다시 시작하는 윈도우 프로그래밍
프로세스 이야기 2

# 목차

목차 ... 1
소개 ... 1
연재 가이드 ... 1
필자소개 ... 1
필자 메모 ... 2
Introduction ... 2
프로세스 열거하기 ... 5
모듈 열거하기 ... 6
프로세스 이미지 경로 구하기 ... 8
Going Deep ... 9
도전 과제 ... 13
참고자료 ... 13

# 소개

PSAPI 라이브러리를 사용해서 시스템에 실행중인 프로세스를 열거하는 방법, 해당 프로세스에 로드된 모듈 정보를 구하는 방법에 대해서 살펴본다. 또한 윈도우 커널이 내부적으로 어떻게 프로세스 정보를 관리하는지에 대해서도 간략하게 소개한다.

# 연재 가이드

운영체제: Windows XP
개발도구: Visual Studio 2005
기초지식: C/C++ 문법
응용분야: 윈도우 응용 프로그램

# 필자소개

신영진 codewiz@gmail.com, http://www.jiniya.net
웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

## 필자 메모

미인대회의 심사 위원들은 자신이 예쁘다고 생각하는 참가자보다 다른 사람들이 아름답다고 생각할만한 참가자에게 높은 점수를 준다고 한다. 우리는 의식적으로든 무의식적으로든 남을 의식하고 있는 셈이다. 정도의 차이가 있을 뿐 대부분의 사람들은 남에게 보여지는 자신을 만들기 위해서 시간을 소비하며 살아간다.

패밀리 레스토랑에 밥을 먹으로 가면 꼭 음식을 먹기 전에 음식 사진을 찍는 사람들이 있다. 그 음식을 먹는 것을 즐기러 온 것인지 아니면, 다른 사람들에게 자신이 그 음식을 먹었다는 것을 알리기 위함인지 조금은 궁금해지는 장면이다. 여름 휴가 계획을 세울 때도 자신이 가고 싶은 곳보다는 다른 사람들이 근사하다고 생각할 법한 곳을 고민하는 사람들이 있다. 정작 자신은 그곳에서 아무 감흥을 느끼지 못하더라도 말이다.

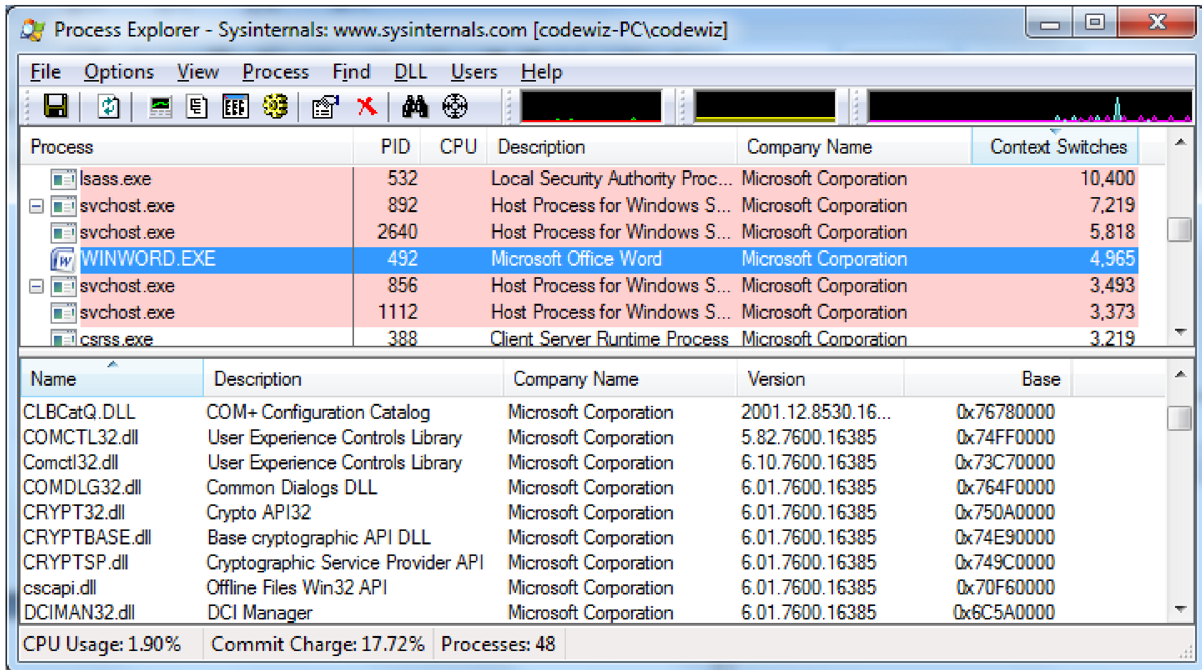
이제 2009년도 얼마 남지 않았다. 올해는 또 얼마나 많은 시간을 남에게 보여지는 자신을 만들기 위해서 낭비했는지를 돌아보게 된다. 남은 2009년의 시간은 나를 위한, 나만의 시간을 늘려보고 싶다는 생각을 해본다.

## Introduction

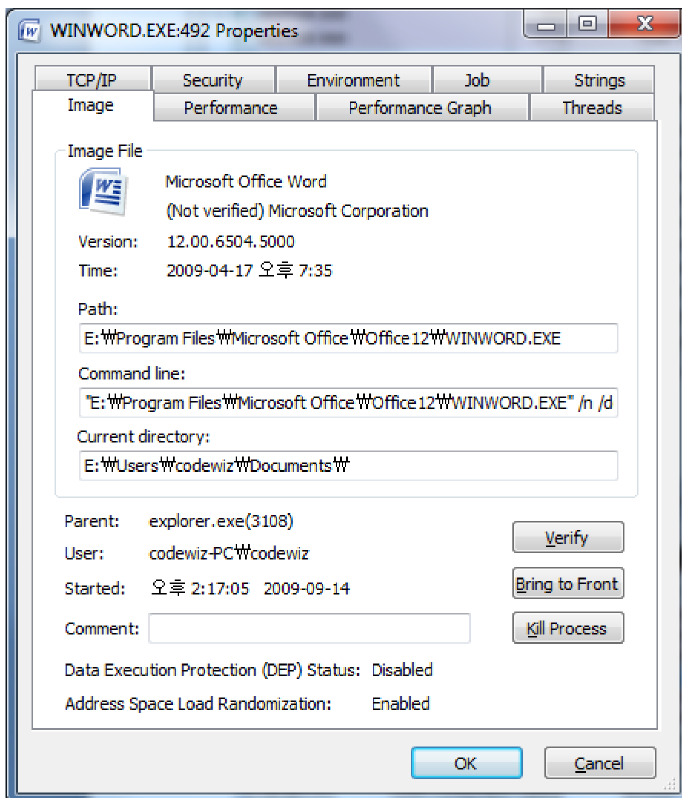
<화면 1>에는 Process Explorer의 실행 화면이다. 화면 상단에는 현재 실행되고 있는 프로세스 정보가 출력되고, 하단에는 선택된 winword.exe라는 프로세스에 로드된 모듈(DLL) 목록을 보여준다. <화면 2>는 선택된 프로세스의 실행 파일 경로, 현재 디렉터리, 실행 명령줄과 같은 정보를 보여주는 창이 나타나있다. 이번 시간에는 이러한 정보를 구하는 방법에 대해서 알아보도록 하자.

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/




**화면 1 시스템에 실행된 프로세스와 로드된 모듈 목록**




**화면 2 프로세스 정보**

윈도우에서 시스템에 실행된 프로세스 정보를 구하기 위해서는 다양한 방법을 사용할 수 있다. <


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

표 1>에는 이러한 여러 방법에 대한 장단점이 표시되어 있다. 각자 상황에 맞는 것을 사용하면 되겠다. ToolHelp를 사용한 예제는 Jeffrey Ritchter의 역작 Windows via C/C++에서 여러 차례 소개된 적이 있기 때문에 여기서는 PSAPI 라이브러리를 사용한 방법을 살펴보도록 한다.

**표 1 프로세스 정보를 구하는 방법에 따른 특징**

| 방법 | 장점 | 단점 |
| --- | --- | --- |
| ToolHelp 라이브러리 | Windows 9x 계열과 Windows NT 계열의 운영체제에서 모두 사용할 수 있다. | NT 계열 운영체제에 특화된 정보를 구하는 것이 제한된다. |
| PSAPI 라이브러리 | 기본 적인 정보뿐만 아니라 Windows NT 계열의 운영체제에 특화된 정보를 구할 수 있다. | Windows NT 계열의 운영체제에서만 사용할 수 있으며, psapi.dll이 있어야 한다. |
| NT 네이티브 API | Win32 API를 통해 노출되지 않은 다양한 정보를 구할 수 있다. | Windows NT 계열의 운영체제에서만 사용할 수 있다. 사용 방법이 어렵다. |
| 커널 구조체 직접 접근 | Win32 API, NT API를 통해 노출되지 않은 다양한 정보를 구할 수 있다. | 구현하기가 까다롭고, 운영체제 버전에 따른 차이점이 많다. |

**박스 1 PSAPI 이야기**

PSAPI는 NT의 초창기 때부터 존재했던 라이브러리다. 하지만 NT4까지 PSAPI는 기본적으로 운영체제 설치 시에 같이 설치되지 않았다. 이후 Windows 2000부터는 PSAPI가 기본적으로 배포되기 시작했다. Windows 2000의 PSAPI는 19개의 함수가, XP에는 24개의 함수가, Windows Vista, Windows 7에는 27개의 함수가 노출되어 있다. 운영체제가 업그레이드 되면서 PSAPI 또한 자연스럽게 기능이 추가된 것이다. 또한 Windows 7에서는 PSAPI의 기능이 모두 KENREL32.DLL로 이전되었다. Windows 7에 포함된 PSAPI.DLL 파일은 하위호환성을 위해서 KERNEL32.DLL에서 노출한 함수를 호출하는 형태로 존재한다.

과거 NT4에는 PSAPI가 기본적으로 배포되지 않았기 때문에 PSAPI를 사용하는 프로그램들은 별도로 해당 DLL 파일을 같이 배포해야 했었다. 그러한 관습 때문인지 요즘도 프로그램을 배포하면서 PSAPI.DLL을 같이 배포하는 프로그램이 더러 있다. 하지만 이는 자칫 매우 위험한 일이 될 수 있다. 특히 높은 버전의 운영체제에 해당하는 라이브러리를 배포하는 경우는 더더욱 그렇다. 예를 들어 XP와 함께 배포된 PSAPI를 배포하면서 해당 라이브러리에 포함된 GetProcessImageFileName 함수를 사용한다고 생각해보자. 이 경우에 해당 프로그램을 Windows 2000에서 구동하면 정상적으로 동작하지 않는다. 이유는 GetProcessImageFileName에서 호출하는 NtQueryInformationProcess 함수의 일부 인자를 Windows 2000은 지원하지 않기 때문이다. 따라서 같이 배포해야 한다면 가급적 낮은 버전을 사용하고, 힘들다면 시스템에

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

| 같이 배포된 PSAPI.DLL을 사용하는 것이 안전하다. |
|---|

## 프로세스 열거하기

PSAPI에서 프로세스 열거를 위해서 준비해둔 함수는 EnumProcesses다. 함수를 호출하면 시스템에 실행된 프로세스 목록을 구해서 배열에 저장해 준다. 함수 원형은 다음과 같다.

BOOL WINAPI EnumProcesses(DWORD *pProcessIds, DWORD cb, DWORD *pBytesReturned);

함수 인자의 의미는 그것들의 이름만큼이나 간단하다. 첫 번째 인자는 실행된 프로세스 목록을 저장할 배열의 포인터를 나타낸다. 두 번째 인자는 해당 배열의 바이트 단위의 크기다. 끝으로 마지막 인자는 입력으로 들어간 배열에 기록된 바이트 수가 저장된다.

기본적인 설명은 위의 내용이 끝이다. 하지만 여기에는 함정이 있다. 관심 있는 독자라면 아래 내용을 보기 전에 한번 어떻게 코드를 작성하면 시스템에 실행된 프로세스를 구할 수 있을지 생각해 보도록 하자.

고민을 해 봤다면 <리스트 1>의 코드와 한번 비교해 보도록 하자. 함정은 다름아닌 마지막 인자에 있다. 보통의 윈도우 API들은 위와 같은 함수를 디자인할 때 마지막 인자에 필요한 버퍼 크기가 들어가도록 설계한다. 즉, 버퍼가 작다면 마지막 인자로 넘어온 만큼 새롭게 할당해서 다시 함수를 호출하는 방식이다. 하지만 EnumProcesses는 그런 식으로 설계되지 않고, 마치 ReadFile처럼 얼마만큼을 기록했는지를 마지막 인자를 통해서 리턴 한다. 그렇다고 EnumProcesses를 다시 호출한다고 ReadFile처럼 이전에 읽은 다음 위치부터 프로세스 정보를 구할 수 있는 것도 아니다. 또한 버퍼가 부족하더라도 EnumProcesses 함수는 FALSE를 반환하지 않으며, GetLastError에 어떠한 설정도 하지 않는다. 따라서 모든 프로세스 정보를 구하기 위해서는 항상 넘어간 버퍼 사이즈가 기록된 내용보다 큰지를 체크해야 한다.

한 가지 더 지켜야 하는 것은 버퍼 크기는 항상 sizeof(DWORD)의 배수로 지정해야 한다는 점이다. 만약 그렇지 않다면 비교문 자체가 의미가 없어진다. 간단한 예로 버퍼 크기로 10을 넣고, 프로세스 세 개가 떠 있는 경우를 생각해보자. 그러면 EnumProcesses는 기록된 바이트 수를 8로 리턴할 것이다. 그러면 입력으로 넣은 버퍼 크기는 기록된 바이트 크기보다 더 크지만 실제로는 모든 프로세스 정보를 구하지 않은 상황이 되는 것이다.

**리스트 1 시스템에 실행된 프로세스 목록 출력하기**

```
#include <windows.h>
#include <stdio.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
```


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD BufferSize = 1024 * sizeof(DWORD);
    DWORD *Buffer = (DWORD *) HeapAlloc(GetProcessHeap(), 0, BufferSize);
    if(!Buffer)
        return 0;

    DWORD BufferSizeReturned = 0;

    for(;;)
    {
        if(!EnumProcesses(Buffer, BufferSize, &BufferSizeReturned))
        {
            HeapFree(GetProcessHeap(), 0, Buffer);
            return 0;
        }

        if(BufferSize > BufferSizeReturned)
            break;

        HeapFree(GetProcessHeap(), 0, Buffer);
        BufferSize += 1024 * sizeof(DWORD);
        Buffer = (DWORD *) HeapAlloc(GetProcessHeap(), 0, BufferSize);
        if(!Buffer)
            return 0;
    }

    DWORD ProcessCount = BufferSizeReturned / sizeof(DWORD);
    for(DWORD i=0; i<ProcessCount; ++i)
    {
        printf("%d₩n", Buffer[i]);
        PrintModuleList(Buffer[i]);
    }

    HeapFree(GetProcessHeap(), 0, Buffer);
    return 0;
}
```

## 모듈 열거하기

윈도우의 프로세스는 하나의 실행 파일로 구동되는 경우는 없다. 대부분 다른 DLL의 도움을 받아서 실행된다. 이렇게 한 프로세스의 주소 공간에 로드되어 있는 모듈의 정보를 구하는 방법에 대해서 알아보자. PSAPI는 이런 일에 사용하기 위해서 EnumProcessModules라는 함수를 준비해놓고 있다. 원형은 다음과 같다.

BOOL WINAPI EnumProcessModules(HANDLE hProcess, HMODULE *lphModule, DWORD cb, LPDWORD lpcbNeeded);

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

첫 번째 인자는 모듈 정보를 구할 프로세스 핸들이다. PROCESS_QUERY_INFORMATION, PROCESS_VM_READ 권한을 필요로 한다. lphModule에는 모듈 핸들을 리턴 받을 배열을 지정한다. cb에는 모듈 배열의 크기를 바이트 단위로 넘겨주면 된다. 끝으로 lpcbNeeded에는 모듈 정보를 모두 저장하기 위해서 필요한 크기가 넘어온다.

이 함수도 앞선 EnumProcesses와 마찬가지로 함정이 있다. 바로 리턴 값이다. 일반적인 함수들은 버퍼 크기가 부족한 경우라면 오류를 반환한다. 하지만 EnumProcessModules 함수는 EnumProcesses 함수와 마찬가지로 버퍼가 부족하더라도 기록할 수 있는 만큼만 기록하고 TRUE를 리턴한다. 물론 GetLastError에도 에러 값이 설정되지 않는다. 따라서 이 함수 또한 리턴 값이 아닌 필요한 바이트 크기를 사용해서 모듈 정보가 모두 구해졌는지를 별도로 체크해 주어야 한다. <리스트 2>에는 EnumProcessModules 함수를 사용해서 특정 프로세스에 로드된 모듈 목록을 출력하는 함수 예제가 나와 있으니 참고하도록 하자.

**리스트 2 특정 프로세스에 로드된 모듈 목록 출력하기**

```
BOOL PrintModuleList(DWORD pid)
{
    BOOL status = FALSE;

    HANDLE Process = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, pid);
    if(!Process)
        return FALSE;

    DWORD Size = sizeof(DWORD) * 1024;
    HMODULE *Modules = (HMODULE *) HeapAlloc(GetProcessHeap(), 0, Size);
    if(!Modules)
        goto $cleanup;

    DWORD SizeNeeded = 0;
    for(;;)
    {
        if(!EnumProcessModules(Process, Modules, Size, &SizeNeeded))
            goto $cleanup;

        if(Size > SizeNeeded)
            break;

        HeapFree(GetProcessHeap(), 0, Modules);
        Size = SizeNeeded;
        Modules = (HMODULE *) HeapAlloc(GetProcessHeap(), 0, Size);
        if(!Modules)
            goto $cleanup;
    }

    DWORD Count = SizeNeeded / sizeof(DWORD);
```

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
    TCHAR Name[MAX_PATH];
    for(DWORD i=0; i<Count; ++i)
    {
        GetModuleFileNameEx(Process, Modules[i], Name, ARRAYSIZE(Name));
        _tprintf(_T("    - %s₩n"), Name);
    }

    status = TRUE;

$cleanup:
    if(Modules)
        HeapFree(GetProcessHeap(), 0, Modules);

    if(Process)
        CloseHandle(Process);
    return status;
}
```

# 프로세스 이미지 경로 구하기

<리스트 3>에는 프로세스 이미지 경로를 구하는 함수 코드가 나와 있다. 소스를 살펴보면 알겠지만 PrintProcessImagePath가 하는 일은 첫 번째 모듈의 경로를 가져와서 출력하는 일이 전부다. 이 방법은 Windows NT부터 Windows 7까지 두루 사용할 수 있는 방법으로 MSDN의 예제도 이와 같은 방법을 사용하고 있다. 이 방법으로 프로세스의 이미지 경로를 구할 수 있는 이유는 다음 장에서 자세히 살펴보도록 하자.

**리스트 3 모듈 정보를 사용해서 실행 파일 경로 출력하기**

```
BOOL PrintProcessImagePath(DWORD pid)
{
    HANDLE Process = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, pid);
    if(!Process)
        return FALSE;

    HMODULE Module;
    DWORD SizeNeeded = 0;
    if(!EnumProcessModules(Process, &Module, sizeof(Module), &SizeNeeded))
    {
        CloseHandle(Process);
        return FALSE;
    }

    TCHAR Name[MAX_PATH];
    GetModuleFileNameEx(Process, Modules[i], Name, ARRAYSIZE(Name));
    _tprintf(_T("    - %s₩n"), Name);

    CloseHandle(Process);
    return TRUE;
```


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
}
```

Windows XP부터는 위와 같은 우회적인 방법 외에도 PSAPI에 추가된 GetProcessImageFileName 함수를 통해서 손쉽게 프로세스의 실행 파일 경로를 구할 수 있게 되었다. 함수 사용 방법은 특별한 것이 없기 때문에 <리스트 4>의 소스 코드를 참조하도록 하자.

**리스트 4 GetProcessImageFileName 함수를 사용한 실행 파일 경로 출력하기**

```
BOOL PrintProcessImagePath2(DWORD pid)
{
    HANDLE Process = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, pid);
    if(!Process)
        return FALSE;

    TCHAR Name[MAX_PATH];
    GetProcessImageFileName(Process, Name, ARRAYSIZE(Name));
    _tprintf(_T("    - %s₩n"), Name);

    CloseHandle(Process);
    return TRUE;
}
```

<리스트 3>과 <리스트 4>의 코드를 모두 실행해 본 독자라면 알겠지만 둘의 실행 결과는 조금 다르다. <리스트 3>의 출력 결과는 우리에게 친숙한 DOS 기반의 경로가 출력되지만 <리스트 4>의 결과는 NT 기반의 경로가 출력되기 때문이다. 이를 DOS 기반의 경로로 바꾸기 위해서는 QueryDosDevice와 같은 함수를 사용해야 한다.

## Going Deep

이제 좀 더 내부로 들어가 윈도우 커널이 어떻게 프로세스 정보를 관리하는지에 대해서 살펴보자. 윈도우 커널은 프로세스가 생성되면 EPROCESS라는 커널 객체를 생성한다. EPROCESS는 윈도우에서 프로세스 그 자체를 나타내는 커널 객체라고 생각하면 된다. 일반적인 경우에는 실행된 프로세스 개수와 EPROCESS의 개수는 동일하다. EPROCESS 객체는 커널 메모리에 생성되며, 따라서 이 객체는 일반적인 유저 모드 응용 프로그램에서는 참조할 수 없다. 이 객체를 직접 참조하기 위해서는 커널 모드에서 동작하는 드라이버를 작성해야 한다. <그림 1>에는 프로세스가 4개가 실행된 윈도우 시스템에서의 EPROCESS 구조를 보여주고 있다. EnumProcesses 함수는 이렇게 연결된 EPROCESS 구조체의 목록을 리턴하는 함수라고 생각하면 된다.


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

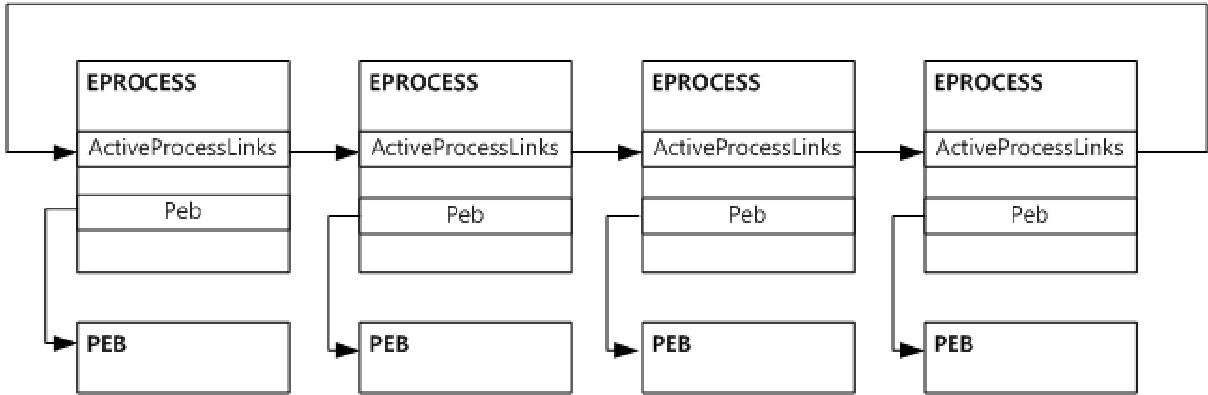



**그림 1 EPROCESS 연결 구조**

윈도우 커널은 프로세스가 생성될 때 EPROCESS라는 커널 구조체 외에도 <그림 1>에 나타난 것과 같이 PEB라는 구조체를 해당 프로세스의 주소 공간 내에 생성한다. 이 구조체는 EPROCESS와는 달리 해당 프로세스의 주소 공간에 생성되기 때문에 응용 프로그램에서 직접 참조가 가능하다. PEB는 프로세스 환경 블록(Process Environment Block)의 약자로 해당 프로세스 내에서만 참조될 필요가 있는 다양한 정보를 저장하고 있다.

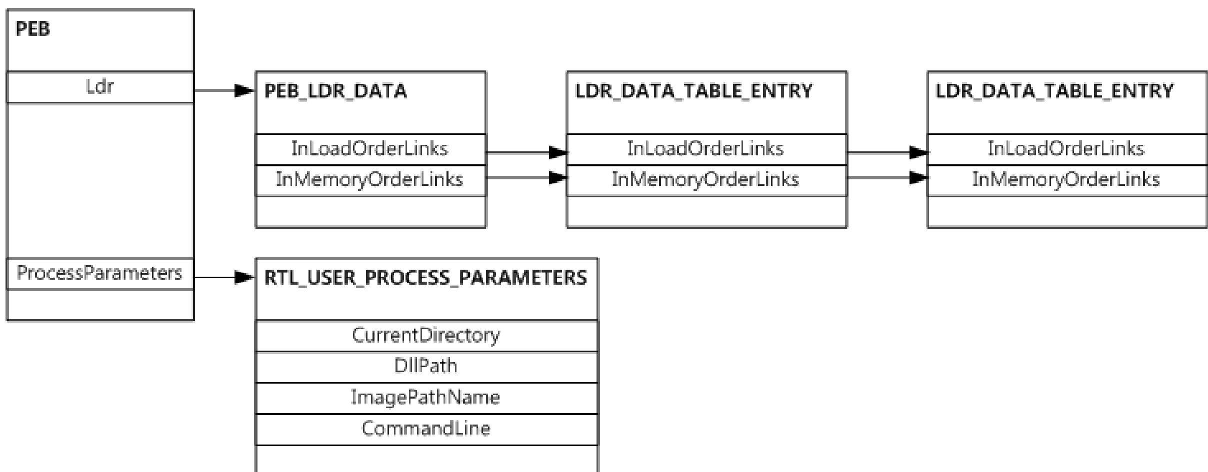



**그림 2 PEB 구조**

<그림 2>에는 PEB와 관련된 주요 구조체들의 관계를 보여준다. Ldr 필드는 현재 로드된 모듈 정보를 저장하고 있는 리스트의 헤더 필드인 InLoadOrderLinks와 InMemoryOrderLinks 필드를 저장하고 있는 PEB_LDR_DATA 구조체를 가리킨다. 해당 리스트에 연결된 아이템인 LDR_DATA_TABLE_ENTRY 구조체는 프로세스에 로드된 모듈 정보를 저장하고 있다. 해당 구조체 하나가 NTDLL.DLL과 같이 로드된 하나의 모듈에 대응한다고 보면 된다. InLoadOrderLinks와 InMemoryOrderLinks라는 리스트는 이름 때문에 서로 다른 순서로 연결된 링크로 보여진다. 하지만 실상은 둘 다 로드된 순서대로 연결된 동일한 리스트다.

PEB의 ProcessParameters 필드는 RTL_USER_PROCESS_PARAMETERS 구조체를 가리킨다. 해당 구


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

조체에는 해당 프로세스의 현재 디렉터리(CurrentDirectory), DLL 탐색 경로(DllPath), 실행 파일 경로(ImagePathName), 실행 명령줄(CommandLine)등의 정보가 포함되어 있다.

그렇다면 왜 EnumProcessModules의 첫 번째 값이 프로세스의 실행 이미지가 되는 것일까? 그 이유는 간단하다. EnumProcessModules 함수는 PEB 구조체의 InMemoryOrderLinks를 탐색해서 그 순서대로 결과를 리턴한다. 따라서 첫 번째 값은 항상 리스트의 첫 번째 값이 되고, 그것은 해당 프로세스에서 가장 먼저 로드된 실행파일 이미지가 되기 때문이다.

<리스트 5>에는 앞서 소개한 내용을 토대로 PEB 구조체의 Ldr에 포함된 리스트 링크를 조작하는 간단한 예제가 나와있다. 프로그램이 실행되면 첫 번째 모듈 링크를 다음 것으로 연결시켜서 실행 파일의 정보를 없앤다. 이 프로그램을 실행한 상태에서 <리스트 3>의 코드로 프로세스 정보를 구하는 방법을 시도해보면 정상적인 실행파일 경로가 출력되지 않는다. 우리가 리스트의 구조를 변경했기 때문에 앞서 설명한 가정이 틀리게 된 것이다. 반면에 GetProcessImageFileName 함수는 정상적으로 경로를 구해오는 것을 알 수 있다. 그 이유는 GetProcessImageFileName은 PEB에서 경로를 구하지 않고, EPROCESS 구조체의 특정 필드를 이용해서 직접 매핑된 파일 경로를 구하기 때문이다.

**리스트 5 ModuleHide 소스**

```
#include <windows.h>
#include <stdio.h>

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING;

typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;

typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PVOID Ldr;
    PVOID ProcessParameters;
    BYTE Reserved4[104];
    PVOID Reserved5[52];
    PVOID PostProcessInitRoutine;
    BYTE Reserved6[128];
    PVOID Reserved7[1];
    ULONG SessionId;
} PEB,  *PPEB;
```

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
typedef struct _PEB_LDR_DATA {
    BYTE Reserved1[8];
    PVOID Reserved2;
    LIST_ENTRY  InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA,  *PPEB_LDR_DATA;

typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID BaseAddress;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    SHORT LoadCount;
    SHORT TlsIndex;
    LIST_ENTRY HashTableEntry;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

typedef struct _RTL_USER_PROCESSPARAMETERS
{
    BYTE reserved[48];
    UNICODE_STRING  DllPath;
    UNICODE_STRING  ImagePathName;
    UNICODE_STRING  CommandLine;
    BYTE data[1];
} RTL_USER_PROCESSPARAMETERS, *PRTL_USER_PROCESSPARAMETERS;

__declspec(naked) PPEB GetPeb()
{
    __asm mov eax, fs:[0x30]
    __asm ret
}

int _tmain(int argc, _TCHAR* argv[])
{
    printf("%08X₩n", GetPeb());

    PPEB peb = GetPeb();
    PEB_LDR_DATA *pld = (PEB_LDR_DATA *) peb->Ldr;
    PLDR_DATA_TABLE_ENTRY first = (PLDR_DATA_TABLE_ENTRY) &pld->InLoadOrderModuleList;
    PLDR_DATA_TABLE_ENTRY        current        =        (PLDR_DATA_TABLE_ENTRY)        pld-
>InLoadOrderModuleList.Flink;
    PLDR_DATA_TABLE_ENTRY        next        =        (PLDR_DATA_TABLE_ENTRY)        current-
```

Copyright © 2011, 신영진 

이 문서는 Creative Commons 라이센스를 따릅니다.

http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
>InLoadOrderModuleList.Flink;

    pld->InLoadOrderModuleList.Flink = (PLIST_ENTRY) next;
    pld->InMemoryOrderModuleList.Flink = (PLIST_ENTRY) next;

    PRTL_USER_PROCESSPARAMETERS p = (PRTL_USER_PROCESSPARAMETERS) peb->ProcessParameters;
    printf("%S\n", p->ImagePathName.Buffer);

    getchar();
    return 0;
}
```

## 도전 과제

PSAPI에는 지면에 소개한 것 외에도 프로세스와 관련된 다양한 함수들이 있다. 해당 함수들이 어떠한 프로세스 정보를 구하는지 알아보고, 실제로 프로그램에 적용해서 출력해 보도록 하자. 그리고 지면에 소개하지는 않았지만 프로세스 정보를 구하는데는 NtQuerySystemInformation과 NtQueryInformationProcess라는 NT 네이티브 API가 사용된다. 지금은 부분적으로 문서화가 되었기 때문에, 해당 함수를 사용해서 프로세스 목록과 모듈 목록을 출력해 보도록 하자.

## 참고자료


찰스 페졸드의 Programming Windows, 5th Edition
Charles Petzold, 한빛미디어

Windows API 정복
김상형, 가남사

Windows Internals 5/e
Mark Russinovich, David A. SolomonDavid A. Solomon, Alex Ionescu, Microsoft Press

Windows via C/C++
Jeffrey M. Richter (Author), Christophe Nasarre, Microsoft Press



Copyright © 2011, 신영진



이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/