 ...');
```

다음의 코드는 UnmapViewOfFile 함수를 이용해서 프로세스 주소 공간에서 파일 뷰를 파괴하는 부분이다. 만약에 파일 뷰가 매핑된 후에 바뀐 내용이 있으면, UnmapViewOfFile 함수에 의해 변화된 부분을 디스크 파일에 복사한다.

```
if not UnmapViewOfFile(lpMapAddress) then ErrorHandler('파일 뷰를 해제 못함 ... ');
```

프로세스가 파일 매핑을 모두 사용하고, 파일 뷰의 매핑을 모두 해제 했으면, 파일 매핑 객체를 다음과 같이 닫는다.

```
CloseHande(hMapFile);
```

FlushViewOfFile 함수는 파일 뷰에서 지정된 바이트 수만큼 물리적 파일에 기록한다.

```
if not FlushViewOfFile(lpMapAddress, dwBytesToFlush) then
    ErrorHandler('메모리를 디스크에 기록할 수 없다네 ... ');
```

메모리 맵 파일(memory mapped file)을 이용한 데이터 공유 기법

앞에서 설명한 파일 매핑을 이해한다면, 다음의 루틴 들도 쉽게 이해할 수 있을 것으로 믿는다. 다음에 소개하는 OpenMap 과 CloseMap 은 실제 어플리케이션을 사용할 때 그대로 가져가서 사용할 수도 있는 부분이므로 유용하게 사용하기 바란다.
참고로, 파일 매핑 객체에 대한 핸들과 파일 뷰에 대한 포인터에 대한 전역 변수를 선언해서 써야 한다. 그러므로, 먼저 다음과 같이 변수 선언을 전역으로 한다.

```
var
  HMapping: THandle;
  PMapData: Pointer;

procedure OpenMap(MapFileSize: Integer, MappingObject: String);
begin
  HMapping := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, MAPFILESIZE,
```

```
    PChar(MappingObject));
  if (hMapping = 0) then
  begin
    ShowMessage('메모리 맵을 생성할 수 없습니다.');
    Application.Terminate;
    Exit;
  end;
  PMapData := MapViewOfFile(HMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
  if PMapData = nil then
  begin
    CloseHandle(HMapping);
    ShowMessage('파일 뷰를 매핑할 수 없습니다.');
    Application.Terminate;
    Exit;
  end;
end;

procedure CloseMap;
begin
  if PMapData <> nil then UnMapViewOfFile(PMapData);
  if HMapping <> 0 then CloseHandle(HMapping);
end;
```

하나 이상의 어플리케이션이나 DLL 에서 같은 물리적 메모리 블록의 포인터를 얻기 위해서는 이런 방법을 써야 한다. 여기서 PMapData 변수는 MapFileSize 파라미터에 넘겨진 바이트 크기의 데이터에 대한 버퍼를 가리키는 포인터이다. 한가지 생길 수 있는 문제점은 메모리에 접근하는 데이터를 동기화해야 한다는 것이다. 이를 위해서 뮤텍(mutex)을 사용할 수 있는데, 뮤텍을 이용해서 동기화를 할 수 있도록 다음의 LockMap, UnLockMap 프로시저를 적절하게 활용하면 된다. 사용 방법은 메모리 맵 파일에 데이터를 읽거나 쓰기 전에는 LockMap 을 호출한다. 그리고, 자료의 업데이트가 끝나면 바로 UnLockMap 을 호출하면 된다.
그리고, 각 프로젝트에 뮤텍의 핸들을 저장한 전역 변수를 다음과 같이 선언해 주어야 한다.

```
var
  HMapMutex: THandle;
```

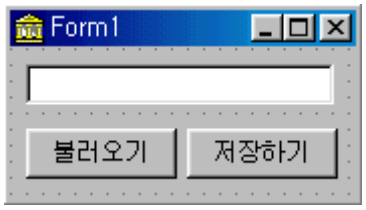
```
const
  REQUEST_TIMEOUT = 1000;

function LockMap(MutexName: String): Boolean;
begin
  Result := True;
  HMapMutex := CreateMutex(nil, False, PChar(MutexName));
  if HMapMutex = 0 then
  begin
    ShowMessage('뮤텍을 생성할 수 없습니다.');
    Result := False;
  end
  else
  begin
    if WaitForSingleObject(HMapMutex, REQUEST_TIMEOUT) = WAIT_FAILED then
    begin        // timeout
      ShowMessage('메모리 맵 파일을 잠글 수 없습니다.');
      Result := False;
    end;
  end;
end;

procedure UnlockMap;
begin
  ReleaseMutex(HMapMutex);
  CloseHandle(HMapMutex);
end;
```

여기에 대한 더 자세한 설명은 쓰레드와 동기화에 대해서 설명한 장을 참고하기 바란다.

메모리 맵 파일의 활용

메모리 맵 파일은 대용량의 데이터를 순식간에 처리하는 빠른 처리 속도가 가장 큰 장점이다. 그렇기 때문에 데이터의 임시 저장 장소로 사용하거나 대용량의 데이터(그래픽 등)를 실시간에 주고 받아야 하는 경우에 활용도가 높다. 그러면, 간단한 예제를 통해서 문자열을 임시 저장할 공간으로 메모리 맵 파일을 활용하는 방법을 익혀보도록 하자. 먼저 폼에

버튼 2개와 에디트 박스를 하나 얹어서 다음과 같이 디자인하도록 하자.



그리고, 파일 매핑 핸들로 사용할 변수와 매핑할 데이터로 사용할 변수, 에러 코드 변수를 public 섹션에 선언하고 문자열의 최대 길이를 상수로 다음과 같이 선언한다.

```
  public
    hFileMap: THandle;
    MapError: Integer;
    SharedPChar: PChar
  end;

const
  CharLen = 100;
```

이제 OnCreate 이벤트 핸들러에서 파일 매핑 객체를 생성하고, 뷰를 매핑하도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  hFileMap := 0;
  SharedPChar := nil;
  try
    hFileMap
      := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, CharLen + 1, 'SampleMMF');
    if hFileMap = 0 then
      raise Exception.Create('메모리 맵 파일 생성에 실패했습니다 !')
    else
      MapError := GetLastError();
    SharedPChar
      := PChar(MapViewOfFile(hFileMap, FILE_MAP_READ + FILE_MAP_WRITE, 0, 0, 0));
    if MapError <> ERROR_ALREADY_EXISTS then
      StrPLCopy(SharedPChar, '샘플입니다 !', CharLen);
```

```
    Edit1.Text := SharedPChar;
  except
    on E: Exception do begin
      if SharedPChar <> nil then
        UnmapViewOfFile(SharedPChar);
      if hFileMap <> 0 then
        CloseHandle(hFileMap);
    end;
  end;
```

이미 앞에서 CreateFileMapping 과 MapViewOfFile API 함수의 사용법에 대해서는 자세히 설명하였으므로, 이를 참고하기 바란다. 이 코드가 성공적으로 수행되면 SharedPChar 는 공유 데이터를 담는 변수로 사용된다. 그리고, 초기 값으로 '샘플입니다 !' 라는 문자열을 설정한다.
반대로 OnDestroy 이벤트 핸들러에서는 생성된 파일 매핑을 해제해야 한다.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  UnmapViewOfFile(SharedPChar);
  CloseHandle(hFileMap);
end;
```

일단 이렇게 설정되면, 사용하는 방법은 매우 간단하다. SharedPChar 변수가 공유 데이터라고 간주하고, 이를 직접 Edit1.Text 프로퍼티를 설정하거나 읽어오도록 다음과 같이 코딩하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text := SharedPChar;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  StrPLCopy(SharedPChar, Edit1.Text, CharLen);
end;
```

이것으로 간단한 예제가 완성되었다. 컴파일하고 여러 개의 인스턴스를 실행한 뒤에 하나의 인스턴스의 에디트 박스의 내용을 변경하고 '저장하기' 버튼을 클릭한다. 그리고, '불러오기' 버튼을 클릭하면 저장된 내용으로 에디트 박스의 내용이 변경될 것이다.

정 리 (Summary)

이번 장에서는 델파이의 메모리를 관리하는 방법과 메모리를 공유하기 위해 메모리 맵 파일을 사용하는 방법에 대해서 알아보았다. 이런 내용은 실제 어플리케이션을 작성할 때 직접적인 영향을 미치지는 않지만, 전반적인 수행성능을 향상시키거나 기능을 확장하는데 도움이 되는 것들이므로 잘 익혀놓으면 잘 만들어진 어플리케이션을 만드는데 커다란 도움이 될 것이다.