20장. 메모리 관리(Virtual Memory, Heap, MMF)

01. 가상 메모리 컨트롤

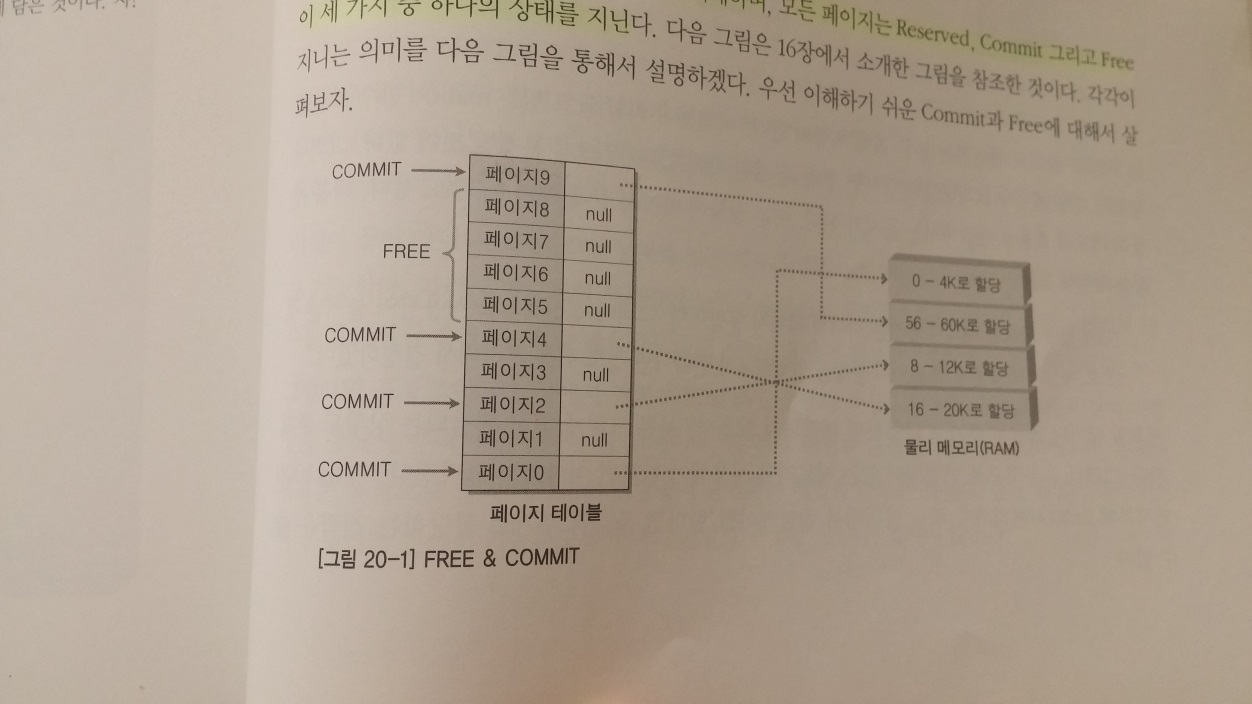
Windows에서는 가상 메모리를 컨트롤할 수 있는 기능을 제공하는데, 이는 메모리의 효율적인 사용을 위해서 도움이 된다. 함수와 예제를 제시하는 형태로 Windows에서 제공하는 가상 메모리 콘트롤에 대해 알아보자.

**| Reserve, Commit 그리고 Free**

Reserve는 예약, Commit은 할당, Free는 할당 되지 않았음을 의미. 이것은 Windows 시스템에서 부여할 수 있도록 정의한 페이지의 상태를 의미하는 것으로 페이지의 총 개수는 다음과 같은 공식으로 계산할 수 있다.

**가상 메모리의 크기 / 페이지 하나당 크기 = 페이지 개수**

즉, 페이지의 개수는 가상 메모리의 크기에 비례하며, 모든 페이지는 Reserved, Commit 그리고 Free이 세가지 중 하나의 상태를 지닌다.

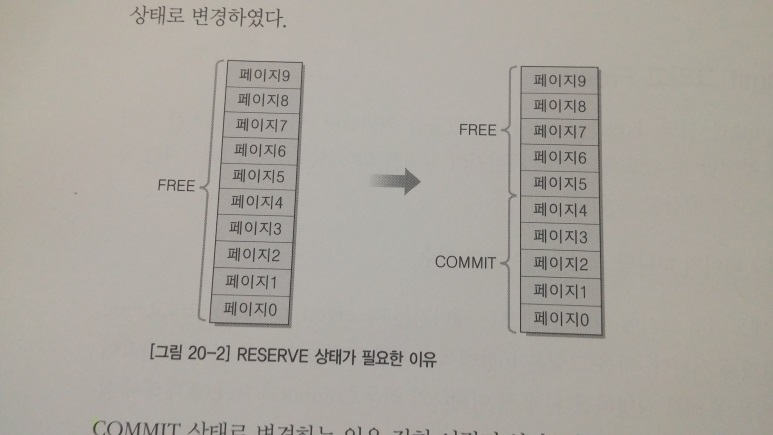
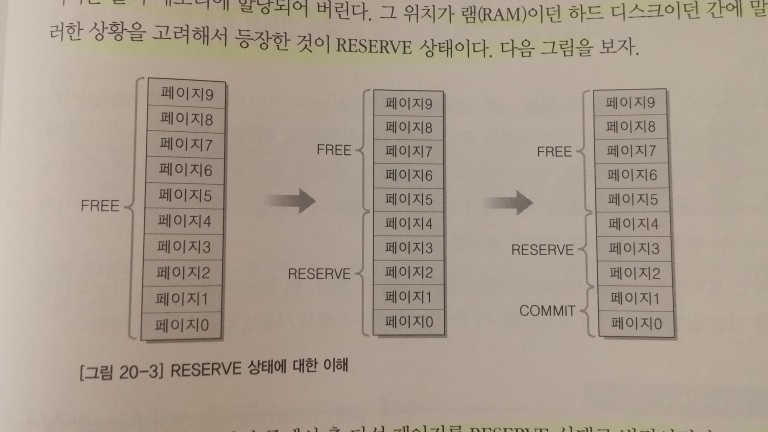


위 그림엔 페이지 별 상태가 표시되어 있다.

1. COMMIT은 물리 메모리에 할당이 이루어진 부분들이다. 물리 메모리란 램, 하드디스크를 모두 포함한 개념이며, 즉 해당 페이지가 물리 메모리에 할당된(malloc) 상태를 가르켜 COMMIT상태라 한다.

2. FREE는 물리 메모리에 전혀 할당이 이뤄지지 않은 영역을 뜻한다.

언뜻 위의 2개의 상황만으로 전체 메모리를 설명할 수 있을 것 처럼 보인다. 하지만 Windows에서는 RESERVE라는 상태를 하나 더 두어 메모리 사용의 효율성을 높일 기회를 제공한다.

RESERVE 상태가 필요한 예, “내게 당장 필요한 메모리 공간은 하나의 페이지 분량이지만, 점진적으로 사용량이 늘어 날 것이고 예상되기로 100 페이지 가량 필요할 것이다.”

3. RESERVE는 위 같은 상황을 위해 만들어진 개념이다. RESERVE는 COMMIT과 FREE의 중간 상태이다. 일부 페이지를 RESERVE 상태로 둠으로써 다른 메모리 할당함수에 의해 해당 번지가 할당되지 못하도록 선언할 수 있다. 그리고 예약을 했을 뿐 아직 메모리에 할당이 되지 않은 상태이다. 또, RESERVE 상태에 있는 메모리 중에서 일부만 COMMIT 상태로 변경하는 것도 가능하다. 따라서 메모리의 사용량이 늘어남에 따라서 점진적으로 COMMIT 상태의 페이지 개수를 증가 시킬 수 있다.

정말 필요한 만큼의 페이지만 물리 메모리에 할당하는 것이 가능하다.

**| 메모리 할당의 시작점과 단위 확인하기**

페이지 단위로 메모리를 할당하기에 앞서 고려 해야 할 내용이 있다. 메모리를 할당하기 전에 기본적으로 생각해야 하는 것은 다음 두 가지 이다.

**메모리 할당의 시작 주소, 할당할 메모리의 크기**

가상 메모리 시스템은 페이지 단위로 관리된다. 때문에 페이지의 중간 위치에서부터 할당을 시작할 수 없으며, **페이지 크기의 배수 단위로 할당을 해야만 한다.**

Windows에서는 메모리가 지나치게 조각나는 것을 막기 위해 조금 더 넓은 범위의 값을 할당의 경계로 정의한다. 이것을 가리켜 ‘Allocation Granularity Boundary’라 한다.

‘Allocation Granularity Boundary’의 크기를 알아야 합리적인 메모리 할당이 가능하다. 원하는 정보는 GetSystemInfo 함수를 통해서 얻을 수 있다.

[예제 20-1 SYSTTEM\_INFO.cpp]

실행 결과를 통해 알 수 있듯이 ‘Allocation Granularity Boundary’도 페이지 크기의 배수이다.

**| VirtualAlloc & VirtualFree 함수**

가상 메모리를 관리 하기 위해서는 두 가지 시스템 함수를 알아야 한다.

첫 번째 함수는 VirtualAlloc 이다. 페이지 상태를 RESERVE와 COMMIT 상태로 만드는 역할을 한다.

|  |
| --- |
| LPVOID VirtualAlloc (  LPVOID lpAddress,  SIZE\_T dwSize,  DWORD flAllocationType,  DWORD flProtect  ); |

- **lpAddress** : 예약 및 할당하고자 하는 메모리의 시작 주소를 지정, 일반적으로 NULL을 전달하게 되는데, NULL이 전달되면 할당하고자 하는 크기에 맞춰서 메모리의 위치를 임의로 지정, 그러나 RESERVE 상태에 있는 페이지를 COMMIT 상태로 변경할 때에는 해당 페이지의 시작주소를 지정해야 한다. 예약을 할 때에는 ‘Allocation Granularity Boundary’를 기준으로 값이 조절되고, 할당을 할 때에는 페이지 크기 단위로 값이 조절 된다.

- **dwSize** : 할당하고자 하는 메모리의 크기를 바이트 단위로 지정한다. 메모리의 할당은 페이지 크기 단위로 결정된다. Ex) 페이지 크기가 4byte인 경우 14byter할당을 요청하면 16byte가 할당됨

- **flAllocationType** : 메모리 할당의 타입을 결정한다. RESERVE 상태로 구조가 한다면 MEM\_RESERVE를 인자로 전달, COMMIT의 경우 MEM\_COMMIT 전달

- **flProtect** : 페이지 별 접근 방식에 제한을 두는 용도로 사용. Windows 시스템 함수를 사용했을 때 얻게 되는 장점이다. 기본적으로 RESERVE 상태에 둘 때에는 접근을 허용하지 않는 PAGE\_NOACCESS를, COMMIT 상태로 변경할 때에는 읽기 쓰기를 모두 허용하는 PAGE\_READWRITE를 인자로 전달한다. (MSDN 참조)

- **반환값 :** 함수 호출이 성공하면 할당이 이뤄진 메모리의 시작 번지를 반환한다.

두 번째 함수는 VirtualFree이다. 할당된 페이지를 되돌리는 역할을 수행한다.

|  |
| --- |
| BOOL VirtualFree (  LPVOID lpAddress,  SIZE\_T dwSize,  DWORD dwFreeType  ); |

- lpAddress : 해제할 메모리 공간의 시작 주소를 지정한다.

- dwSize : 해제할 메모리 크기를 바이트 단위로 지정한다.

- dwFreeType : MEM\_DECOMMIT과 MEM\_RELEASE 중 하나를 지정할 수 있다.

MEM\_RELEASE를 지정할 경우, 해당 페이지는 FREE 상태가 된다. 물리적 메모리가 할당되어 있다면 해당 메모리는 반환된다. 주의할 사항은 MEM\_RELEASE 전달 시 dwSize는 반드시 0이어야 하고, lpAddress의 값은 VirtualAlloc함수 호출을 통해 예약된 메모리의 시작 번지가 되어야 한다는 것이다. 그리고 예약이 이뤄진 메모리의 일부만 반환하는 것은 불가능하다.

MEM\_DECOMMIT을 인자로 전달할 경우에는 해당 페이지의 상태를 RESERVE 상태로 되돌리게 된다. 이 경우에도 물리적 메모리가 할당되어 있으면 해당 메모리는 반환하게 된다.

**| Dynamic Array Design**

Dynamic Array라는 정식 명칭은 아니지만 보편적으로 쓰이는 용어이다. 일반적인 배열처럼 한 순간에 배열 크기 만큼 물리 메모리가 할당되는 것이 아니라 사용양의 증가에 따라 물리 메모리에 할당되는 배열의 크기가 점진적으로 증가하는 배열이다.

시나리오

**1단계**, 시스템 페이지 사이즈와 ‘Allocation Granularity Boundary’ 값을 얻어 온다. 할당하고자 하는 메모리의 위치에 직접적으로 관여하지 않겠다면, 페이지 사이즈만 얻어와도 된다.

**2단계**, 메모리를 예약(RESERVE)한다. 예약을 할 때에는 필요하다고 예상되는 최대의 크기를 예약한다.

**3단계,** 필요한 만큼의 메모리를 물리 메모리에 할당(COMMIT)한다. 필요에 따라 점진적으로 할당의 크기를 증가 시킨다.

**4단계**, 할당했던 메모리를 반환한다.

위의 시나리오를 기반으로 Dynamic Array를 구현하는 방법으로 성능적인 부분이나 구현의 편리성을 종합해서 보았을 때 가장 최고로 꼽히는 방법은 구조적 예외처리 (SEH)를 활용하는 것이다.

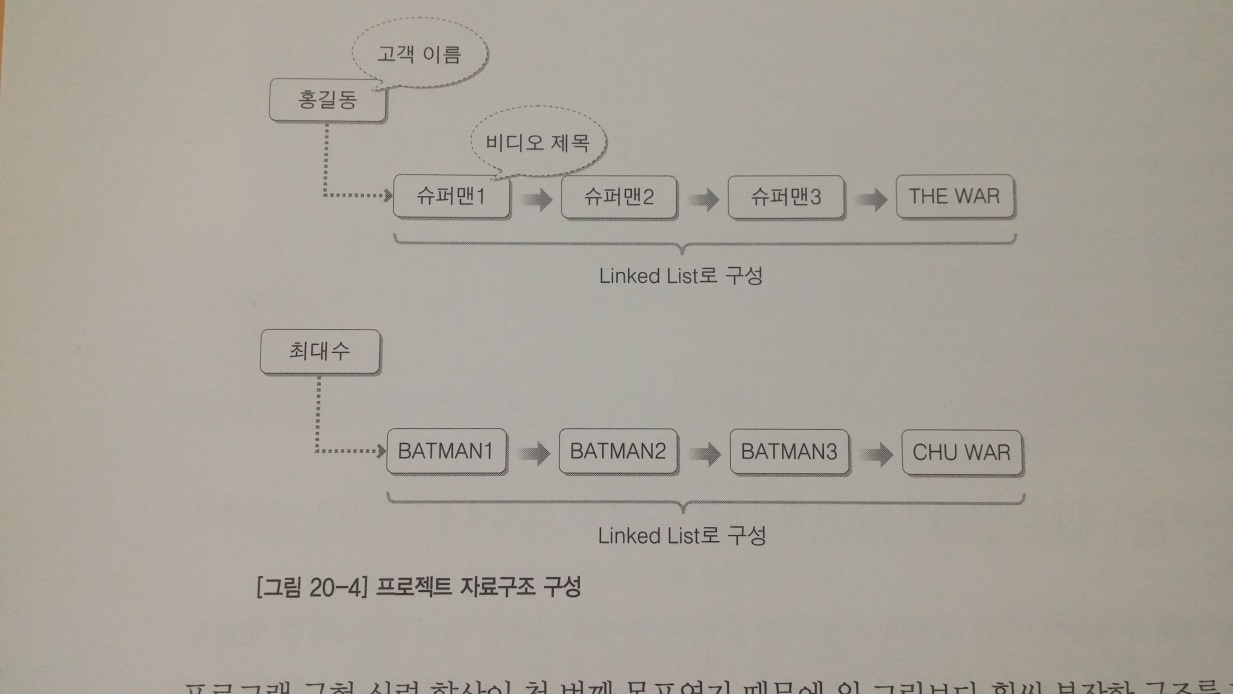
[예제 20-2 DynamicArray.cpp]

배열 선언 이후로 배열 사용량이 증가함에 따라서 한 페이지씩 물리 메모리로의 할당이 추가로 이뤄지게 된다. 한 페이지씩 할당할 것인지, 둘 이상의 페이지씩 할당할 것인지는 결정의 문제이나 한 페이지씩 할당할 경우 너무 잦은 SHE 처리와 빈번한 함수 호출로 인해서 성능에 문제가 발생할 수 있다.

02. 힘(Heap) 컨트롤

힙 을 컨트롤 하는 것은 가상 메모리를 컨트롤하는 것보다 최소 몇 배 이상 유용한 것이다. 가상 메모리 컨트롤이 개념적인 측면에서 중요하게 여겨 진다면, 힙 컨트롤은 프로그램의 성능 뿐만 아니라, 구현의 용이성을 위해서도 필요하다.

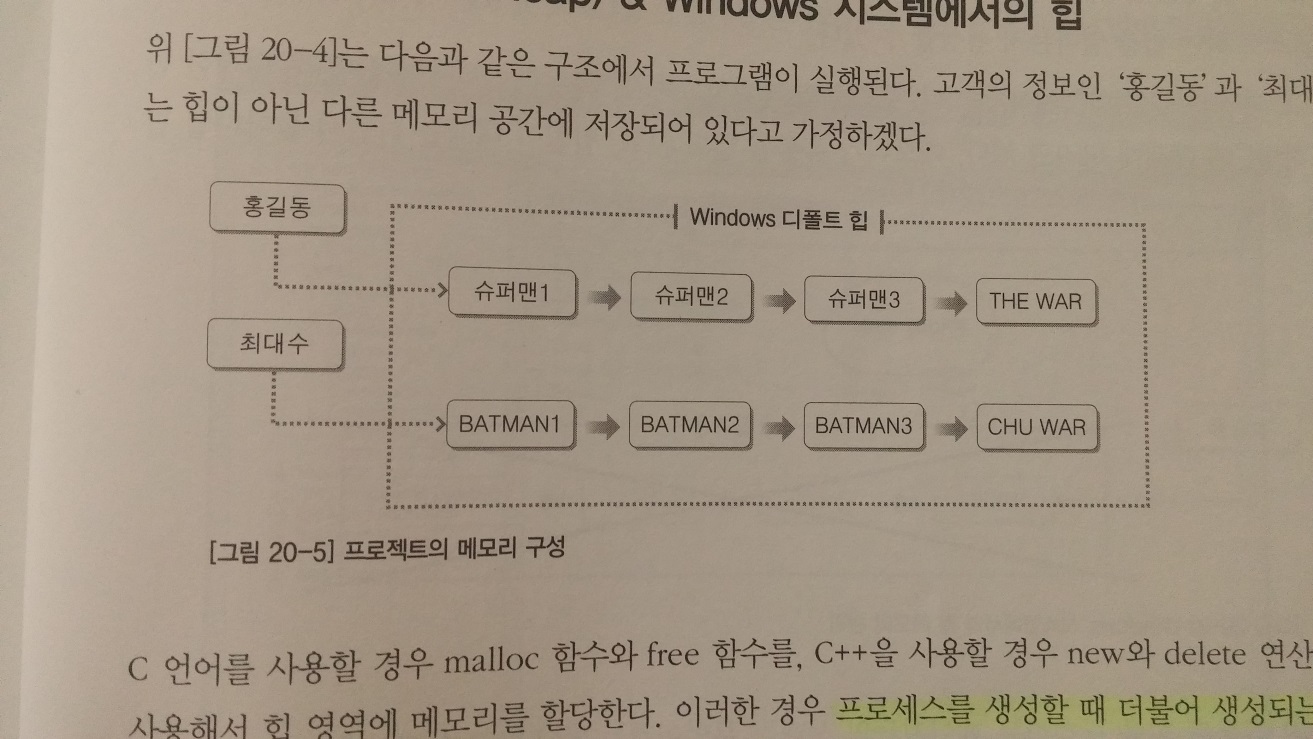
**| 힙(Heap) 컨트롤에 대한 필자의 기억**

****

위 그림에서는 리스트 자료 구조가 곳곳에서 등장하였는데, 문제는 리스트 자료구조를 삭제하는 과정에서 발생하였다. 만약에 회원 홍길동이 탈퇴를 요구하면 홍길동에 대한 정보는 모두 삭제하게 된다. 리스트 자료 구조의 특성상 각각의 리스트 정보는 동적 할당 되어 힙에 저장된다. 이때, 리스트의 처음부터 끝까지 추적해 가면 일일이 반환 해야 한다.

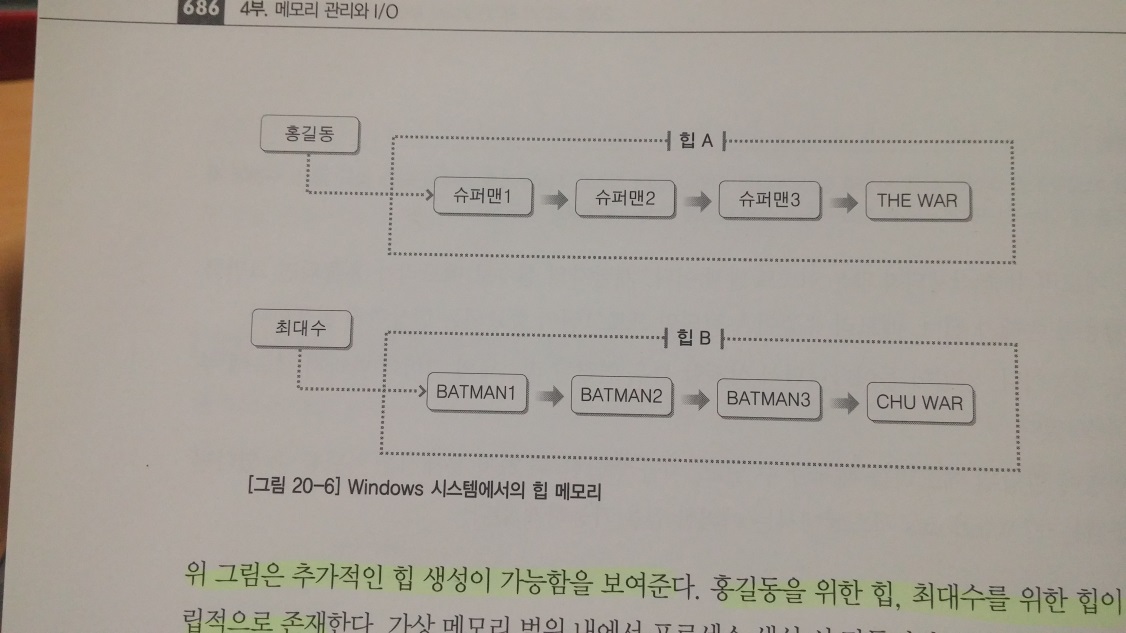
여기서 문제점은, 1. 메모리 유출, 개발자의 실수로 메모리가 유출 될 수 있다. 2. 성능, 일일이 루프를 도는 것은 성능에 부담이 될 수 있다.

**| 디폴트 힙(Default Heap) & Windows 시스템에서의 힙**

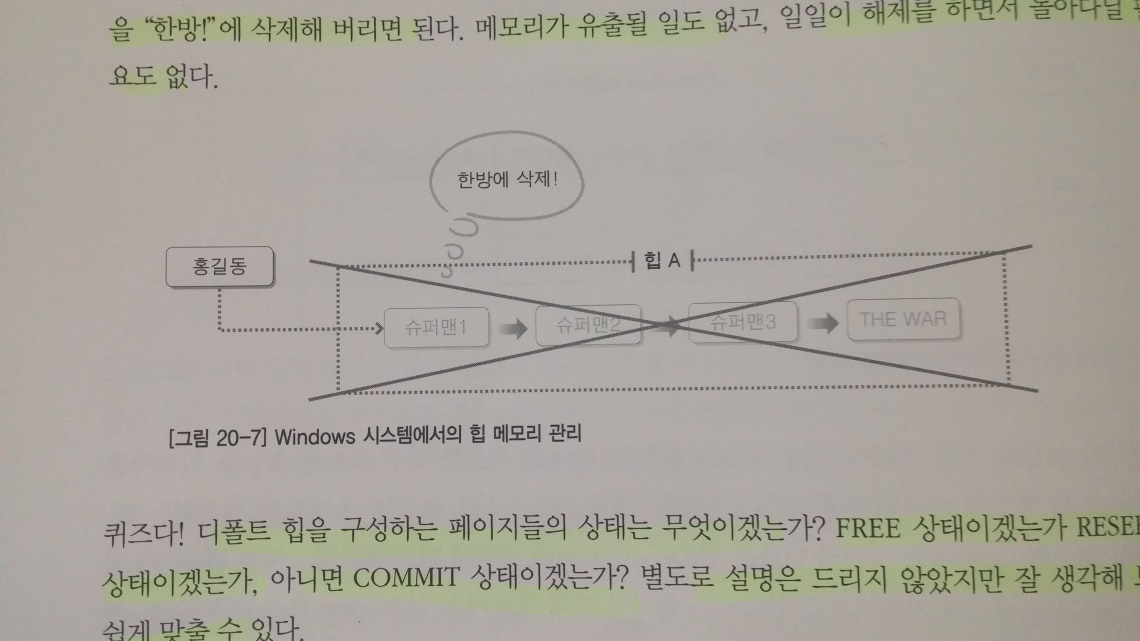
****

Malloc, free, new, delete 연산자를 사용해서 힙 영역에 메모리를 할당하게 되는데, 이런 경우 프로세스를 생성할 때 더불어 생성되는 힙, 1M 바이트 크기의 디폴트 힙(Default Heap) 영역에 메모리를 할당하게 된다. 이 디폴트 힙은 프로세스 힙(Process Heap)이라고도 부른다.

Windows 시스템 함수를 활용하면 다음과 같은 구조로 메모리 구성을 변경할 수 있다.



위 그림은 추가적인 힙 생성이 가능함을 보여준다. 힙 A, 힙 B로 각각 독립적으로 힙이 존재하고 있으며, 필요로 하는 힙을 얼마 든지 만들 수 있다는 것을 알 수 있다. 만약 홍길동이 탈퇴 한다면, 홍길동을 위해 할당한 힙을 삭제해 버리면 된다. 메모리가 유출될 일도 없고, 일일이 해제를 하면서 루프를 돌 필요도 없다.



추가로 디폴트 힙을 구성하는 페이지들의 상태는 RESERVE 상태이다.

**| 디폴트 힙 컨트롤**

디폴트 힙의 크기는 1M바이트 이지만, 링커(Linker)옵션을 통해서 변경이 가능하다. 다음과 같은 형식으로 링커 옵션을 통해, RESERVE 상태에 놓이게 되는 힙의 전체 크기와 이중에서 초기에 COMMIT 상태로 둘 메모리 크기를 지정할 수 있다.

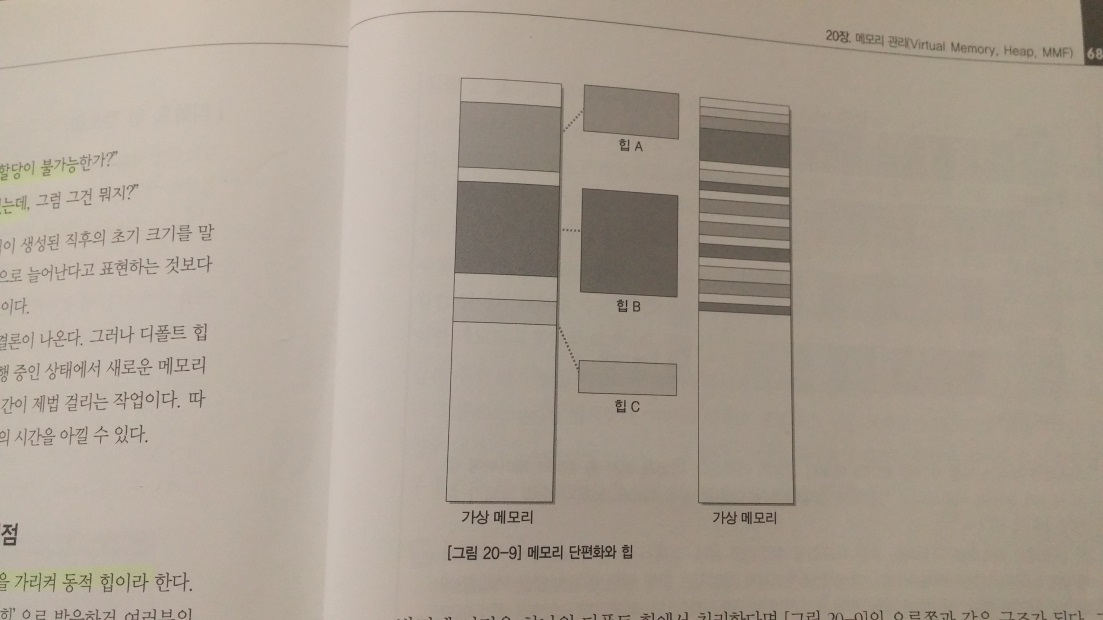
또, 디폴트 힙의 기본 크기 1M바이트는 힙이 생성된 직후의 초기 크기를 말하는 것이다. 필요에 따라서 그 크기는 자동으로 늘어 난다.

**| 힙(Dynamic Heap) 생성이 가져다 주는 또 다른 이점**

디폴트 힙 이외의 Windows 시스템 함수 호출을 통해서 생성되는 힙을 가리켜 동적 힙이라 한다.

힙을 추가로 생성하는 것에 대한 장점으로 한가지 예를 보여 줬었는데, 이 밖에도 다른 장점들은 존재한다.

**장점1. 메모리 단편화의 최소화에 따른 성능 향상**



A라는 기능을 위해서 힙A를 생성하고, B라는 기능을 위해서 힙B를 생성하고, C라는 기능을 위해서 힙C를 생성한다면, 왼쪽과 같은 구조로 메모리가 할당된다. 반면에 이것을 하나의 디폴트 힙에서 처리한다면 오른쪽 그림과 같은 구조가 될 것이다. 오른편은 심각한 메모리 단편화(Fragmentation)가 발생할 소지가 높다.

일단, A, B, C의 힙을 미리 선언하면 할당된 페이지가 RESERVE 상태에 놓이기 때문에 메모리 단편화가 발생하지 않는다. 반면에 디폴트 힙을 활용할 경우 프로그램 실행과정에서 무작위 메모리 할당 및 그에 따른 힙 크기의 증가에 의해 메모리 단편화가 심하게 발생한다.

**장점2. 동기화 문제에서 자유로워짐으로 인한 성능 향상**

스레드 별로 사용할 힙을 별도로 할당해 주는 것은 의미 있는 일이다. 일반적으로 힙은 스레드가 공유하는 메모리 영역이다. 때문에 둘 이상의 스레드가 동시 접근할 때 문제가 발생할 수 있다. 같은 주소 번지에 둘 이상의 스레드가 동시에 메모리를 할당 및 해제하는 상황이 발생할 경우메모리 오류(Corrupt)가 발생한다. 때문에 디폴트 프로세스 힙은 스레드가 메모리를 할당하려고 하는 경우 내부적으로 동기화 처리를 하고 있다. 그런데 하나의 스레드당 하나의 힙을 할당할 경우 동기화 처리가 필요가 없기 때문에 성능향상을 기대할 수 있다.

**| 힙의 생성과 소멸 그리고 할당**

힙을 생성하는 함수

|  |
| --- |
| HANDLE HeapCreate (  DWORD flOptions,  SIZE\_T dwInitialSize,  SIZE\_T dwMaximumSize  ); |

- **flOptions** : 생성되는 힙의 특성을 부여하는 데 사용된다. 0을 전달할 경우 가장 일반적인 힙이 생성된다.

HEAP\_GENERATE\_EXCEPTIONS을 전달할 경우 오류 발생 시 NULL을 반환하는 것이 아니라 예외를 발생시킨다.

HEAP\_NO\_SERIALIZE을 지정할 경우 생성된 힙에 메모리를 할당 및 해제할 때 동기화 처리를 하지 않게 된다**(장점 2.)**

스레드 별 힙이 독립된 상황에서 HEAP\_NO\_SERIALIZE을 인자로 지정 하는게 성능에 좋다.

- **dwMaximumSize :** 생성되는 힙의 크기를 결정한다. 지정하는 크기에 해당하는 페이지 수만큼 RESERVE 상태가 된다. 0이 값으로 전달되면, 힙은 증가 가능한 메모리(Growable Heap)가 된다. 따라서 0이 전달될 경우 메모리가 허락하는 한도 내에서 힙의 크기는 증가한다.

- **dwInitialSize** : dwMaximumSize에서 지정한 메모리 중에서 초기에 할당할 물리 메모리 크기를 지정한다. 여기서 지정한 크기에 해당하는 페이지 수만큼힙이 생성되자 마자 COMMIT 상태가 된다.

- **반환값 :** 생성된 힙을 컨트롤 하는 데 사용되는 힙의 핸들이 반환된다.

힙을 소멸하는 데 사용하는 함수, RESERVE 상태에 놓여 있던 페이지를 (COMMIT 샅애에 있던 페이지 포함해서)FREE 상태로 되돌리는 함수로 이해해도 된다.

|  |
| --- |
| BOOL HeapDestroy (  HANDLE hHeap  ); |

- **hHeap** : 반환하고자 하는 힙의 핸들을 인자로 전달한다.

힙을 생성했다면 힙에 메모리를 할당하고 해제할 차례이다. 힙에 메모리를 할당할 때에는 다음 함수를 사용한다. 요청 크기에 해당하는 페이지 수만큼 COMMIT 상태로 변경시킨다는 점을 이해하자.

|  |
| --- |
| LPVOID HeapAlloc (  HANDLE hHeap,  DWORD dwFlags,  SIZE\_T dwBytes  ); |

- **hHeap :** 메모리 할당이 이뤄질 힙의 핸들을 지정한다.

- **dwFlags :** HEAP\_GENERATE\_EXCEPTIONS가 인자로 올 경우, 오류 발생 시 NULL을 반환하지 않고 예외를 발생시킨다. HEAP\_NOSERIALIZE를 인자로 전달할 경우 함수 호출은 동기화 처리 되지 않는다. HeapCreate 함수 호출 과정에서 이미 HEAP\_NOSERIALIZE을 전달하였다면 이 전달 인자를 통해 중복 지정할 필요가 없다. 그리고 HEAP\_ZERO\_MEMORY가 전달되면 할당된 메모리는 0으로 초기화 된다. 둘 이상의 속성을 비트 단위 OR 연산자 (|)로 동시 지정이 가능하다.

- **dwBytes :** 할당하고자 하는 메모리의 크기를 지정한다. 참고로 힙이 증가 가능한 힙이 아니라면 (Non-Growable Heap), 다시 말해서 힙 생성 시 HeapCreate 함수 호출의 마지막 인자가 0이 아니면,. 최대 할당 크기는 0x7FFF8로 제한된다.

힙에 할당된 메모리를 해제하는 데 사용하는 함수이다. 메모리가 해제되는 과정에서 페이지는 다시 RESERVE 상태가 될 수 있다.

|  |
| --- |
| BOOL HeapFree (  HANDLE hHeap,  DWORD dwFlags,  LPVOID lpMem  ); |

- **hHeap :** 해제할 메모리를 담고 있는 힙을 지정한다.

- **dwFlags :** HEAP\_NO\_SERIALIZE가 인자로 올 수 있다. HeapCreate 함수 호출 과정에서 이미 HEAP\_NO\_SERIALIZE을 전달하였다면 이 전달인자를 통해서 중복 지정할 필요가 없다.

- **lpMem** : 해제할 메모리의 시작 주소를 지정한다.

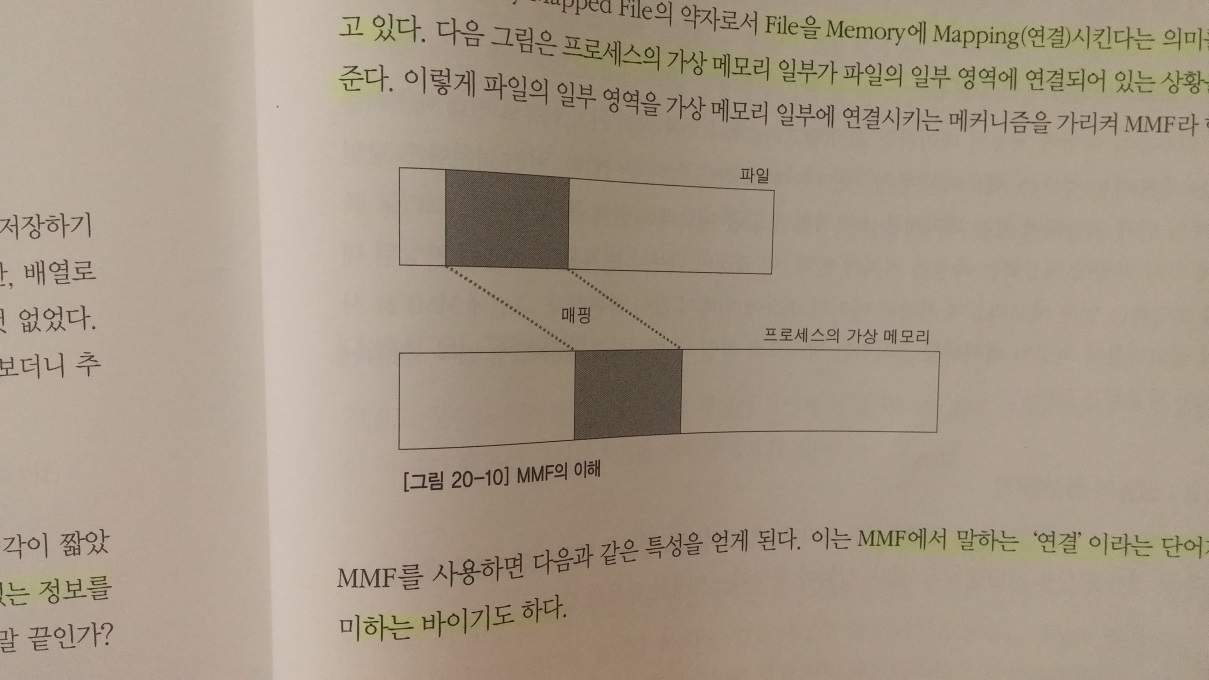
**| Heap & Linked List 예제**

[예제 20-3 DynamicHeap.cpp]

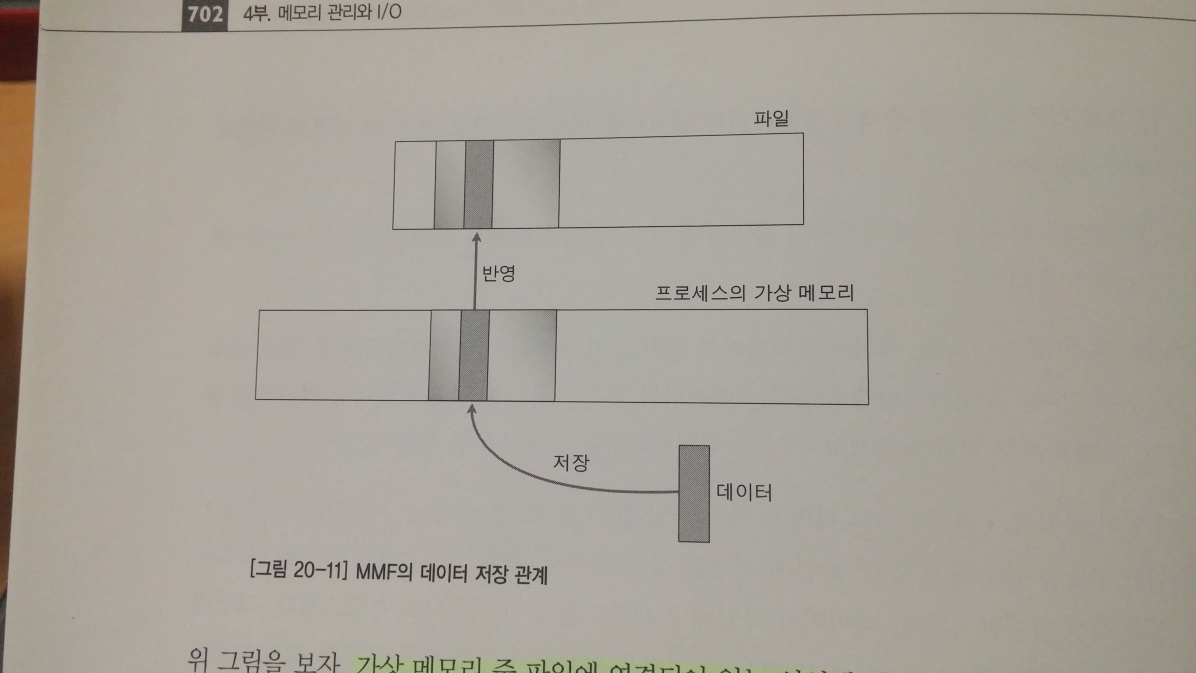
**03 MMF (Memory Mapped File)**

**| MMF의 이해**

MMF는 Memory Mapped File의 약자로서 File을 Memory에 Mapping 시킨다는 의미를 지니고 있다. 다음 그림에서 프로세스의 가상 메모리 일부가 파일의 일부 영역에 연결 되어 있는 상황을 보여 준다.



MMF를 사용하면 얻게 되는 특징

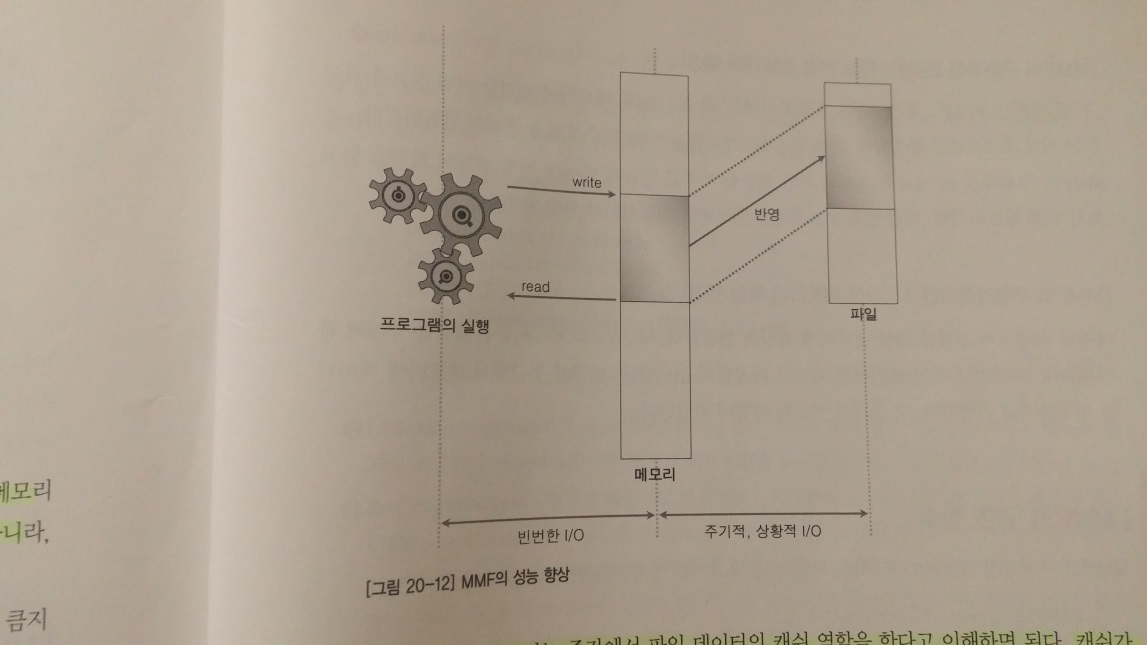
   
가상 메모리 중 파일에 연결되어 있는 영역에 데이터를 저장하며, 이렇게 저장된 데이터는 실제 파일에도 효과가 미친다. 즉 , 메모리에 연결된 파일에 실제 데이터가 저장되는 것. 이 매커니즘은 두 가지 장점으로 정리 할 수 있다.

장점 1: 프로그래밍하기 편하다.

파일 안에 저장된 데이터를 조작하려면 메모리로 읽어 들여야 하며, 조작 후에 다시 파일에 저장하는 과정을 거쳐야 하는데, 이는 너무 불편하다. 하지만 MMF를 사용하면 메모리상에 데이터를 조작하는 방식으로 파일 내 데이터를 조작할 수 있다.

장점 2: 성능이 향상된다.

성능이 저하될 것 같지만, 일반적으로 성능이 향상된다. 이유는 다음과 같다.



메모리는 중간에서 파일 데이터를 캐쉬 역할을 한다고 이해하면 된다. 캐쉬가 존재하게 되므로 직접 파일에 접근하는 것보다 효율적인 접근이 이뤄진다. 메모리에 저장되어 있는 데이터는 주기적으로(혹은 특정 상황) 파일에 저장된다.

하지만 메모리에 저장된 데이터가 변경될 때마다 파일에 바로 적용시키는 것도 가능 하지만 성능향상은 기대 하기 힘들다.

**| MMF의 구현과정**

MMF의 구현과정 1단계 : 파일 개방

MMF는 파일의 일부를 메모리에 연결 시키는 것이므로 당연히 파일이 존재해야 한다. 즉, 파일의 핸들을 필요로 한다. CreateFile 함수 호출을 통해서 파일을 열고 파일의 핸들을 얻어야 한다.

MMF의 구현과정 2단계 : 파일 연결 오브젝트 생성

파일 [연결/매핑] 오브젝트라 불리는데, 메모리에 연결할 파일 정보를 담고 있는 커널 오브젝트를 생성하는 것이다. CreateFileMapping 함수 호출을 통해서 만들어 진다.

MMF의 구현과정 3단계 : 가상 메모리에 파일 연결

실제 가상 메모리에 파일을 연결한다. MapViewOfFile 함수 호출을 통해서 완성되는데, 이때 반환되는 포인터를 가지고 메모리에 접근한다.

**| MMF의 구현 함수**

|  |
| --- |
| HANDLE CreatFileMapping (  HANDLE hFile,  LPSECURITY\_ATTRIBUTES lpAttributes,  DWORD flProtect,  DWORD dwMaximumSizeHigh,  DWORD dwMaximumSizeLow,  LPCTSTR lpName  ); |

- hFile : 메모리에 연결한 파일의 핸들을 지정한다.

- lpAttributes : 커널 오브젝트 생성하는 다른 함수와 마찬가지로 보안 속성을 지정

- flProtect : 파일에 연결된 메모리의 접근 권한을 지정한다. PAGE\_READONLY를 지정하면 읽기만 가능, PAGE\_READWRITE를 지정하면 읽기, 쓰기 모두 가능, 그 외에 ‘Copy-on-wirte’를 위한 PAGE\_WRITECOPY 및 비트 단위 연산 ‘|’ 으로 묶어서 함께 설정 가능한 SEC\_COMMIT, SEC\_RESERVE 등이 있다.

- dwMaximumSizeHigh : 연결할 메모리 최대 크기의 상위 4바이트를 지정한다. 대용량이 아니라면 0으로 충분

- dwMaximumSizeLow : 연결할 메모리 최대 크기의 하위 4바이트를 지정한다. 0이 전달되면, 첫 번째 인자로 전달된 핸들의 크기로 지정된다.

- lpName : 파일 연결 오브젝트의 이름을 지정한다. NULL을 전달할 수 있다.

위 함수의 호출이 성공할 경우 **파일 연결 오브젝트의 핸들을 반환**한다. 이 핸들을 이용햐여 다음 함수를 호출하면 실제 메모리로의 연결이 완성된다. MMF의 구현과정 3단계이다.

|  |
| --- |
| LPVOID MapVuewOfFile (  HANDLE hFileMappingObject,  DWORD dwDesiredAccess,  DWORD dwFileOffsetHigh,  DWORD dwFileOffsetLow,  SIZE\_T dwNumberOfBytesToMap  ); |

- hFileMappingObject: : CreateFileMapping 함수 호출을 통해 얻은 커널 오브젝트의 핸들을 인자로 전달

- dwDesiredAccess : 연결된 메모리의 접근권한을 지닌다. CreateFileMapping 함수의 세번쨰 인자자 PAGE\_READWRITE였다면, 읽기 쓰기 둘 중하나 혹은 둘 다 지정할 수 있다. 반면에 PAGE\_READ라면 오로지 읽기권한만 지정할 수 있다. 읽기 권한은 FILE\_MAP\_READ, 쓰기는 FILE\_MAP\_WRITE, 읽기 쓰기는 FILE\_MAP\_READ | FILE\_MAP\_WRITE를 전달 한다.

- dwFileOffsetHigh : 지정한 파일 전부를 메모리에 연결해야만 하는 것은 아니다. 파일의 일부 영역만 선택해서 연결할 때에 파일의 오프셋 Offset을 지정한다. 오프셋의 상위 4바이트

- dwFileOffsetLow : 메모리에 연결할 파일의 오프셋의 하위 4바이트를 지정

- dwNumberOfBytesToMap : 메모리에 연결할 실제 크기를 바이트 단위로 지정한다. 오프셋을 기준으로 지정된 바이트 만큼 메모리에 전달된다. 0을 전달하게 되면 오프셋으로부터 파일의 끝까지 메모리에 연결된다.

위 함수호출 결과로 void형 포인터가 반환되며, 이 포인터를 용도에 맞게 형 변환하여 사용하면 된다. Malloc 함수 호출 시 얻게 되는 포인터를 사용하는 방법과 같다. 이렇게 파일과 연결된 메모리는 작업이 끝나고 나면, 연결 해제 과정을 거쳐야 한다. 다음 함수를 활용해서 연결을 해제한다.

|  |
| --- |
| BOOL UnmapViewOfFile (  LPCVOID lpBaseAddress  ); |

- lpBaseAddress : 연결 해제할 메모리의 시작 주소를 지정한다. MapViewOfFile 함수 호출 시 반환되었던 주소값을 전달하면 된다.

**| MMF의 구성 예제 1 : READONLY**

[예제 20-4 MemoryMappedFileRead.cpp]

**| 읽고 쓰기 위한 MMF : READWRITE**

[예제 20-5 MemoryMappedFileSort.cpp]

[예제 20-5]가 [예제 20-4]와 다른 가장 큰 차이점은 파일이 비어있는 상태에서 MMF를 구성했다는 점이다. [예제 20-4]에서는 파일을 생성하고 문자열을 하나 저장하였다. 그리고 그 문자열을 읽기위해서 MMF를 구성했다. 그러나 이번 예제는 MMF를 구성한 다음에 데이터를 저장하고자 한다. 파일 연결 오브젝트를 생성할 때 파일과 메모리를 연결할 수 있는 최대 크기를 지정하도록 되어 있다. CreateFileMapping 함수의 4, 5 번째 전달인자를 통해서 지정한다. 각각 상위 4바이트 정보와 하위 4바이트 정보를 전달하도록 되어 있는데, 모드 0이 전달된다면 연결하고자 하는 파일의 크기로 값이 결정 된다.

MMF 기반으로 데이터를 저장할 경우, 메모리에 캐쉬 된 다음 주기적으로 혹은 특정 조건하에서 파일로 저장된다고 하였다. 그런데 때로는 메모리에 캐쉬된 데이터를 파일에 저장할 것을 명령하고 싶다,. 마치 파일 I/O의 fflush 함수 처럼 말이다. 그렇다면 다음 함수를 활용하면 된다.

|  |
| --- |
| BOOL FlushViewOfFile (  LPCVOID lpBaseAddress,  SIZE\_T dwNumberOfBytesToFlush  ); |

- lpBaseAddress : 파일에 저장할 메모리의 시작 주소를 지정한다.

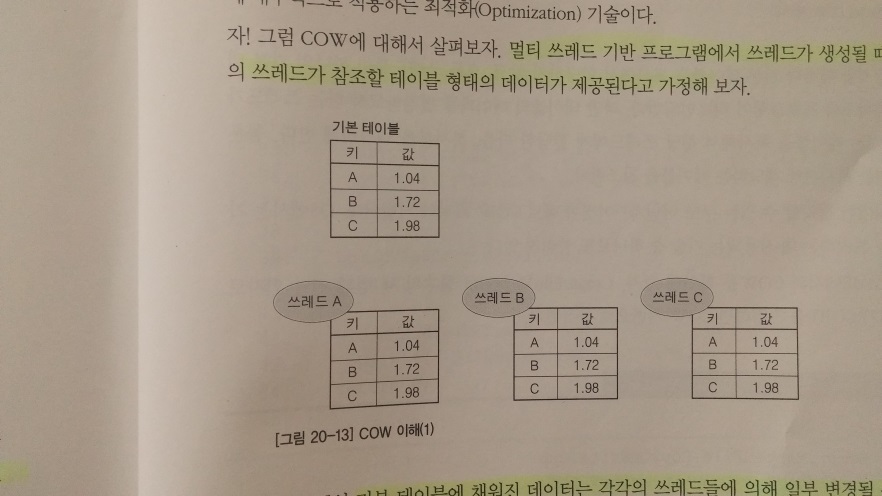
- dwNumberOfBytesToFlush : 파일에 저장할 데이터 크기를 바이트 단위로 지정한다. 만약에 두 번째 전달인자로 0을 전달할 경우 연결된 메모리 영역의 끝까지 파일로 저장된다.

**| Copy-On-Write(COW)**

Copy-On-Write는 의미상 ‘write 할 때 copy 하라’ 정도이다. 즉, 데이터를 쓸 때 복사하라는 의미이다,. COW는 일반 개발자들이 주로 사용하는 기술이 아니며, MMF 같은 시스템 수준에서 제공 되거나, OS같은 고급 소프트웨어를 구현할 때에 내부적으로 적용하는 최적화 기술이다.

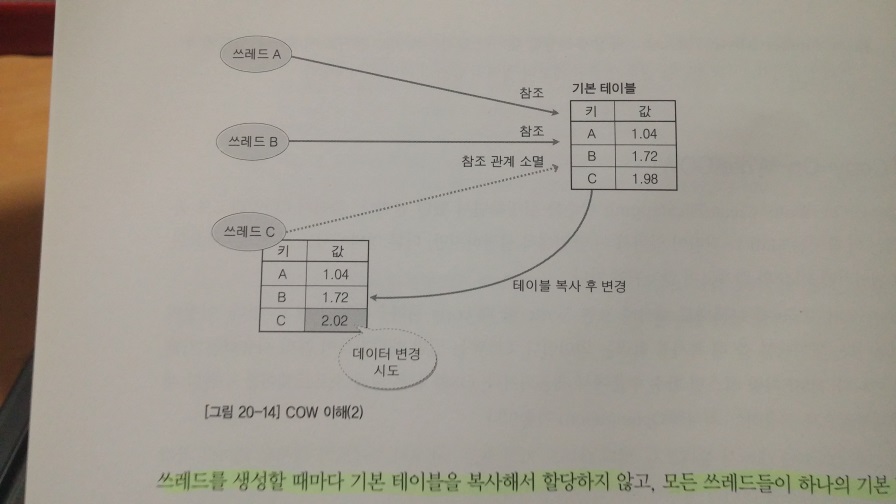
**COW에 대한 이해**

멀티 스레드 기반 프로그램에서 스레드가 생성될 때마다 각각의 스레드가 참조할 테이블 형태의 데이터가 제공된다고 가정해 보자.



기본 테이블에 채워진 데이터는 각각의 스레드들에 의해 일부 변경될 수 있다. 때문에 모든 스레드는 자신만의 테이블을 별도로 지녀야 한다. 즉, 스레드 생성 시 기본 테이블의 복사가 발생하는 것이다. 여기서 한가지 특징이 있다. 대부분의 스레드가 기본 테이블 정보를 참조만 하고 있고, 변경은 드물게 일어나고 있다. 즉 드물게 일어나는 데이터 변경 때문에 모든 스레드들이 독립된 테이블 정보를 지니도록 해야만 하는가? 너무 큰 메모리 손실이다.

이러한 문제가 있을 때 COW기법을 도입하면 보다 효율적인 구조가 된다.



스레드를 생성할 때마다 기본 테이블을 복사해서 할당하지 않고, 모든 스레드들이 하나의 기본 테이블을 공유하도록 프로그램이 디자인 되었다. 다만 테이블의 데이터를 변경하고자 하는 스레드가 등장하면, 기본 테이블을 복사해서 해당 스레드에게 할당한 다음, 복사본을 변경하게 한다. 물론 그 이후 부터는 복사본에 해당하는 테이블을 참조한다.

메모리를 최대한 절약할 수 있는 구조이다. OS에서는 가상 메모리를 관리하는 데 사용되는 기술 중 하나로도 알려져 있다.

[예제 20-6 MemoryMappedFile\_CopyOnWrite.cpp]