1 장 윈도 95 프로그래밍 입문

윈도 3.1

마이크로소프트의 출세작 MS-DOS 는 그 화려한 명성과 성공에도 불구하고 사실 기술적으로나 운영체제로서의 품질로서나 형편없는 졸작이었다고 할 수 있을 것이다. 유닉스 명령들을 흉내낸 명령어들과 사용법으로 인해서 PC 입문자들을 끊임없이 괴롭혀 왔다. 반면, 애플 매킨토시 운영체제는 그 화려하고도 편리한 사용자 인터페이스를 가지고 많은 PC 사용자들을 유혹해 오고 있었다. 사실, 필자도 대학 때 처음 매킨토시를 접했을 때, 왜 이런 PC 가 성공적으로 보급되지 못하고 있는 것일까? 하는 의문을 가졌던 적이 있었을 정도였다.

마이크로소프트의 윈도 프로그램은 사실, 역사 깊은 프로그램이다. 초창기 윈도 프로그램은 지금처럼 운영체제가 아니라 단지 MS-DOS 상에서 수행되는 응용프로그램이었다. 당시로는 파격적으로 마우스를 이용한 인터페이스와 하나의 화면에 여러개의 창이 떠서 동시에 여러 작업을 할 수 있었다. 하지만, 상업적으로 별로 성공을 거두지는 못했다.

본격적으로 윈도 시대를 연 것은 바로 윈도 3.0 이었다. 윈도 3.0 은 최초로 운영체제 성격을 띄기 시작했다고 볼 수 있는데, MS-DOS 를 기반으로 수행되기는 하지만 윈도가 기동되고 나면 나름대로의 메모리 관리 등으로 MS-DOS 와는 전혀 다른 환경을 응용프로그램에 제공해 주었던 것이었다. 여기에 멀티미디어 기능등을 일부 추가해서 윈도 3.1 이 발표되었는데, 이것은 당시의 최상위 프로세서였던 386 아키텍쳐의 향상된 기능들을 마음껏 이용할 수 있는 최초의 IBM-PC 용 운영체제였다.

큰 인기를 끌면서 많은 사용자들을 확보하기는 했지만 윈도 3.1 은 사용자 인터페이스 측면에서나 멀티태스킹 운영체제로서나 미흡한 면이 많았다. 그래픽 사용자 인터페이스는 애플 매킨토시 운영체제에 비하면 초라한 것은 물론이고 유치하기까지 하다고 생각될 정도였으며, 멀티태스킹 환경이라고 대대적으로 선전하기는 했지만 사실, 두 세 개 이상의 응용프로그램을 수행시키기에도 벅찼다고 말할 수 있다.

윈도 3.1 의 멀티태스킹

멀티태스킹이란, 말그대로 여러 작업을 동시에 수행한다는 뜻이다. MS-DOS 시절에는 하나의 프로그램을 수행해 놓으면, 그 프로그램을 종료하기 전에는 다른 프로그램을 수행할 수 없었다.

윈도 3.1 의 멀티태스킹은 작업 중인 프로그램을 종료시키지 않고도 또 다른 프로그램을 기동시킬 수 있으며, 작업 도중에 한 응용프로그램에서 다른 응용프로그램으로 작업 전환도 자유롭다는 점에서 당시에는 획기적인 일이었다.

하지만, 실제로 윈도 3.1 에서 응용프로그램을 사용하다보면, 과연 제대로 멀티태스킹이 되는 것인지 의심스러울 때가 많다. 여러개의 응용프로그램을 띄워 동시작업을 할 때, 배경에서 수행되는 작업이 현저하게 느리게 수행된다거나 또는 심지어 어떤 프로그램이 수행되는 도중에는 배경에 있는 프로그램이 전혀 수행되지 않게 되는 경우도 심심치 않게 볼 수 있었다

그렇다면 윈도 3.1 은 어떤 방식의 멀티태스킹을 하는 것일까? 프로그램을 수행시켜 주는 프로세서는 하나 뿐인데, 여러개의 프로그램을 동시에 수행시키려면 어떻게 해야 할까? 가장 간단히 생각해 볼 수 있는 것이 시분할 방식이다. 시분할 방식이란 프로세서가 시간을 잘게 나누어 각 프로그램들 수행을 조금씩 해 주는 방식을 말한다. 이것이 빠른 속도로 이루어지면 프로그램들이 동시에 수행되고 있다는 느낌이 들 것이다. 다른 방법으로는, 프로그램들이 서로 협력하면서 자신이 일을 하고 있지 않을 때 다른 프로그램에게 프로세서의 사용권을 넘겨주어 동시 작업을 하는 협력형 방식이 있다. 윈도 3.1 에서 도스용 프로그램들은 시분할방식으로 멀티태스킹을 하며, 윈도용 프로그램들은 협력형 방식으로 멀티태스킹을 한다.

협력형 또는 비선점형 멀티태스킹

협력형 멀티태스킹 환경에서는 하나의 프로그램에서 다른 프로그램으로 프로세서 사용권이 넘어가려면 명시적으로 특정한 시스템 함수를 호출해 주어야 한다. 윈도 응용프로그램에서는 GetMessage, PeekMessage, WaitMessage 같은 함수가 바로 이 역할을 하는 것들인데, 만약 어떤 한 응용프로그램이 이들 함수 중 하나를 오랫동안 호출하지 않는다면 다른 응용프로그램들은 CPU 를 사용할 기회를 얻을 수가 없게 된다. 예를 들어, 복잡한 계산이나 시

간이 많이 걸리는 인쇄 작업을 하는 경우에 그 작업 중에 GetMessage, PeekMessage, WaitMessage 를 불러 주지 않으면, 다른 응용프로그램들은 CPU 사용권을 얻을 수 없어 전혀 작업이 이루어지지 않게 된다. 따라서 수행 중인 다른 프로그램들에게 해를 끼치는 깡패같은 프로그램을 작성하는 것은 아주 쉬운 일이다. 한참동안 GetMessage, PeekMessage, WaitMessage 중 하나를 불러 주지 않으면 되는 것이다. 만약 배경에서 통신 응용프로그램이 자료를 내려받기(download)하고 있었다면 십중팔구 문제가 발생할 것이다. 이런 이유로 해서 협력형 멀티태스킹을 하는 윈도 시스템에서 수행되는 윈도 응용프로그램은 기본적으로 메시지 루프라고 하는 부분을 꼭 가져야 한다. 또한 프로그램 중에 어떤 루프를 일정시간 이상 돌게 되는 경우가 생긴다면, 위와 같은 이유로 해서, 이 부분에도 역시 메시지 루프를 넣어 주어야 한다. 앞에서 생각해 보았듯이 많은 양을 인쇄하는 경우나 시간이 많이 걸리는 연산이 행해지는 부분에는 꼭 메시지 루프를 넣어 주어야 한다. 이런 경우에는 다음과 같은 방식으로 코딩을 해야 한다. 시간이 많이 걸리는 작업이 시작하는 순간에 취소 버튼을 갖는 대화상자를 하나 보여 준다. 작업이 끝나면 대화상자를 없애는데 작업이 끝날 때까지 시간이 많이 걸린다면, 그 동안 다른 응용프로그램으로 CPU 사용권이 넘어가지 않는다. 다른 프로그램이 CPU 를 사용할 수 있도록 하려면 작업이 수행되는 부분 안에 메시지 루프를 포함시켜 주어야 한다.

```
BOOL fAbort = FALSE;
lpProcAbortDlg = MakeProcInstance(AbortDlg, hInst);
hAbortDlgWnd = CreateDialog(hInst, "취소 상자", hWnd, lpProcAbortDlg);
ShowWindow(hAbortDlgWnd, SW_NORMAL);
UpdateWindow(hAbortDlgWnd);
EnableWindow(hWnd, FALSE);

/*
 * ......
 * ......
 * 시간이 많이 걸리는 작업을 위한 준비를 한다.
 */

while(!fAbort)
{
        /*
         * ..........
         * ..........
         * 작업을 수행한다. 위의 while 문을 통해 반복되는 작업이어야 한다.
         */
        while(PeekMessage((LPMSG)&msg, NULL, NULL, PM_REMOVE))
        {
                if(IsDialogMessage(hAbortDlgWnd, (LPMSG)&msg))
                {
                        TranslateMessage((LPMSG)&msg);
                        DispatchMessage((LPMSG)&msg);
                }
```

```
        }
}
EnableWindow(hWnd, TRUE);
DestroyWindow(hAbortDlgWnd);
FreeProcInstance(lpProcAbortDlg);
```

윈도 95

윈도 3.1 이 큰 인기를 끌긴 했지만 프로그램 관리자의 불편함과 유치한 인터페이스는 언제나 사용자들의 불평을 들어 왔다. 또한 대대적으로 선전했던 멀티태스킹도 알고보니 그 성능이 형편없는 것이었다. 마이크로소프트는 새로운 윈도 운영체제를 개발하겠다고 약속했는데, 그 발표가 계속 늦어지다가 95 년도 하반기에야 윈도 95 라는 이름으로 발표되었다. 윈도 95 는 32 비트 운영체제이며 선점형 멀티태스킹을 지원한다고 발표되었다.

윈도 95 의 멀티태스킹

윈도 3.1 이 멀티태스킹 운영체제라고 하지만, 사실 윈도 3.1 는 협력형 또는 비선점형 운영체제라고 해서 응용프로그램들의 협력하에만 가능한 멀티태스킹이었다. 윈도 버전에서 최초의 선점형 멀티태스킹 운영체제는 윈도 NT 다. NT 가 개인용으로 보다는 기업에서 서버용 운영체제로 사용되고 있는 현실을 생각해 보면, 윈도 95 가 나오게 됨으로써 비로소 PC 를 사용하는 많은 사람들이 선점형 운영체제의 강력한 성능을 맛보게 되었다고 하겠다.

선점형 멀티태스킹

협력형 멀티태스킹과는 달리 선점형 멀티태스킹 환경에서는 운영체제가 언제라도 응용프로그램의 수행을 막을 수 있다. 운영체제의 권한이 비선점형에 비해 막강해 졌다고 할 수 있다. 따라서 응용프로그램이 CPU 사용권을 다른 응용프로그램에 넘겨 주기 위해 GetMessage 류의 함수를 호출하지 않아도 된다. 선점형 운영체제에서는 응용프로그램이 시

간이 많이 걸리는 루프를 돌더라도, 다른 응용프로그램의 메시지 처리나 수행에는 아무 영향을 미치지 못한다.

선점형 멀티태스킹은 보통 일정하게 발생하는 타이머 인터럽트 같은 하드웨어 인터럽트를 사용해서 구현된다. 프로세스가 인터럽트를 받으면 그 제어를 운영체제에 넘기는 식으로 말이다.

윈도 95 와 윈도 NT 의 멀티태스킹, 멀티스레딩은 기능적으로 거의 차이가 없다. 다만 윈도 NT 는 윈도 95 와는 달리 16 비트 시스템을 고려하지 않았다는 점과 보완 기능이 윈도 95 에 비해 뛰어나다는 점이 두드러진 차이점이다.

윈도 3.1 이 비선점형 멀티태스킹이긴 하지만 가상 기계(virtual machine) 사이의 선점형 멀티태스킹은 이미 이루어지고 있었다. 가상 기계는 80386 마이크로프로세서 고유의 개념이다. 80306 마이크로프로세서는 가상 8086 상태에서 동작될 수 있다. 가상 8086 상태는 80386 이상의 프로세서가 하나 이상의 real-mode 8086 프로세서를 흉내낼 수 있도록 해준다. 윈도 3.1 은 이런 80386 이상의 프로세서 기능을 이용해서 여러개의 동시에 동작하는 MS-DOS 를 사용한다. 즉, 흔히 MS-DOS 프롬프트라고 부르는 도스 창들은 각각 시분할 방식으로 동작했다는 말이다.

도스 프로그램은 사실, 협력형 멀티태스킹에 포함될 수 없다. 왜냐하면 도스 프로그램에 GetMessage 같은 부분이 있을 리 없기 때문이다. 윈도 3.1 에서도 도스 세션과 윈도 시스템(윈도 시스템 자체도 하나의 가상기계로 취급된다) 사이에는 선점형 멀티태스킹 기구가 사용되었다.

윈도 95 프로그래밍

윈도 95 가 Win32 환경이라고는 하지만, 엄밀히 말해 완전히 Win32 는 아니고 Win32c 라는 Win32s 와 Win32 의 중간정도의 단계이므로, Win32 프로그래밍이라는 말보다는 윈도 95 프로그래밍이란 용어를 사용하도록 하겠다. 윈도 95 프로그래밍에 대해 살펴보기 전에 먼저 윈도 3.1 프로그래밍의 개념 정도만 짚고 넘어가도록 하자.

윈도 3.1 프로그래밍

윈도 3.1 프로그래밍에서 주의해야 할 점은 크게 다음의 두가지로 생각할 수 있다.

1. 윈도 3.1 은 협력형, 또는 비선점형 운영체제이기 때문에 응용프로그램에 대해 강력한 영향력을 행사할 수가 없다. 어떤 한 프로그램이 시스템 자원을 받아 사용하면서 이를 놔 주지 않는다하더라도 운영체제는 이것에 대해 아무런 조치도 취할 수 없다. 윈도 3.1 응용프로그램이 윈도 시스템에서 잘 수행되도록 하려면 응용프로그램을 만들 때, 착실히 잘 수행되도록 사려깊게 만들어야 한다는 말이다.

2. 윈도가 호출하는 함수는 PASCAL 형의 호출규약을 따라야 한다. 윈도가 호출하는 함수의 대표적인 예는 WinMain 함수이다. 이 함수는 꼭 PASCAL 호출 규약을 따라야 한다.

윈도 3.1 프로그래밍에서 중요한 개념들을 먼저 살펴보자. 이 내용들은 윈도 3.1 에만 국한된 내용은 아니기 때문에 초보자라면 잘 읽어보도록 하자.

핸들 (handle)

도스에서 프로그램을 하다가 윈도 프로그램을 처음 접하게 되는 사람들이 가장 당황해 하는 것은 아마도 수많은 새로운 용어들과 또 수많은 새로 정의된 타입(type)들과 구조체들의 홍수가 아닐까 생각한다. 핸들이란 용어도 도스 프로그래밍에서는 접하기 힘든 용어였을 것이다.

윈도 시스템에서 제공하는 여러 가지 자원들을 이용하려면, 그 자원을 사용하기 전에 먼저, 사용하고자 하는 자원의 핸들을 윈도 시스템으로부터 얻어내야 한다. 프로그래머가 직접 시스템 자원에 접근하지 않고, 핸들이라는 매개를 거치는 이유는 이렇다. 시스템으로부터 핸들을 얻어 프로그래밍을 하면, 시스펨 자원의 내용이 수정되더라도 프로그램을 이에 따라 일일이 수정해 주지 않아도 되기 때문이다.

인스턴스(instance)

윈도 시스템에서 인스턴스는 수행되어 메모리에 올라와 있는 프로그램을 말한다.

윈도 3.1 에서는 하나의 프로그램이 여러번 수행될 수도 있는데, 쉬운 예로 메모장 같은 프로그램을 수행시켰다고 해보자. 이미 메모장이 실행 중인데, 다시 한번 메모장 프로그램을 수

행시키면 메모리에 다시 똑같은 내용의 프로그램이 올라올까? 그렇지 않다. 이미 실행 중인 메모장 프로그램이 있다면 두 번째 실행에서는 메모장 프로그램의 데이터 영역만을 새로 할당하고, 실행 코드 부분은 첫 번째 것을 공유하게 된다. 이렇게 함으로써 메모리를 아낄 수는 있었겠지만, 한쪽 메모장 프로그램에 어떤 오류가 발생하게 되면 다른 쪽에도 그 영향이 곧바로 미치게 되는 것은 어쩔 수가 없게 된다.

가장 먼저 인스턴스를 접하게 되는 곳은 바로 WinMain 에서 이다.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpszCmdParam, int nCmdShow)
```

hInstance 는 이 WinMain 함수로 수행되게 되는 프로그램의 인스턴스이고, hPrevInstance 는 이 프로그램이 수행되기 전에 똑같은 프로그램이 이미 메모리에 떠 있다면 바로 그 프로그램의 인스턴스 핸들값을 가리킨다. 따라서 이 인수가 0 이 아니라면, 이미 같은 프로그램이 수행중이라는 의미가 된다.

메시지(message)

윈도 시스템 안에서 일어나는 모든 일은 메시지(message)를 통해 전달된다. 마우스를 누르거나 움직인다든가 키보드를 누른다든가 하는 사용자의 입력과 시스템 내부에서 일어나는 모든 일들이 바로 이 메시지 형태로 변환되어 전달된다. 메시지를 전달할 때, 메시지 그 자체 이외의 부가 정보를 전달하고자 할 때 메시지 매개변수를 사용하는데, 부가 정보라는 것은 예를 들어 키보드를 눌렀다는 메시지가 발생하여 이를 전달하고자 할 때, 키보드에서 어떤 키가 눌렸는지를 알려 주어야 하는데, 이런 정보를 말한다. 윈도 시스템에서 사용되는 메시지를 나타내는 구조체는 다음과 같이 정의되어 있다.

```
typedef struct  tagMSG
{
    HWND hwnd;
    WORD message;
    WORD wParam;
    LONG lParam;
    DWORD time;
    POINT pt;
}           MSG;
```

hwnd 는 메시지를 받을 창(window)의 핸들이고, message 는 메시지번호, wParam 과 lParam 은 각각 부가 정보를 전달할 WORD 형 변수와 LONG 형 변수이다. time 은 이 메시지가 발생한 시간이고, pt 는 메시지가 발생한 시점에서 마우스의 위치를 가리킨다. 그렇다면 어떤 메시지의 경우에 wParam 을 사용하고, 어떤 메시지의 경우에 lParam 을 사용하는지는 어떻게 알 수 있는가? 함수 사용 설명서를 뒤져 보는 수밖에 다른 방법은 없다.

메시지 큐(message queue)

메시지가 발생되어 해당되는 창(window)으로 전달되었을 때 그 창이 미처 그 메시지를 처리하지 못한 상태에서 새로운 메시지가 또 발생하여 전달되게 되면 어떻게 될까? 이전의 메시지를 처리하느라 새로 전달된 메시지는 읽어가지 못하는 것은 아닐까? 언제나 전달된 메시지를 그때 그때 착착 처리할 수는 없는 일이기 때문에 들어오는 메시지들을 담아둘 곳이 필요하다. 이런 장소를 메시지 큐라고 하는데, 시스템은 발생한 메시지를 이 메시지 큐에 넣어주고, 응용프로그램은 이 메시지 큐에서 메시지들을 읽어가게 된다. 큐라는 자료구조는 항상 스택과 비교되는데, 스택이라는 자료구조는 가장 나중에 들어간 자료가 가장 먼저 나오는 구조이다. 이와는 달리 큐는 먼저 들어간 자료가 먼저 나오는 (First In First Out:FIFO)구조로 되어 있다. 따라서 발생된 메시지가 큐에 저장되게 되면 가장 먼저 도착한 메시지부터 차례로 처리할 수 있게 된다. 윈도의 메시지 큐는 시스템 큐와 응용프로그램 큐 두가지로 구분할 수 있는데, 시스템 큐는 시스템에서 발생하는 모든 메시지를 보관하는 곳이고, 응용프로그램 큐는 각 응용프로그램이 갖고 있는 큐인데, 시스템 큐에서 응용프로그램으로 보내오는 메시지를 담아두는 곳이다. 예를 들어, 두 개의 응용프로그램이 실행되어 있는데, 각 응용프로그램이 각각 하나의 창을 갖는다고 하자. 마우스로 각 창을 한번씩 눌러주면 마우스 왼쪽 버튼 클릭 메시지인 WM_LBUTTONDOWN 이 발생하게 되는데, 이렇게 발생된 메시지는 먼저 시스템 큐에 들어가게 된다. 그 다음에 적당한 순간에 시스템은 자신의 큐에 있는 메시지들을 해당 창을 소유하고 있는 응용프로그램의 메시지 큐에 전달해 주게 된다. 프로그래머 입장에서 시스템 큐와 응용프로그램 큐를 모두 고려해 주어야 할 필요는 없다. 단지 자신의 메시지 큐만을 읽어 와서 해당 메시지에 대해 적절한 행동을 해주면 된다.

SendMessage 와 PostMessage

키보드나 마우스 메시지처럼 시스템 큐쪽에서 전달되는 메시지도 있지만 다른 응용프로그램쪽에서나 같은 응용프로그램 내부에서 전달되는 메시지도 있다. 이렇게 메시지를 보낼 때 쓰는 API 로는 SendMessage 와 PostMessage 가 있다. 이 둘의 차이점은 다음과 같다. SendMessage 는 메시지를 메시지 큐로 보내지 않고 직접 윈도 프로시저로 메시지를 전달한다. 따라서 메시지 큐에서 대기 중인 메시지보다 먼저 처리된다. 메시지 큐에 메시지를 넣어 주는 것이 아니라 윈도 프로시저를 직접 호출해서 메시지 처리를 하기 때문에 윈도 프로시저 내에서 처리가 다 끝나기 전에는 종료되지 않는다. 즉, 다시 말해서 메시지를 보내고 그 메시지에 대한 처리가 끝날 때까지 다음 처리를 하지 않게 된다는 말이다. 따라서 꼭 어떤 메시지에 대한 처리가 완료된 다음, 다음 메시지의 처리를 해 주어야 하는 경우에 사용하면 된다. 단, 메시지 처리가 완료된다는 보장이 있어야 한다. 그렇지 않으면, 다음 처리가 불가능하기 때문이다.

PostMessage 는 말그대로 메시지를 부치는(post) 것이다. 우체통에 쏙 넣어버리고 나서는 곧바로 돌아서버리고 자기 일을 봐도 되는 것이다. 즉, 메시지를 받는 쪽의 메시지 큐에 메시지를 전달만 한다. SendMessage 처럼 메시지에 대한 처리가 끝날 때까지 기다리지 않는다는 뜻이다. 따라서 PostMessage 한 메시지는 이미 메시지 큐에 메시지들이 쌓여 있다면 즉시 처리되지 않고, 이전에 도착된 메시지들부터 차례로 처리되게 된다.

메시지 루프(message loop)

응용프로그램의 메시지 큐에 전달된 메시지들을 응용프로그램은 어떻게 읽어 들일까? 메시지 큐에 있는 메시지들을 읽어 오는 부분을 메시지 루프라고 하는데, 이 메시지 루프는 GetMessage 함수를 계속해서 부르는데, 이 함수는 WM_QUIT 메시지를 메시지 큐에서 읽어오게 되면 거짓값을 돌려준다. 따라서 메시지 루프는 WM_QUIT 메시지를 받게 되면 메시지 루프를 종료하게 된다. 이 메시지 루프는 WinMain 함수의 끝부분에 위치하는게 보통인데, 따라서 메시지 루프를 종료한다는 말은 곧 응용프로그램을 종료하게 된다는 말이 된다.

```
while(GetMessage(&msg, NULL, 0, 0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

메시지 큐에서 WM_QUIT 이외의 메시지를 읽어 오게 되면 먼저 가상키입력을 문자 메시지

로 변환하고(TranslateMessage), 이어 메시지를 윈도 프로시저로 보내게(DispatchMessage)된다. 이 메시지 루프에서 읽어올 메시지가 없다면, 즉 메시지 큐가 비었다면, 시스템은 다른 응용프로그램으로 CPU 사용권을 넘겨주고 다른 응용프로그램은 다시 자신의 메시지 큐를 검사해서 처리할 메시지가 있는지 알아본다. 있다면 그 응용프로그램이 윈도 프로시저를 통해 메시지를 처리하고 난 후 다시 또 다른 응용프로그램에 CPU 사용권한을 넘겨주는 방식으로 윈도 응용프로그램 간에 멀티태스킹을 구현하게 된다. 따라서 어떤 하나의 응용프로그램에 아주 빈번히 메시지가 발생해서 미처 다른 응용프로그램에 메시지 처리 기회를 주지 않는다면, 기회를 거의 받지 못한 응용 프로그램은 아주 느리게 동작하거나 거의 멈춰 있게 되는 현상이 일어나게 된다.

윈도 프로시저(window procedure)

하나의 창(window)에는 그 창에 발생하는 메시지를 처리하는 함수 즉, 윈도 프로시저가 있게 된다. 이 윈도 프로시저는 윈도 클래스를 등록할 때 설정하게 된다. 보통은 다음과 같은 구조를 갖는다.

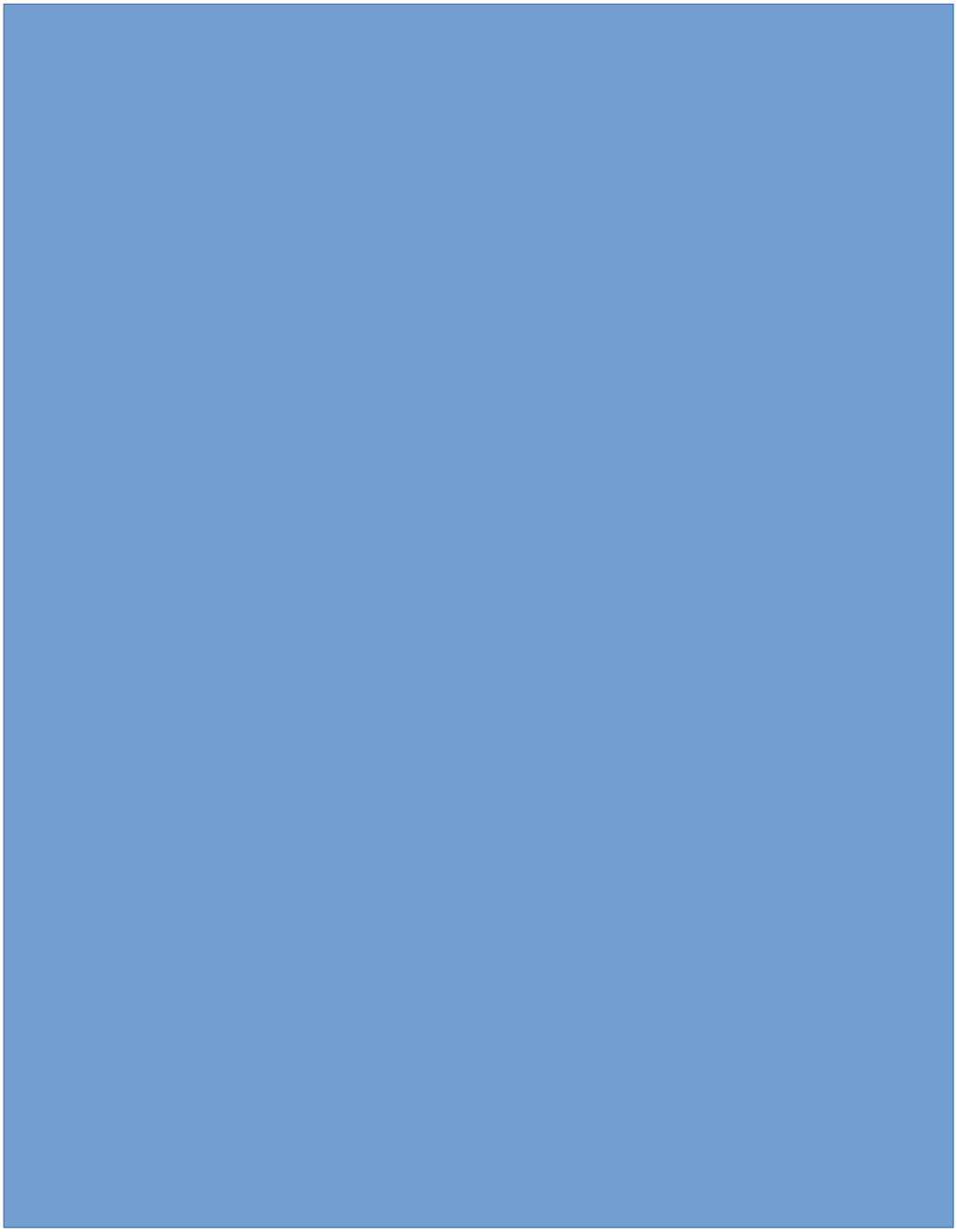
```
long FAR PASCAL MainWndProc(HWND hwnd, unsignend uMsg, WORD, wParam
LONG lParam)
{
        switch(uMsg)
        {
                case WM_PAINT:
                        .....
                        .....
                case WM_CREAT:
                        .....
                        .....
                default:
                        .....
                        .....
        }
}
```

윈도 프로시저는 발생하는 메시지들에 따라 적당한 일을 해주는 부분이기 때문에 일반적으로 전달된 메시지에 대해 switch-case 문으로 구성된다.

헝가리식 표기법(hungarian notation)

규모가 큰 프로그램을 짜거나 또는 다른 사람이 짠 프로그램을 분석해 보다 보면 수많이 만나게 되는 변수들이 정수형 변수인지 실수형 변수인지 아니면 또 다른 어떤 변수인지 쉽게 알 수 없다. 때문에 꼼꼼한 프로그래머들은 나름대로 변수 작명 규칙을 정해 놓고 쓰기도 했는데 이런 방법은 다른 프로그래머들에게는 오히려 암호처럼 보이기도 했다. 윈도 프로그래밍에서 변수 작명 규칙에 대해서 거의 표준으로 생각되다시피 한 방법이 있는데 이것이 바로 헝가리식 표기법이다.

변수이름의 앞부분에 변수의 종류를 알아볼 수 있는 기호같은 것을 붙여서 이 변수의 형이 무엇인지 한눈에 알아볼 수 있도록 하는 변수 표기법을 말한다. 찰스 시모니라는 프로그래머가 즐겨 사용했으며, 이 사람이 헝가리 사람이었다는데서 유래되었다고 한다. 이 표기법은 소스코드의 가독성을 좋게 한다고 해서 많은 사람들의 지지를 받고 있으며, 특히 윈도 프로그래밍에서는 널리 사용되고 있다.



데이터 형

윈도 95 프로그래밍

객체(object)와 핸들(handle)

객체는 파일이나, 쓰레드, 그래픽 이미지 같은 시스템 자원을 대표하는 내부 구조체 이다. 응용프로그램은 그 객체가 대표하는 시스템 자원이나 그 객체의 내부 구조에 직접 접근할 수는 없다. 그렇다면 이들 객체에는 어떻게 접근할 수 있을까? 이들 객체에 접근하려면 시스템으로부터 이들 객체에 대한 핸들값을 먼저 얻어내야 한다.

시스템 입장에서는 보면, 응용프로그램들이 시스템 자원에 직접 접근하지 않고 언제나 시스템에서 자원에 대한 핸들값을 얻어 사용하기 때문에 시스템 자원의 내부 구조를 응용프로그램에서 독립시킬 수 있었다. 개발자들이 시스템 자원의 내부 구조의 값을 수정할 수 없게 함으로써 운영체제의 기능이 변화하거나 추가되었을 때도 응용프로그램은 전혀 수정할 필요가 없어지는 잇점이 있다.

윈도 95 에서는 프로세스가 종료하면, 그 프로세스가 만들었던 모든 객체들을 없애고, 열려있는 객체의 핸들을 모두 닫아 준다. 하지만 쓰레드 종료 때에는 이런 일이 일어나지 않는다. 윈도 객체나 DDE 객체같은 몇가지 예외적인 객체들은 이 객체를 만든 쓰레드가 종료되면 자동으로 삭제된다.

윈도에서 사용되는 객체는 크게 사용자(user) 객체, 그래픽 장치 인터페이스(Graphic

Device Interface:GDI) 객체, 커널(kernel) 객체, 세가지로 나눌 수 있다. 사용자 객체는 창(window) 관리를 위해서, GDI 객체는 그래픽을 지원하기 위해서, 커널객체는 메모리 관리, 프로세스 실행, 프로세스간 통신등을 지원하기 위해 사용한다.

사용자 객체나 GDI 객체는 언제나 한번에 하나의 핸들만을 지원한다. 시스템은 응용프로그램이 객체를 만들 때 그 객체에 대한 핸들을 제공하고, 응용프로그램이 객체를 지울 때 그 핸들을 회수한다. 반면 커널 객체는 하나의 객체에 대해 여러개의 핸들을 지원한다. 운영체제는 객체에 대한 마지막 핸들이 닫히면, 자동적으로 그 객체를 메모리에서 제거한다.

사용자 객체와 GDI 객체는 하나의 객체에 대해 오직 한 개의 핸들만을 지원한다. 따라서 프로세스들은 사용자 객체와 GDI 객체에 대한 핸들을 상속하거나 복사할 수 없다.

사용자 객체에 대한 핸들은 어떤 프로세스에서라도 접근할 수 있다. 반면에 GDI 객체에 대한 핸들은 어떤 한 프로세스에 속하게 된다. 즉, GDI 객체를 만든 프로세스만이 그 객체 핸들을 사용할 수 있다.

창(window)객체를 하나 만들면, 창에 어떤 내용을 표시하거나 그 모양을 변화시키기 위해 그 창의 핸들을 이용하게 되는데, 이 핸들은 창 객체를 만드는 함수 CreateWindow 함수가 반환하는 값이다. DestroyWindow 함수를 이용해서 창 객체를 메모리에서 제거하면 그 창의 핸들도 이제 사용할 수 없게 된다.

32 비트 프로그래밍 개요

32 비트 API 는 16 비트로 작성된 프로그램을 될 수 있는 한 최소한만 수정해도 되도록 설계되었지만, 커진 메모리 공간 때문에 몇가지 변화가 생겼다. 잘 알다시피 포인터는모두 32

비트로 되어, 그 동안 골치아팠던 세그먼트 메모리(segmented memory)는 아주 잊어버려도 좋다. 따라서 near 니 far 니 하는 구분도 없어지고 말하자면 모든 포인터는 far 포인터가 되어버린 것이다.

32 비트로 된 항목에는 다음과 같은 것들이 있다.

창의 핸들(window handle)

펜, 브러시, 메뉴 같은 객체들의 핸들

그래픽 좌표

윈도 3.1 과 윈도 95 의 윈도 프로시저

윈도 3.1 의 윈도 프로시저는 다음과 같다.

```
LONG FAR PASCAL MainWndProc(HWND hwnd,
        unsigned message,
        WORD wParam,
        LONG lParam)
```

윈도 95 에서 사용되는 윈도프로시저는 다음과 같다.

```
LRESULT CALLBACK MainWndProc(HWND hwnd,
        UINT message,
        WPARAM wParam,
        LPARAM lParam)
```

둘 사이의 차이점을 정리해 보면 다음 표와 같다.

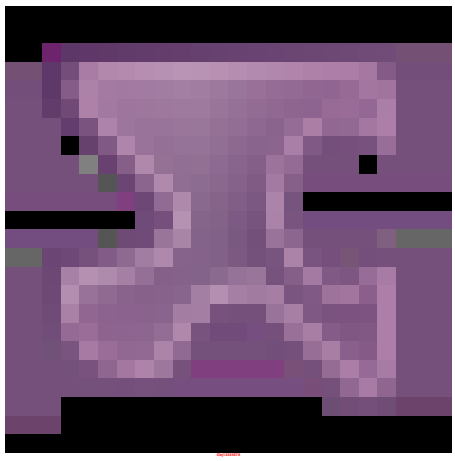
Win32 에서 새로 정의되어 윈도 95 에서 사용되는 표준 형(type)들은 아래 표와 같은 것들이 있다.

윈도 3.1 과 윈도 95 윈도 프로시저에서 가장 큰 차이점은 바로 wParam 이다. 윈도 95 같은 Win32 환경에서는 wParam 이 32 비트로 커졌다. 이전 16 비트 환경에서는 wParam 이 WORD 형(16 비트)이었다. 이름이 여전히 wParam 이라고 해서 WORD 형이라고 생각하면

안된다. wParam 은 32 비트 WPARAM 형이라고 생각해야 한다.

32 비트 메시지 처리

윈도 3.1 에서는 16 비트인 WORD 형 wParam 에 메시지의 ID 를 넣고, LONG 형인 lParam 에 윈도 핸들과 WM_COMMAND 의 메시지를 16 비트 씩 나누어 담아 보냈다. 하지만 윈도 95 와 같은 32 비트 환경에서는 핸들이 모두 32 비트 크기로 늘어났다. 때문에 lParam 에 윈도 핸들을 넘겨주는 경우에 lParam 전체를 윈도 핸들이 사용하게 된다. 그래서 이전에 lParam 의 상위워드를 차지했던 부분이 wParam 의 상위워드로 옮겨졌다.



32 비트와 16 비트 프로세스

윈도 95 에서 수행되는 16 비트 프로세스들의 멀티태스킹은 어떻게 이루어질까? 윈도 95 에서 수행되는 16 비트 프로그램도 다른 32 비트 프로세스와 함께 선점형 멀티태스킹을 할까? 윈도 95 에서 수행되는 16 비트 프로세스들은 16 비트 프로세스만을 위한 별도의 환경에서 수행된다. 즉, 윈도 95 안에 가상의 윈도 3.1 시스템이 있다고 생각하면 된다. 윈도 3.1 응용 프로그램을 위한 별도의 가상기계가 존재하며 따라서 16 비트 프로세스들끼리는 여전히 비선점형 멀티태스킹을 하게 된다.

메시지 큐와 스레드

윈도 3.1 에서는 하나의 프로세스가 하나의 스레드만을 가질 수 있다. 시스템 메시지 큐도 하나만을 가질 수 있는데, 윈도 95 는 각 실행 스레드가 별도의 메시지 큐를 가질 수 있다. 따라서 윈도 95 의 프로세스들은 다른 프로세스와 동시에 메시지 처리하는데 아무런 지장을 받지 않게된다. 프로세스는 스레드를 포함한 개념이다. 하나의 프로세스에는 하나 이상의 스레드가 포함될 수 있다.

또 한가지 중대한 차이점은 SendMessage 에 대한 것이다. 윈도 3.1 에서는 SendMessage 가 단순히 메시지를 받는 윈도의 윈도 프로시저를 호출하는 것에 불과했지만 멀티 스레드 환경인 윈도 95 에서는 약간 사정이 달라진다. SendMessage 가 주 스레드가 아닌 스레드에서 호출되는 경우에, 윈도 프로시저를 호출하는 스레드가 아닌 윈도를 만든 스레드에서 이루어져야만 하는데, 윈도 95 는 호출된 스레드가 SendMessage 실행을 완료할 때까지 호출한 스레드를 중지시키는 기구를 제공한다.

그렇다면, 선점형 멀티태스킹 환경에서는 더 이상 시간이 오래 걸리는 작업이 일어나고 있을 때, 메시지 읽어가기를 하지 않아도 되는가? 대답은 '아니다'이다. 그 이유는 다음과 같다. 다음은 앞에서 들었던 예이다.

```
BOOL fAbort = FALSE;
lpProcAbortDlg = MakeProcInstance(AbortDlg, hInst);
hAbortDlgWnd = CreateDialog(hInst, "취소 상자", hWnd, lpProcAbortDlg);
ShowWindow(hAbortDlgWnd, SW_NORMAL);
UpdateWindow(hAbortDlgWnd);
EnableWindow(hWnd, FALSE);

/*
 * ......
 * ......
 * 시간이 많이 걸리는 작업을 위한 준비를 한다.
 */

while(!fAbort)
{
        /*
         * ..........
         * ..........
         * 작업을 수행한다. 위의 while 문을 통해 반복되는 작업이어야 한다.
         */
        while(PeekMessage((LPMSG)&msg, NULL, NULL, PM_REMOVE))
        {
                if(IsDialogMessage(hAbortDlgWnd, (LPMSG)&msg))
                {
                        TranslateMessage((LPMSG)&msg);
                        DispatchMessage((LPMSG)&msg);
```

```
                }
        }
}
EnableWindow(hWnd, TRUE);
DestroyWindow(hAbortDlgWnd);
FreeProcInstance(lpProcAbortDlg);
```

이 코드에서 while 루프 안에 있는 PeekMessage 를 빼버리면 어떻게 될까? 물론 윈도 95 가 선점형 멀티태스킹 환경이기 때문에 윈도 3.1 과는 달리 다른 프로그램으로 전환하거나(작업 표시줄을 통해), 다른 프로그램을 기동시키거나 하는 등의 작업을 자유롭게 할 수는 있다. 하지만, 그 시간이 걸리는 작업을 하고 있는 응용프로그램 자체는 계속 메시지를 처리하지 못하고 있기 때문에 작업을 취소할 수도 없고, WM_PAINT 같은 메시지도 처리하지 못하기 때문에, 다른 창에 가려지거나 하면 자신의 창을 다시 그리기 할 수도 없다.

선점형 멀티태스킹이란 것이 잘못된 응용프로그램으로부터 시스템을 보호해 주기는 하지만, 응답하지 않는 응용프로그램 자체에 무슨 해결책을 제시해 줄 수는 없다는 것을 명심해야 한다.