개발자를 위한 윈도우 후킹 테크닉

SendMessage 후킹 하기

지난 시간에 WH_GETMESSAGE 훅을 통해서 메시큐에서 메시지가 처리되는 과정을 후킹해 보았다. 이번 시간에는 WH_CALLWNDPROC, WH_CALLWNDRETPROC 훅을 통해서 SendMessage 의 처리 과정을 후킹하는 방법과 다수의 훅 프로시저를 설치한 경우에 동기화 하는 방법에 대해서 설명할 것이다.

목차

필자 소개

신영진 pop@jiniya.net

부산대학교 정보,컴퓨터공학부 4 학년에 재학 중이다. 모자란 학점을 다 채워서 졸업하는 것이 꿈이되 버린 소박한 괴짜 프로그래머. 병역특례 기간을 포함해서 최근까지 다수의 보안 프로그램 개발에 참여했으며, 최근에는 모짜르트에 심취해 있다.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: 마이크로소프트 비주얼 스튜디오 2003
기초지식: C/C++, Win32 프로그래밍
응용분야: 메시지 모니터링 프로그램

연재 순서

2006. 05 키보드 모니터링 프로그램 만들기
2006. 06 마우스 훅을 통한 화면 캡쳐 프로그램 제작
2006. 07 메시지훅 이용한 Spy++ 흉내내기
2006. 08 SendMessage 후킹하기
2006. 09 Spy++ 클론 imSpy 제작하기
2006. 10 저널 훅을 사용한 매크로 제작
2006. 11 WH_SHELL 훅을 사용해 다른 프로세스 윈도우 서브클래싱 하기
2006. 12 WH_DEBUG 훅을 이용한 훅 탐지 방법
2007. 01 OutputDebugString 의 동작 원리

필자 메모

필자가 처음 Visual C++을 공부한 것은 2000 년도 였다. 이상엽님의 기념비적인 저서 Visual C++ 6.0 Bible 을 사들고 집에서 공부하고 있었다. 그러나 사실 그 책은 처음 Visual C++을 접한 독자가 이해하기에 너무도 어려웠다. 필자가 처음 버튼의 이벤트 핸들러에다 정확한 코드를 집어넣는데만도 한참의 시간의 걸렸다. 아마 선배한테 물어봤으면 3 초만에 배웠을것을 물어보지 않아서 몇날 몇일을 고생한 것이다.

독자 여러분들은 필자와 똑 같은 실수를 되풀이 하지 않기를 바란다. 필자의 메일함(pop@jiniya.net)은 독자 여러분의 질문에 답하기 위해서 24 시간 열려있다. 그러니 강좌에서 모르는 부분이나 의문점이 생기면 언제든지 메일을 통해서 질문 하도록 하자. 또한 블로그(http://www.jiniya.net)를 통해서 지면 관계상 못한 이야기나 설명이 잘못된 부분에 대한 내용을 제공하고 있으니 그 곳도 참고하면 공부에 도움이 될 것 같다.

Intro

Windows 의 응용 프로그램이 가장 많이 사용하는 함수 하나를 꼽으라면 아마도 SendMessage 가 될 것 이다. 왜냐하면 대부분의 Windows 응용 프로그램이 GUI 기반이고, GUI 프로그램의 경우 필연적으로 메시지를 기반으로 동작하기 때문이다. 또한

SendMessage 의 경우 동기화 문제 때문에 PostMessage 나 다른 함수보다 많이 사용된다. 이번 시간에는 이러한 SendMessage 의 호출 과정을 후킹해 보도록 하자.

SendMessage 는 어떻게 동작할까?

SendMessage 의 경우 같은 쓰레드의 윈도우로 메시지를 보내는지 아니면 다른 쓰레드에 의해서 생성된 윈도우로 메시지를 보내는지에 따라 동작 방식이 조금 다르다. 우선 같은 쓰레드의 윈도우로 메시지를 보내는 경우에는 SendMessage 는 해당 메시지 프로시저를 직접 호출하고 리턴 값을 반환한다. 이 과정은 직접 함수를 호출하는 것과 다를 바가 없다.

다른 쓰레드에 의해 생성된 윈도우로 메시지를 보내는 경우에는 메시지를 받을 윈도우를 생성한 쓰레드의 메시지 큐에 메시지를 추가하고 해당 큐의 QS_SENDMESSAGE 플래그를 설정한다. 그리고는 자신의 응답 메시지 큐에 응답이 오기를 기다린다. 만약 이 과정에서 메시지를 받는 윈도우가 무한 루프나 기타 상황으로 메시지를 처리할 수 없는 상황이라면 SendMessage 를 호출한 쪽의 쓰레드도 같이 잠기게 된다.

메시지 처리 순서

SendMessage 도 단순히 메시지 큐에 메시지를 추가하는 것이라면 어떻게 PostMessage 보다 먼저 처리되는 것일까? 이에 대한 해답은 윈도우가 메시지를 검사하는 알고리즘에 있다. 쓰레드의 메시지 큐를 조작하는 함수로는 GetMessage 와 PeekMessage 가 있다. 이 함수들은 메시지 큐를 무작위적으로 검사하지 않는다. 아래와 같은 순서에 따라서 메시지 큐를 검사한다.

1. QS_SENDMESSAGE 플래그가 설정되어 있으면 해당 메시지를 적절한 윈도우 프로시저로 보낸다. 이 작업 후에 함수는 리턴 하지 않고 계속 다음 메시지를 기다린다.
2. 포스트 메시지 큐에 메시지가 있으면 인자로 전달된 MSG 구조체에 해당 메시지 정보를 복사한 후 리턴한다.
3. QS_QUIT 플래그가 설정되어 있으면 WM_QUIT 메시지를 리턴하고, QS_QUIT 플래그를 제거한다.
4. 하드웨어 입력 큐에 메시지(WM_KEYDOWN, WM_LBUTTONDOWN, ...)가 있다면 MSG 구조체에 해당 메시지 정보를 복사한 후 리턴한다.
5. QS_PAINT 플래그가 설정되어 있으면 WM_PAINT 메시지를 리턴한다.
6. QS_TIMER 플래그가 설정되어 있으면 WM_TIMER 를 리턴한다.

쉽게 말하면 SendMessage 로 전달된 메시지, PostMessage 로 전달된 메시지, PostQuitMessage 로 전달된 메시지, 하드웨어 입력 메시지, WM_PAINT 메시지, WM_TIMER 메시지의 순으로 처리된다는 것이다. 이를 통해서 PostMessage 보다 늦게 메시지 큐에 포스트된 SendMessage 가 왜 먼저 처리되는지를 알 수 있다.

WM_KEYDOWN, WM_CHAR, WM_KEYUP 의 처리 순서도 위의 알고리즘을 통하면 쉽게 이해할 수 있다. 기본적으로 사용자가 키보드를 눌렀다 떼게 되면 WM_KEYDOWN, WM_KEYUP 이 하드웨어 입력 메시지 큐에 추가된다. WM_KEYDOWN 이 TranslateMessage 를 통하면 적절한 형태의 WM_CHAR 메시지가 PostMesage 로 큐에 추가된다. 물론 이 과정에서 WM_CHAR 가 WM_KEYUP 보다 늦게 추가되었지만 위의 알고리즘의 우선순위 판정에 의해서 다음 번 처리되는 메시지는 WM_CHAR 가 되는 것이다.

무한 대기에 빠지지 않는 법

SendMessage 를 사용할 경우 메시지를 받는 대상 윈도우의 쓰레드가 무한 루프에 빠진 경우라면 같이 블록되는 단점이 있다. 이를 사전에 알아내거나 차단하는 방법과 관련된 함수들을 간단하게 살펴보도록 하자. 함수의 파라미터와 리턴 값에 대한 자세한 설명은 MSDN 을 참고하자.

```
BOOL IsHungAppWindow(HWND hWnd);
```

위 함수를 사용하면 SendMessage 를 보낼 대상 윈도우가 현재 메시지를 처리할 수 있는지 없는지를 알 수 있다. TRUE 를 리턴 하는 경우는 해당 윈도우가 더 이상 메시지를 처리할 수 없음을 의미한다. 따라서 이 경우에 SendMessage 를 호출하지 않으면 무한 대기에 빠지는 문제를 방지할 수 있다. 하지만 이렇게 하더라도 SendMessage 의 메시지 루프에서 무한 대기에 빠지는 경우는 해결책이 없다.

```
LRESULT SendMessageTimeout( … );
BOOL SendMessageCallback( … );
```

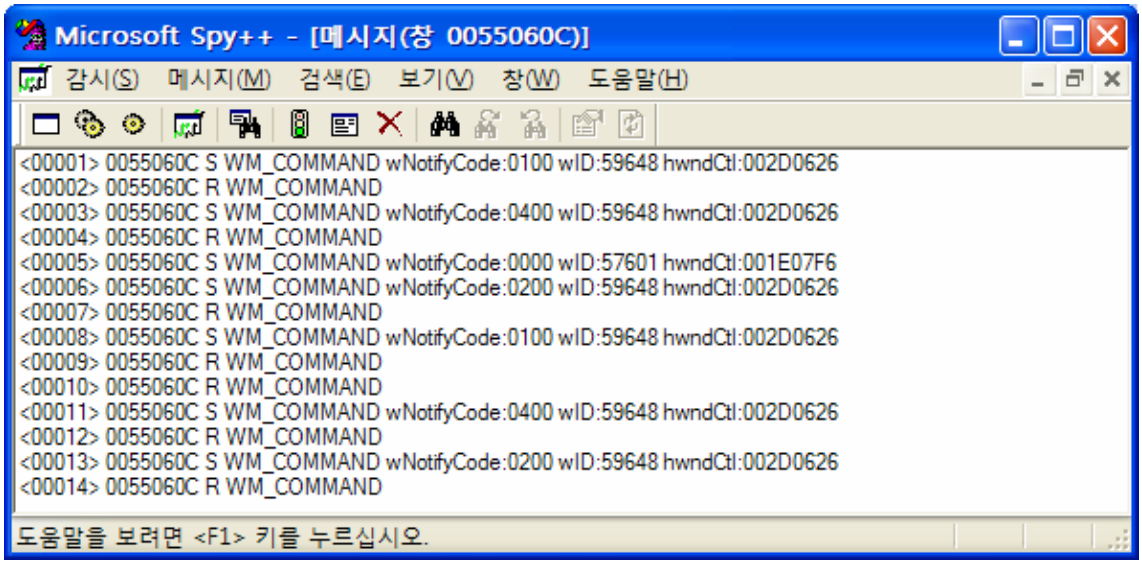
위 두 함수를 사용하면 어떠한 상태의 교착 상황도 피할 수 있다. SendMessageTimeout 의 경우는 타임아웃 시간을 지정해서 해당 시간이 지날 경우 자동으로 리턴한다. 두 번째 함수는 호출하는 즉시 리턴 하고, 메시지가 완료된 것은 콜백을 통해서 알 수 있다.

```
BOOL SendNotifyMessage(…);
BOOL ReplyMessage(LRESULT lResult);
BOOL InSendMessage(VOID);
DWORD InSendMessageEx(LPVOID lpReserved);
```

위의 함수들은 자주 사용하지는 않지만 알아두면 유용한 함수들이다. SendNotifyMessage 의 경우는 다른 쓰레드로 보낼 경우 바로 리턴하고 같은 쓰레드에서 생성한 윈도우로 보낼 경우에는 SendMessage 와 동일한 역할을 하는 함수다. ReplyMessage 는 메시지 함수가 처리가 완료되지 않은 시점에서 SendMessage 의 리턴 값을 전송하는 역할을 한다. InSendMessage 와 InSendMessageEx 함수는 서로 다른 쓰레드 간의 메시지 전송이면 TRUE 를 같은 쓰레드 간의 메시지 전송이면 FALSE 를 리턴 하는 함수다.

WH_CALLWNDPROC 훅

WH_CALLWNDPROC 훅은 SendMessage 를 호출하는 시점을 후킹한다. 즉, 이는 SendMessage 가 메시지 프로시저로 전달되어서 처리되기 전 시점을 후킹하는 것을 의미한다. Spy++을 보면 SendMessage 로 전달된 메시지에 대해서 S,R 플래그가 나오는 것을 볼 수 있다(화면 1 참고). S 의 경우 WH_CALLWNDPROC 을 통해서 SendMessage 를 호출하는 시점에 표시하는 것이고, R 은 다음 장에 소개되는 WH_CALLWNDRETPROC 을 통해서 SendMessage 가 리턴 되는 시점에 표시하는 것이다.



화면 1 Spy++을 통해서 SendMessage 과정을 모니터링한 화면

```
LRESULT CALLBACK CallWndProc(int code, WPARAM wParam, LPARAM lParam);
```

code – [입력] code 값이 HC_ACTION 인 경우 훅 프로시저를 수행하고, 0 보다 작은 경우에는 훅 프로시저를 수행하지 않고 CallNextHookEx 를 호출한 다음 리턴 해야 한다.

wParam – [입력] 메시지가 현재 쓰레드에 의해서 보내진 것인지 아닌지를 나타낸다. 현재 쓰레드에 의해서 보내진 경우 0 이 아닌 값을 가지고, 그렇지 않은 경우 0 으로 설정된다.
lParam – [입력] SendMessage 와 관련된 정보를 담고 있는 CWPSTRUCT 구조체의 포인터를 담고 있다. CWPSTRUCT 는 아래와 같은 형태를 하고 있다. 구조체의 필드별 의미는 <표 1>을 참고하자.

```
typedef struct {
    LPARAM lParam;
    WPARAM wParam;
    UINT message;
    HWND hwnd;
} CWPSTRUCT, *PCWPSTRUCT;
```

표 1 CWPSTRUCT 구조체 필드 별 의미

필드명	의미
hwnd	메시지 받을 윈도우 핸들
message	메시지 ID
wParam	메시지 WPARAM 파라미터
lParam	메시지 LPARAM 파라미터

리턴 값: code 가 0 보다 작은 경우에는 CallNextHookEx 의 리턴 값을 그대로 리턴 해야 한다. 그렇지 않은 경우에도 CallNextHookEx 의 리턴 값을 그대로 사용하는 것이 좋다. CallNextHookEx 를 통해서 다음 훅 체인을 호출하지 않은 경우엔 0 을 리턴 해야 한다.

WH_CALLWNDRETPROC 훅

WH_CALLWNDRETPROC 훅은 SendMessage 가 처리되고 리턴 되는 시점을 후킹한다. 따라서 메시지가 처리되고 난 후 어떤 값이 리턴되는지를 알 수 있다.

```
LRESULT CALLBACK CallWndRetProc(int code, WPARAM wParam, LPARAM lParam);
```

code – [입력] code 값이 HC_ACTION 인 경우 훅 프로시저를 수행하고, 0 보다 작은 경우에는 훅 프로시저를 수행하지 않고 CallNextHookEx 를 호출한 다음 리턴 해야 한다.
wParam – [입력] 메시지가 현재 프로세스 의해서 보내진 것인지 아닌지를 나타낸다. 현재 프로세스에 의해서 보내진 경우 0 이 아닌 값을 가지고, 그렇지 않은 경우 0 으로 설정된다.
lParam – [입력] SendMessage 의 리턴 정보를 담고 있는 CWPRETSTRUCT 구조체의 포인터를 담고 있다. CWPRETSTRUCT 는 아래와 같은 형태를 하고 있다. 구조체의 필드별 의미는 <표 2>를 참고하자.

```
typedef struct {
    LRESULT lResult;
    LPARAM lParam;
    WPARAM wParam;
    UINT message;
    HWND hwnd;
} CWPRETSTRUCT, *PCWPRETSTRUCT;
```

표 2 CWPRETSTRUCT 구조체 필드 별 의미

필드명	의미
hwnd	메시지 받을 윈도우 핸들
message	메시지 ID
wParam	메시지 WPARAM 파라미터
lParam	메시지 LPARAM 파라미터
lResult	SendMessage 리턴 값

WH_CALLWNDRETPROC 훅의 리턴 값은 WH_CALLWNDPROC 의 리턴 값과 동일 하다.

박스 1 메시지 차단

첫 시간에 작성한 키보드 후킹 프로그램에서 우리는 후킹 프로시저에서 1 을 리턴 함으로써 키보드 메시지가 호출되지 않도록 할 수 있다고 배웠다. 하지만 그것은 어디까지나 WH_KEYBOARD 에 국한된 이야기다. 이번 시간에 배운 WH_CALLWNDPROC, WH_CALLWNDRETPROC 모두 CallNextHookEx 를 호출하지 않는다고 해서 해당 메시지 호출이 중지되지 않는다. 단지 CallNextHookEx 를 호출하지 않게 되면 다음 번 훅 프로시저가 수행되지 않을 뿐이다. 따라서 WH_CALLWNDPROC 이나 WH_CALLWNDRETPROC 을 이용해서는 SendMessage 의 진행을 모니터링 할 수는 있지만 작업을 중간에 수정할 수는 없다.

다수의 훅을 설치해 보자

우리는 이제껏 하나의 훅 프로시저를 설치하는 프로그램만 작성했었다. 하지만 때로는 하나의 훅이 아닌 여러 개의 훅을 동시에 설치해야 하는 경우가 있다. Spy++이 그러한 대표적인 경우라 할 수 있다. Spy++의 경우 여러 개의 윈도우의 메시지 전송 과정을 동시에 모니터링할 수 있다. 이러한 기능을 제공하기 위해서는 여러 개의 훅을 설치하고 제거할 수 있도록 만들 필요가 있다. 물론 전역으로 훅을 설치할 경우 그러한 고민을 하지 않아도 된다. 하지만 WH_GETMESSAGE, WH_CALLWNDPROC, WH_CALLWNDPROCRET 등의 훅을 전역으로 설치하면 시스템이 굉장히 느려질 수 있다는 점을 기억해야 한다.

여러 개의 훅을 설치하고 제거하는 부분은 추후에도 사용하기 편리하도록 별도의 dll 로 제작했다. 앞으로 훅에 필요한 유틸리티 함수들은 hkutil.dll 에 추가해서 사용하도록 하자. <리스트 1>에는 훅의 설치와 제거에 사용될 구조체와 DllMain 부분이 나와있다.

HOOKINFO 구조체는 훅 정보를 관리한다. 사실 별도로 관리할 필요는 없지만 추후에 필요할 수도 있기 때문에 저장해 놓기로 했다. 또한 RegisterHook 을 통해서 설치한 훅 핸들만 추후에 제거할 수 있도록 하기 위해서도 어떤 것들을 설치되었는지는 알고 있어야 한다. 설치된 훅 종류와 해당 훅을 설치한 쓰레드 아이디가 저장된다. 이렇게 저장된 정보는 설치된 훅 핸들과 연관지어서 g_hookMap 에 저장된다.

DllMain 은 간단한 두 가지 처리를 해주고 있다. 포로세스에 로드될 때에 DisableThreadLibraryCalls 를 호출해서 쓰레드에 로드되는 부분을 호출하지 않도록 한다. 또한 프로세스에서 언로될 때에 g_hookMap 에서 제거되지 않은 훅 핸들이 있으면 제거해 준다.

리스트 1 훅 정보 구조체 및 DllMain 코드

```
// 훅 정보
typedef struct _HOOKINFO
{
   int        hookId;    // 훅 종류
   DWORD      threadId;   // 훅이 설치된 쓰레드
} HOOKINFO, *PHOOKINFO;

typedef std::map<HHOOK, HOOKINFO>         HookMap;
typedef std::map<HHOOK, HOOKINFO>::iterator HookMIt;

// 훅 정보를 저장할 맵
HookMap g_hookMap;

BOOL WINAPI
DllMain(HINSTANCE inst, DWORD reason, LPVOID)
{
   __try
   {
      switch(reason)
      {
      // 쓰레드에 대한 호출을 하지 않는다.
      case DLL_PROCESS_ATTACH:
         DisableThreadLibraryCalls(inst);
         break;

      // 종료시 제거되지 않고 남아있는 훅 핸들을 해제한다.
      case DLL_PROCESS_DETACH:
```

```
            for(HookMIt it=g_hookMap.begin(); it!=g_hookMap.end(); ++it)
            {
               UnhookWindowsHookEx(it->first);
            }
            break;
         }
      }
      __except(EXCEPTION_EXECUTE_HANDLER)
      {
         return FALSE;
      }

      return TRUE;
}
```

<리스트 2>는 RegisterHook 함수의 코드를 보여준다. 이 함수는 훅을 등록하는 역할을 한다. 기본적으로 SetWindowsHookEx 와 동일한 순서의 파라미터를 가진다. 단지 차이점이라면 훅 함수와 모듈명 대신에 훅 함수명과 dll 이름을 파라미터로 가진다는 것이다. 간단하게 LoadLibrary 로 dll 을 로드한 후 GetProcAddress 로 함수 주소를 구해서 해당 정보를 토대로 SetWindowsHookEx 를 호출하는 간단한 코드다. 성공적으로 추가된 경우에는 해당 정보를 전역 변수인 g_hookMap 에 저장해서 나중에 제거할 수 있도록 한다.

리스트 2 RegisterHook 코드

```
/*--
   훅을 등록한다.

   파라미터
      hookId - [입력] 훅 타입(WH_KEYBOARD, WH_MOUSE, ...)
      functionName - [입력] 훅 프로시저 이름
      dllName - [입력] 훅 프로시저를 담고 있는 dll 이름
      threadId - [입력] 훅을 설치할 쓰레드 ID

   리턴 값
      - 실패 시 NULL, 성공 시 훅 핸들

--*/
HHOOK WINAPI
RegisterHook(int hookId, LPCTSTR functionName, LPCTSTR dllName, DWORD threadId)
{
   HINSTANCE dll = NULL;   // dll 핸들
   HOOKPROC func = NULL;   // 훅 프로시저 함수 포인터
   HHOOK hook = NULL;     // 훅 핸들

   __try
   {
      // dll 을 로드 한다.
      dll = LoadLibrary(dllName);
      if(!dll)
```

```
            __leave;

        // 훅 프로시저를 로드 한다.
        func = (HOOKPROC) GetProcAddress(dll, functionName);
        if(!func)
            __leave;

        // 훅을 수행한다.
        hook = SetWindowsHookEx(hookId, func, dll, threadId);
        if(hook)
        {
            // 성공한 경우 훅 정보를 맵에 기록한다.
            HOOKINFO info;

            info.hookId = hookId;
            info.threadId = threadId;
            g_hookMap.insert(make_pair(hook, info));
        }
    }
    __finally
    {
        // dll 을 해제한다.
        if(dll)
            FreeLibrary(dll);
    }

    // 훅 핸들을 반환한다.
    return hook;
}
```

<리스트 3>은 훅을 제거하는 역할을 하는 UnregisterHook 의 코드다. 이 함수는 위에서 RegisterHook 을 통해서 등록한 훅 핸들을 제거하는 역할을 한다. 훅 핸들이 넘어오면 g_hookMap 에서 해당 훅 정보를 찾는다. 만약 해당 정보가 없다면 RegisterHook 에 의해서 등록된 훅 핸들이 아니기 때문에 FALSE 를 리턴한다. 맵에 존재하는 경우에는 맵에서 해당 정보를 삭제하고, UnhookWindowsHookEx 를 호출해서 훅 핸들을 제거한다.

리스트 3 UnregisterHook 코드

```
/*--
    등록된 훅을 해제한다.

    파라미터
        hook - [입력] 등록된 훅 핸들

    리턴 값
        - 성공 시 TRUE, 실패 시 FALSE

--*/
BOOL WINAPI
UnregisterHook(HHOOK hook)
```

```
{
    BOOL ret = FALSE;

    if(hook)
    {
        // 맵에서 훅 정보를 찾는다.
        HookMIt it = g_hookMap.find(hook);
        if(it != g_hookMap.end())
        {
            // 존재하는 경우 해당 정보를 삭제하고, 훅을 해제한다.
            g_hookMap.erase(it);
            ret = UnhookWindowsHookEx(hook);
        }
    }

    return ret;
}
```

동기화

지금까지 3 회의 강좌에서는 모두 SendMessage 계열의 함수를 사용해서 동기화를 수행했다. SendMessage 의 경우 부작용이 별로 없고 달리 신경쓰지 않아도 메시지 처리가 자동적으로 직렬화 되기 때문에 동기화에 신경쓸 일이 없었다. 하지만 WM_COPYDATA 를 통해서 빈번하게 많은 양의 데이터를 전송하는 것은 성능이 좋지 않다. 이번 강좌를 바탕으로 다음 시간에 제작하게 될 Spy++의 클론인 imSpy 에서는 이러한 단점을 개선하기 위해서 공유 메모리를 통해서 정보를 교환할 것이다.

이렇게 되면 공유 메모리는 하나이고 접근하는 곳은 여러 군데가 된다. imSpy 프로그램 자체는 해당 메모리를 읽기 위해서 접근해야 하고, 나머지 훅 프로시저는 해당 버퍼에 쓰기 위해서 접근해야 한다. 이 과정에서는 동기화를 필수적으로 해 주어야 한다. 그렇지 않을 경우에는 버퍼의 내용이 엉망이 될 것이다. 버퍼의 내용이 엉망이 되지 않도록 하기 위해서는 버퍼를 읽고, 쓰는 작업을 동기화 시켜주어야 한다. 이러한 작업에 적합한 객체로 커널 오브젝트의 하나인 이벤트 객체가 있다.

박스 2 배타적 접근과 직렬화

컴퓨터 용어를 접하다 보면 무슨 말인지 가끔 헷갈리는 것들이 나온다. 동기화에 관한 문서들을 읽을 때에 항상 등장하는 단어인 베타적 접근과 직렬화란 말도 그러한 것 중에 하나다. 보통 다음과 같이 사용된다.

- 자원에 대해서 베타적으로 접근하도록 해야 한다.
- 자원에 대한 접근을 직렬화 시켜야 한다.

결론적으로 말하면 이 두 가지 말은 동일한 의미다. 한 가지 자원에 동시에 두 군데 이상에서 접근하지 못하도록 하라는 의미다. 베타적 접근이란 말은 한 쓰레드가 자원에 접근하고 있을 때 다른 쓰레드는 접근하지 못하도록 해야 한다는 말이다. 직렬화란 말은 해당 자원에 접근하는 놈들을 동시에 접근하지 않고 순서대로 차례로 접근하게 하라는 의미다.

이벤트 객체

이벤트는 가장 원시적인 커널 오브젝트다. 하지만 원시적이라고 해서 쉽다고 이해해서는 안된다. 아래에 나오는 함수들을 읽어보고 잘 이해한 다음 한번씩 멀티 쓰레드 프로그램을 만들어서 실습해 보도록 하자. 오토 리셋과 매뉴얼 리셋 이벤트의 차이점은 꼭 확인해 보도록 하자.

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName
);
```

CreateEvent 함수는 이벤트를 생성하는 역할을 한다. lpEventAttributes 에는 생성될 이벤트 객체가 가질 보안 속성을 지정한다. bManualReset 에는 매뉴얼 이벤트인 경우 TRUE 를 오토 리셋 이벤트인 경우 FALSE 를 지정하면 된다. bInitialState 는 이벤트의 초기 상태가 시그널 상태인지 아닌지를 지정한다. lpName 에는 생성될 이벤트 객체가 가질 이름을 지정한다. 성공적으로 이벤트 객체가 생성된 경우 해당 객체의 핸들을 그렇지 않은 경우에는 NULL 을 리턴한다.

두 번째 인자가 매뉴얼 리셋과 오토 리셋 이벤트를 구분한다고 설명했다. 그렇다면 매뉴얼 리셋과 오토 리셋 이벤트의 차이점은 무엇일까? 보통의 경우 이 둘의 차이를 ResetEvent 의 호출 여부로 알고 있는 경우가 많다. 오토 리셋 이벤트는 ResetEvent 를 호출하지 않아도 되고, 매뉴얼 리셋 이벤트는 ResetEvent 를 호출해 주어야 한다는 것이다. 물론 앞의 설명도 맞긴 하지만 그것이 두 이벤트의 근본적인 차이는 아니다.

오토 리셋 이벤트의 경우 해당 이벤트를 대기중인 쓰레드 중에 임의의 한 쓰레드를 깨운다. 이 말은 대기중인 쓰레드가 세 개가 있다고 가정하면 그 중에 한 쓰레드를 택해서 깨운다는 말이다. 어떤 쓰레드가 스케쥴될지는 알 수 없다. 반면에 매뉴얼 리셋 이벤트의 경우에는 대기중인 쓰레드를 모두 깨운다. 앞의 예와 같이 세 개의 쓰레드가 대기중에 있다면 세

개의 쓰레드를 모두 스케쥴 한다는 의미다. 따라서 어떤 이벤트를 사용할지 결정해야 할 때에는 어떤 쓰레드가 깨워져야 하는지를 보고 판단해야 한다. 모두 다 깨워야 한다면 매뉴얼 리셋 이벤트를 하나만 깨워야 한다면 오토 리셋 이벤트를 사용하면 된다.

```
HANDLE OpenEvent(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);
```

OpenEvent 함수는 생성된 이벤트 객체를 열 때 사용한다. dwDesiredAccess 에는 <표 3>에 나온 것과 같은 값을 조합해서 사용하면 된다. bInheritHandle 은 CreateProcess 로 생성되는 프로세스에 열린 이벤트 객체를 상속시킬지를 나타낸다. 보통의 경우 FALSE 로 지정하면 된다. lpName 에는 열려고 하는 이벤트 객체의 이름을 지정한다.

표 3 dwDesiredAccess 에 지정될 수 있는 값 설명

값	의미
DELETE	객체를 삭제할 수 있음.
READ_CONTROL	객체의 보안 속성을 읽을 수 있음
SYNCHRONICE	해당 이벤트를 사용해 동기화 할 수 있음(WaitForSingleObject, WaitForMultipleObjects 등의 Wait 함수를 사용할 수 있음을 의미한다).
WRITE_DAC	보안 속성을 기록할 수 있음.
WRITE_OWNER	객체의 소유자를 변경할 수 있음.
EVENT_ALL_ACCESS	객체에 대한 모든 작업을 할 수 있음.
EVENT_MODIFY_STATE	이벤트 상태를 변경할 수 있음(SetEvent, PulseEvent, ResetEvent 를 사용할 수 있음을 의미한다).

```
BOOL SetEvent(HANDLE hEvent);
BOOL ResetEvent(HANDLE hEvent);
BOOL PulseEvent(HANDLE hEvent);
```

위 세 함수는 이벤트 객체의 상태를 조작하는 함수들이다. SetEvent 는 이벤트 객체를 시그널 상태로 만들고, ResetEvent 는 이벤트 객체를 넌시그널 상태로 만든다. PulseEvent 함수는 이벤트 객체를 시그널 상태로 만들어서 대기중인 쓰레드를 깨운후에 다시 해당 이벤트를 넌시그널 상태로 만든다.

동기화 프로토콜

앞 절에서 우리는 동기화의 필요성과 이벤트 객체에 대해서 배웠다. 이제 이벤트 객체를 사용해서 실제로 어떻게 동기화를 하는지 살펴 보도록 하자. 우리는 동기화를 위해서 두

개의 이벤트 객체를 사용할 것이다. 각각 쓰기, 읽기를 제어하는 용도의 이벤트 객체다. 이벤트 객체의 오브젝트 명과 역할은 <표 4>를 참고하자.

표 4 동기화에 사용될 이벤트 객체

커널 오브젝트명	의미
SSpy_Buffer_Ready	훅 프로시저가 버퍼에 데이터를 기록할 수 있다.
SSpy_Data_Ready	클라이언트 프로그램이 버퍼의 데이터를 읽어갈 수 있다.

<그림 1>에는 훅 프로시저와 클라이언트 프로그램이 어떻게 서로 동기화 하면서 정보를 주고 받는지가 나와있다. 왼쪽이 후킹 프로시저의 순서도이고, 오른쪽이 클라이언트 프로그램의 순서도다. 진행 순서는 오른쪽의 클라이언트 프로그램이 먼저 실행 된다. 클라이언트 프로그램의 버퍼 읽기 쓰레드는 적절한 형태로 두 개의 이벤트 객체를 생성한 다음 SSpy_Buffer_Ready 이벤트를 설정한다. 그리고 SSpy_Data_Ready 이벤트를 기다린다. 다음 훅 프로시저가 수행되면 SSpy_Buffer_Ready 이벤트가 설정되어 있기 때문에 통과해서 데이터를 버퍼에 쓰고 SSpy_Data_Ready 이벤트를 설정한다. 그러면 이번에는 클라이언트가 깨어나고 해당 데이터를 읽어가게 된다. 데이터를 모두 읽어간 후에 클라이언트는 다시 SSpy_Buffer_Ready 를 설정해서 다음 번 훅 프로시저가 버퍼에 데이터를 쓸 수 있도록 해 준다.

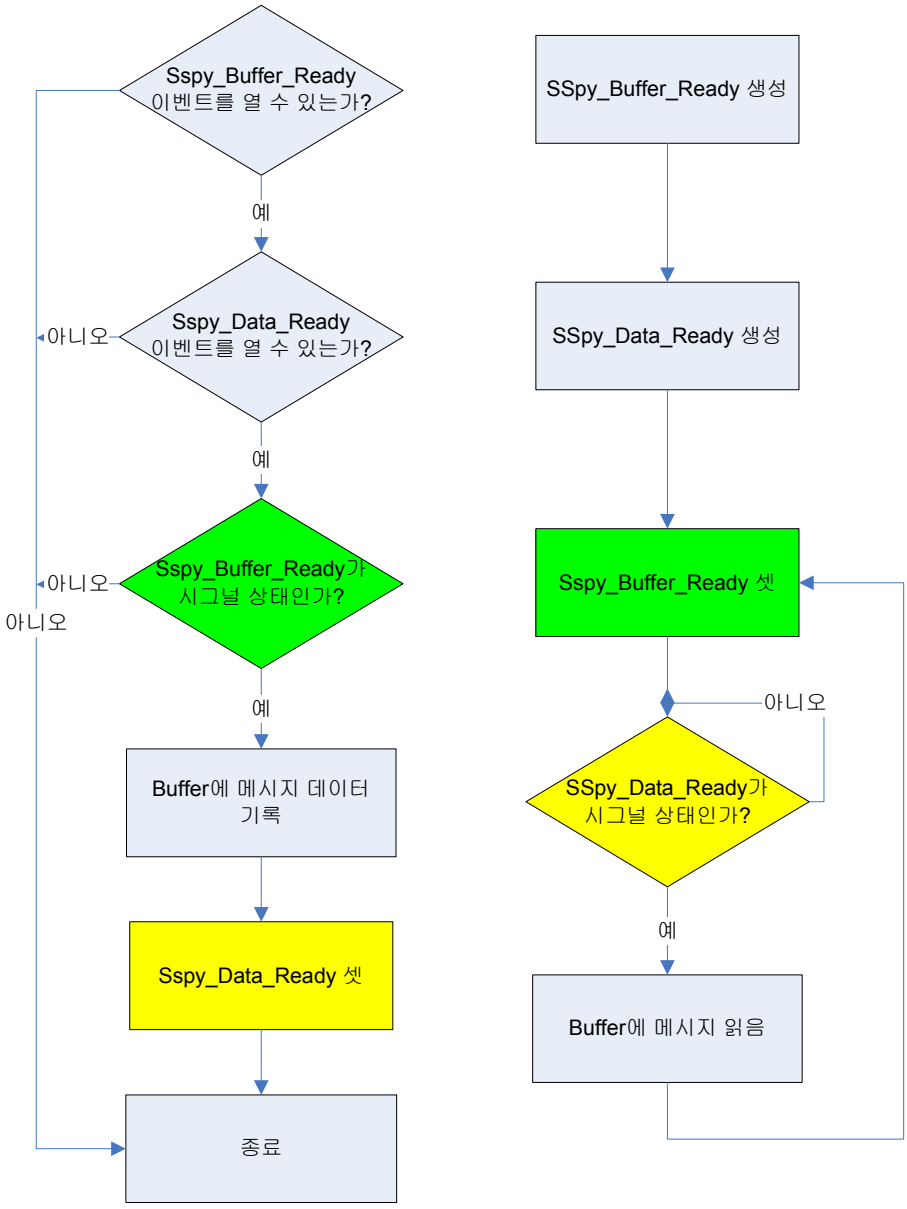


그림 1 훅 프로시저와 클라이언트의 동기화 흐름도

중복 실행 방지

이제까지 언급한 동기화 문제를 해결하는 방법에는 한 가지 제약 사항이 있다. 그것은 바로 클라이언트가 하나만 존재해야 한다는 것이다. 만약 클라이언트가 두 개 이상이라면 동기화 과정에서 문제가 생길 수 있다. SSpy_Buffer_Ready 이벤트가 순차적으로 설정되면서 두 개의 훅 프로시저가 동시에 데이터에 접근할 수 있으며, 자신이 설치하지 않은 훅에 대한 메시지도 받을 수 있다. 따라서 반드시 클라이언트는 중복으로 실행되는 것을 방지해서 그러한 일을 사전에 차단해야 한다.

Windows 에서 중복 실행을 방지하는 방법에는 여러 가지 방법이 있다. FindWindow 를 통해서 자신이 생성한 윈도우의 존재 유무를 검사하는 방법도 있고, 글로벌 커널 오브젝트를 생성해서 해당 오브젝트가 존재하는지를 검사하는 방법도 있다. 커널오브젝트를 사용하는 방법이 <리스트 4>에 나와있다.

리스트 4 커널 오브젝트를 사용해서 중복 실행을 방지하는 코드

```
class CSingleInstance
{
private:
   HANDLE  m_hGlbMutex;    // 전역 뮤텍스 핸들

public:
   CSingleInstance(LPCTSTR lpszMutexName="AppSingInstCheck");
   ~CSingleInstance();

   BOOL IsExist();
};


//
// 전역 뮤텍스를 생성해서 중복 실행 체크
//
// 파라미터
//     lpszMutexName   [in]    뮤텍스 이름
CSingleInstance::CSingleInstance(LPCTSTR lpszMutexName)
{
   m_hGlbMutex = CreateMutex(NULL, FALSE, lpMutexName);
   if(GetLastError() == ERROR_ALREADY_EXISTS)
   {
      CloseHandle(m_hGlbMutex);
      m_hGlbMutex = NULL;
   }
}

// 뮤텍스 핸들 해제
CSingleInstance::~CSingleInstance()
{
   if(m_hGlbMutex)
      CloseHandle(m_hGlbMutex);
}

//
// 이미 실행된 프로그램이 있는지 검사
//
// 파라미터 없음
//
// 리턴값
//     존재하는 경우 TRUE 를, 처음 실행인 경우 FALSE 를 리턴함
```

```
BOOL CSingleInstance::IsExist()
{
    return m_hGlbMutex == NULL;
}
```

기본 적인 원리는 간단하다. 네임드 커널 오브젝트의 경우 프로세스 경계에서 공유해서 사용할 수 있다는 점을 이용한 것이다. CreateMutex 함수로 이미 존재하는 커널 오브젝트를 생성하도록 한 경우에는 생성된 커널 오브젝트에 대한 핸들을 리턴한다(OpenMutex 가 호출되는 것과 동일하다). 이 경우에 GetLastError 를 호출해 보면 ERROR_ALREADY_EXSITS 가 설정되어 커널 오브젝트가 새로 생성된 것인지 아닌지를 판단할 수 있다. 즉, 애플리케이션 시작 부분에서 커널 오브젝트를 생성하고, 종료시에 커널 오브젝트를 해제 한다면 ERROR_ALREDY_EXISTS 가 설정된 경우는 이미 응용 프로그램이 실행된 경우고, 아니라면 처음 실행인 경우라고 생각할 수 있다.

프로그램에 위의 코드를 추가한 다음, 전역 변수로 CSingleInstance 를 하나 생성하도록 하자. 그런 후에 프로그램의 시작 부분에서 IsExist 를 통해서 검사해서 TRUE 인 경우에는 바로 종료하도록 하면 중복 실행이 되지 않는다.

도전과제

이번 시간에는 WH_CALLWNDPROC, WH_CALLWNDPROCRET 훅의 사용법과 함께 여러 개의 훅을 설치한 경우에 동기화 하는 방법에 대해서 자세히 살펴보았다. 이제까지 배운 지식을 바탕으로 다음 시간에는 Spy++의 클론인 imSpy 를 제작해 볼 것이다. 이번 시간까지의 지식을 바탕으로 자신만의 Spy++ 클론을 미리 제작해 보도록 하자.

참고자료

참고자료 1. Jeffrey Richter. <<Programming Applications for Microsoft Windows (4/E)>> Microsoft Press

참고자료 2. 김상형, <<Windows API 정복>> 가남사

참고자료 3. 김성우, <<해킹/파괴의 광학>> 와이미디어

참고자료 4. 매뉴얼 리셋 이벤트와 오토 리셋 이벤트의 차이점
http://www.gosu.net/GosuWeb/Article-detail.aspx?ArticleCode=617