3장 : 64비트 기반 프로그래밍

Section 1 : WIN32 vs WIN64

64비트와 32비트

CPU는 데이터를 내부로 가져오고 또 외부로 보내기 위해 입출력 버스를 사용한다. 이 때, 한 번에 전송 및 수신할 수 있는 데이터의 크기에 따라서 32비트 시스템과 64비트 시스템이 나뉘게 된다.

또 하나의 기준은 데이터 처리 능력이다. CPU에 들어오는 데이터를 한 번에 32비트 만큼 처리할 수 있느냐, 64비트만큼 처리할 수 있느냐에 따라서 32비트 시스템과 64비트 시스템을 나눌 수 있다.

프로그래머 입장에서의 64비트 컴퓨터

프로그래머 입장에서 64비트 시스템의 가장 중요한 이점 중 하나는 주소 값의 범위가 확장된다는 것이다.

32비트 시스템에서는 주소 값 하나에 32비트를 할당한다. 32비트 자료형 하나로 표현할 수 있는 주소값의 범위는 0부터 2의 32승인 4GB까지 주소를 표현할 수 있다. 즉, RAM이 8GB가 있든 16GB가 있든 32비트 시스템에서는 4GB까지 밖에 사용하지 못하는 것이다. (메모리의 주소를 지정할 수가 없으므로)

그러나 64비트 시스템에서는 주소 값을 표현할 때 하나에 64비트를 할당하게 되므로, 메모리를 사용할 수 있는 범위가 어마어마하게 넓어지게 된다.

그렇다면 32비트에서 주소 값을 할당할 때 64비트로 넣으면 되지 않을까? 그러나 이는 속도의 저하를 야기시킨다. CPU가 연산할 때 주소 값을 불러오는 경우가 굉장히 많을 텐데, 주소 값에 32비트를 할당한다면 한 번의 데이터 이동 만으로 주소 값을 불러올 수 있다. 그러나 64비트를 할당하게 되면 두 번의 데이터 이동이 끝난 후에야 연산을 처리할 수 있다.

그러므로 32비트 시스템에서 주소 값에 32비트 할당을 한 것은 속도와 메모리 크기와의 적절한 타협점을 찾을 것이라고 볼 수 있다.

Section 2 : 프로그램 구현 관점에서의 WIN32 vs WIN64

LLP64 vs LP64

지금까지는 int와 long, 그리고 포인터 자료형이 모두 4바이트로 표현된다고 배웠지만, 이는 모두 32비트 환경에서 Windows가 자료형을 표현하는 방식이었다. 64비트 환경에서는 이에 차이가 생긴다.

운영체제	모델	char	short	int	long	포인터
Windows	LLP64	1바이트	2바이트	4바이트	4바이트	8바이트
UNIX	LP64	1바이트	2바이트	4바이트	8바이트	8바이트

여기서 Windows는 LLP64라는 모델을, UNIT는 LP64라는 모델을 따른다. Windows의 LLP64를 보면 int와 long이 4바이트로 표현되어 기존 32비트 시스템과의 호환성을 중시하고 있음을 알 수 있다. UNIX의 LP64모델은 long을 8바이트로 표현하는 것을 볼 수 있다.

64비트와 32비트 공존의 문제점

32비트에서는 int, long, 포인터가 모두 4바이트 자료형이기 때문에 서로 간의 캐스팅 같은 문제에서 비교적 자유로웠다. 그러나 64비트 환경에서 포인터는 8바이트로 표현된다. 그 때문에 32비트 시스템에서처럼 포인터를 4바이트 정수형으로 변경했다가는 데이터 손실이 일어날 수 있다.

Windows 스타일 자료형

Windows에서의 실행만 고려한다면 MS에서 정의하고 있는 자료형을 사용하는 것도 좋은 선택이다.

WINDOWS 자료형	의미	정의 형태
BOOL	Boolean variable	typedef int BOOL
DWORD	32-bit unsigned integer	typedef unsigned long DWORD;
DWORD32	32-bit unsigned integer	typedef unsigned int DWORD32
DWORD64	64-bit unsigned integer	typedef unsigned __int64 DWORD64
INT	32-bit signed integer	typedef int INT;
INT32	32-bit signed integer	typedef signed int INT32
INT64	64-bit signed integer	typedef signed __int64 INT64
LONG	32-bit signed integer	typedef long LONG
LONG32	32-bit signed integer	typedef signed int LONG32
LONG64	64-bit signed integer	typedef signed __int64 LONG64
UINT	Unsigned INT	typedef unsigned int UINT
UINT32	Unsigned INT32	typedef unsigned int UINT32
UINT64	Unsigned INT64	typedef unsigned __int64 UINT64
ULONG	Unsigned LONG	typedef unsigned int ULONG
ULONG32	Unsigned LONG32	typedef unsigned int ULONG32
ULONG64	Unsigned LONG64	typedef unsigned __int64 ULONG64

WIN32시스템에서는 DWORD, INT, LONG과 같은 자료형이 많이 사용되었다. WIN64 시스템으로 넘어가면서 여기에 ~32, ~64형태의 자료형이 추가된 것을 볼 수 있는데, 이는 시스템에 상관없이 동일한 의미를 지니는 자료형을 표현하기 위한 것이다. (시스템에 상관없이 동일하게 32비트를 사용하는 INT형태라던지)

WIN32 시절에 정의된 자료형이 WIN64에서 쓸모가 없어진 것은 아니다. 아직도 WIN32에서 정의된 자료형을 많이 사용하고 있기 때문이다.

Polymorphic 자료형

Polymorphic이라는 것은 다형성을 의미한다. 다형성을 갖는 자료형이라는 것은 무엇일까? 상황에 따라서 자료형의 정의가 유동적이라는 의미이다. _UNICODE의 정의 여부에따라 의미가 바뀌는 TCHAR같은 자료형을 의미한다.

```
#if defined(_WIN64)

typedef __int64 LONG_PTR;
typedef unsigned __int64 ULONG_PTR;

typedef __int64 INT_PTR;
typedef unsigned __int64 UINT_PTR;

#else

typedef long LONG_PTR;
typedef unsigned long ULONG_PTR;

typedef int INT_PTR;
typedef unsigned int UINT_PTR;

#endif
```

자료형은 PTR로 되어있지만 정작 내용은 int, long의 정수형 자료형이다. 이는 포인터값 기반의 산술 연산을 위해 정의된 자료형이기 때문에 PTR이라는 이름을 붙인 것이다. 그런데 왜 이렇게 번거롭게 자료형을 정의해 놓은 것일까?

위에서 알 수 있듯이 포인터는 32비트 시스템과 64비트 시스템에서 크기가 다르다. 그러므로 하나의 고정된 자료형을 이용해서는 둘 중 한 환경에서는 포인터 계산이 잘못될 수 있기 때문이다.

Section 3 : 오류의 확인

GetLastError 함수와 에러코드.

Windows 시스템 함수 호출 중 오류가 발생하면 GetLastError 함수 호출을 통해 오류의 원인을 확인할 수 있다.

대부분의 시스템 함수는 오류가 발생했을 때 NULL을 반환하는데, 이는 오류가 발생했음은 알 수 있지만, 원인을 알 수는 없다. 이때 이어서 바로 GetLastError 함수를 호출하면 원인에 해당하는 에러코드를 얻을 수 있다. MSDN을 참조하면 에러코드의 종류과 의미하는 바를 알 수 있다.

중요한 것은 Windows 시스템 함수가 호출될 때마다 GetLastError 함수가 반환하는 에러코드가 갱신되기 때문에, 오류가 발생한 직후 오류 확인을 해주어야 한다는 것이다.