다시 시작하는 윈도우 프로그래밍
프로세스 이야기

목차

목차 1
소개 1
연재 가이드 1
필자소개 1
필자 메모 2
Introduction 3
프로세스의 시작 함수 4
프로세스 생성 하기 5
프로세스 상태 알아내기 7
프로세스 종료하기 9
현재 프로세스 정보 11
도전 과제 12
참고자료 12

소개

Windows라는 운영 체제에서 프로세스의 의미와 그것을 다루는 방법에 대해서 살펴본다. 프로세스를 생성하는 함수, 종료하는 함수, 현재 동작 중인 프로세스의 상태를 알아내는 함수에 대한 사용 방법을 소개한다. 더불어 각 함수에 대해서 개발자들이 잘못 알고 있는 상식에 대해서도 알아보도록 하자.

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
웹비아닷컴에서 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

얼마 전 국산 운영 체제 런칭 행사가 있었다. 해당 행사에 참석하진 못했지만 해당 행사의 실황 중계를 해주는 곳을 통해서 행사 관련 내용을 접할 수 있었다. 운영 체제 개발 프로젝트 책임자가 발표를 했는데, 그들이 개발한 커널이 마이크로 커널이라고 하면서 안정성이 높다는 점을 강조했다. 그러면서 Windows는 모놀리틱 커널이라 불안하다는 이야기를 했다. 이후 관련 신문 기사에는 두 운영 체제를 비교하면서 마이크로 커널, 매크로 커널이라는 용어를 쓰면서 해당 운영 체제가 안정성을 위주로 개발했다는 사실을 강조했다. 여담이지만 매크로 커널이란 말은 없다. 마이크로 커널의 반대말은 모놀리틱 커널이다.

그렇다면 그들이 좋다고 주장한 마이크로 커널과 모놀리틱 커널의 차이점은 무엇일까? 둘의 차이점이 <그림 1>에 나와 있다. 그림에 나타난 것처럼 모놀리틱 커널은 운영체제의 핵심 모듈들이 모두 단일 커널 모드에서 동작한다. 따라서 해당 모듈들에서 크래시가 발생할 경우에는 시스템이 중단되는 현상이 발생한다. 이를 보완하기 위해서 마이크로 커널은 커널의 진짜 핵심적인 역할 스케줄링, IPC등만 커널 모드에서 구현하고 나머지는 모두 유저 모드 프로세스로 구현해서 각 기능들을 컴포넌트간 통신을 통해서 이루어진다. 이런 구조적인 특징 때문에 주요 모듈들에서 크래시가 발생하더라도 그것이 실제 커널까지 전파되지 않는다는 장점을 가진다.

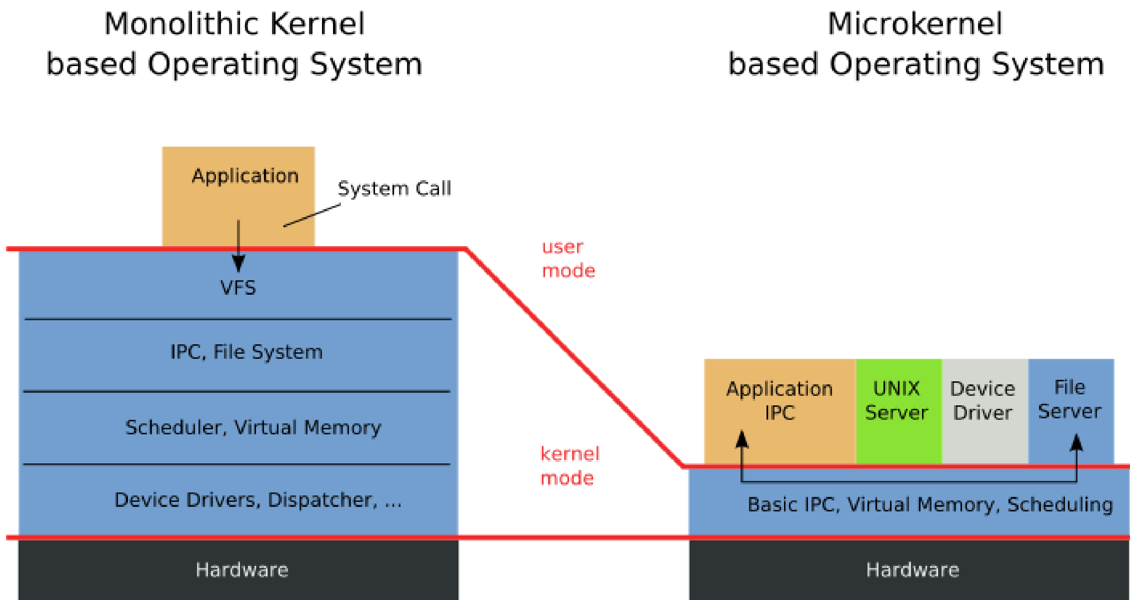


그림 1 모놀리틱 커널과 마이크로 커널의 구조

우선 그들이 주장 한대로 Windows는 모놀리틱 커널일까? 결론부터 말하면 '아니오'다. 현재 Windows 시스템의 근간을 이루고 있는 NT는 기본적으로 마이크로 커널을 모토로 출발했다. NT의 커널 설계자였던 데이비드 커틀러가 Mach 커널에서 많은 영감을 받았기 때문이다. 하지만 최종적으로 구현된 NT 커널은 조금은 중간적인 단계의 구조가 되고 말았다. 이유가 어쨌든 현재

Copyright © 2011, 신영진

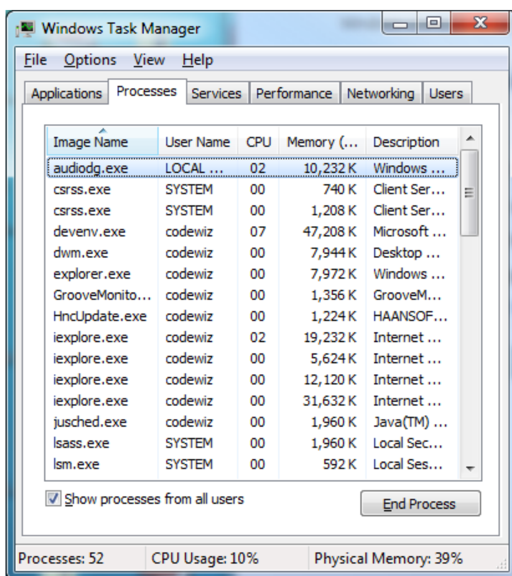
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

NT 커널은 두 가지 모두의 장점을 취하고 있는 하이브리드 커널로 분류된다. Windows의 환경 서브 시스템은 여전히 프로세스로 동작하기 때문이다.

두 번째로 그들이 주장한 것처럼 마이크로 커널이라서 안정적이고, 모놀리틱 커널이라고 해서 불안한 것일까? (물론 이 논쟁은 자칫 이념 논쟁이 될 수도 있다.) 저자는 그렇지 않다고 생각한다. 사실 커널의 설계 방식보다는 개별 컴포넌트들이 얼마나 안정적인가가 시스템의 안정성을 높이는 게 더 중요하다고 생각하기 때문이다. 마이크로 커널 설계자들이 주장하는 것처럼 파일 시스템을 유저 모드에서 구현했다고 생각해 보자. 그 파일 시스템에서 크래시가 발생했을 때 시스템이 중단되지 않고 진행할 수 있을까? 없다. 파일 시스템이 존재하는 위치에 상관없이 그 모듈은 너무 중요하기 때문에 결국 시스템은 중단될 수 밖에 없는 것이다. Windows에서 구현된 환경 서브 시스템은 유저 모드 프로세스로 동작한다. 그럼에도 불구하고 csrss.exe에서 크래시가 발생하면 여지없이 시스템은 중단된다. 설계 방식이 아니라 각 컴포넌트를 안정적으로 구현해 내는 것이 더 중요하다는 의미다.

Introduction

이번 시간엔 윈도우란 운영체제 속에서의 프로세스의 개념에 대해서 살펴보는 시간을 가져보도록 하자. 프로세스란 무엇일까? 프로세스를 만날 수 있는 가장 쉬운 곳은 작업 관리자다. 작업 관리자는 프로세스란 탭을 가지고 있다. 해당 탭을 클릭하면 <화면 1>에 나타난 것과 같이 현재 시스템에서 실행 중인 프로세스를 표시해 준다.



화면 1 작업 관리자에 나타난 프로세스 목록

그렇다면 작업 관리자의 프로세스 목록에 나타나는 것을 프로세스라고 생각하면 되는 것일까? 뭐 크게 나쁜 이유는 없다. 하지만 종종 작업 관리자에 표시가 되지 않는 프로세스도 있다는 점을

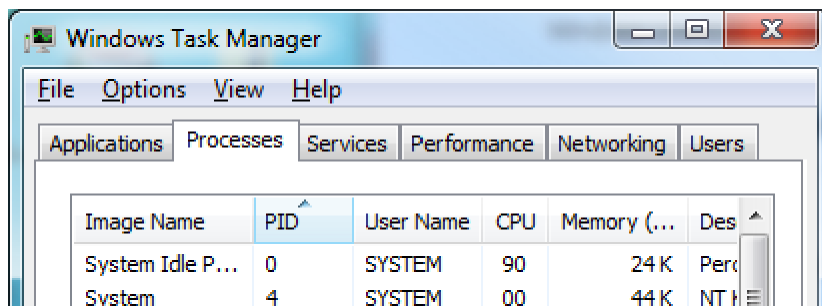
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

기억해 둘 필요는 있겠다.

Windows라는 운영 체제에서 프로세스가 가지는 가장 중요한 개념은 주소 공간이다. notepad.exe와 explorer.exe라는 두 프로세스를 구분하는 가장 큰 특징은 두 프로세스의 주소 공간이 다르다는 점을 나타낸다. 마찬가지로 300번 pid를 가진 notepad.exe와 400번 pid를 가진 notepad.exe의 차이도 주소 공간에 있다.

Windows는 기본적으로 스레드 기반 스케줄링 정책을 사용하기 때문에 프로세스는 프로그램의 실제 동작과 관련된 어떠한 정보도 포함하지 않는다. 실제로 작업을 수행하는 것은 결국 프로세스 내의 어떤 스레드가 되는 셈이다. 궁극적으로 Windows에서 프로세스는 이러한 스레드들이 동시에 동작할 수 있는 주소 공간을 제공하는 컨테이너라고 생각할 수 있다.

개발자들이 가장 많이 오해하는 부분은 실행 이미지와 프로세스의 관계다. 실행 이미지가 메모리에 올라간 것을 프로세스라고 생각하는 개발자들이 많이 있다. 하지만 이는 프로세스의 개념을 잘못 이해한 대표적인 예다. Windows 시스템의 프로세스에 따라서는 실행 이미지가 아예 존재하지 않는 것들도 있다. <화면 2>에는 이러한 대표적인 두 프로세스가 나와 있다. 이 두 프로세스는 시스템 시작 시에 생성되는 프로세스로 실제 프로세스 이미지는 없다. System Idle Process는 시스템의 작업이 없을 때 CPU 자원을 할당 받는 프로세스이고, System 프로세스는 커널 모드 드라이버에서 생성하는 시스템 스레드가 동작하는 주소 공간을 제공하는 역할을 한다.



화면 2 실행 이미지가 없는 대표적인 프로세스

프로세스의 시작 함수

WinMain에 가려서 실제 윈도우 프로세스의 시작 함수가 어떤 원형을 가져야 하는지에 대해서는 모르는 경우가 많다. Windows의 프로세스 시작 함수는 다음과 같이 정의 된다. 즉, 파라미터는 없고, 리턴 값은 DWORD인 stdcall 형태를 사용하는 함수로 작성하면 되는 셈이다.

```
typedef DWORD (WINAPI *PPROCESS_START_ROUTINE)(VOID);
```

CRT를 사용한다면 굳이 진입 함수를 직접 작성할 일은 없지만, CRT를 포함하지 않는 실행 파일을

Copyright © 2011, 신영진
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

만들어야 할 때에는 이러한 지식을 알고 있는 것이 도움이 된다.

프로세스 생성 하기

Windows는 프로세스를 생성하기 위해서 CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, ShellExecute, ShellExecuteEx, WinExec와 같이 다양한 함수를 준비해 놓고 있다. 이 중에서 개발자들이 주로 사용하는 함수는 CreateProcess, ShellExecute, ShellExecuteEx다. 여기서는 자주 사용되는 세 가지 함수에 대해서 간략하게만 살펴볼 것이다. 세 함수 모두 워낙 방대한 옵션을 가지고 있고, 그것들을 일일이 설명하려면 이 지면을 모두 사용해도 모자라기 때문이다.

어떤 경우에 무슨 함수를 사용하는 것이 올바른지에 대해서 생각해 보는 것으로 출발해 보자. 일단 프로세스를 정지된 상태로 생성할 필요가 없다면 CreateProcess를 사용하지 않는 것이 좋다. Windows XP 까지는 CreateProcess에 별다른 보안 정책이 없었지만 Windows Vista 부터는 관리자 권한이 없는 경우에는 CreateProcess의 호출이 실패하기 때문이다. 자식 프로세스가 필요한 경우라서 CreateProcess를 사용해야 한다면 반드시 해당 프로세스의 매니페스트 파일에 관리자 권한을 요구하는 프로그램이라는 사실을 기록해 두어야 한다. 단지 실행만 시키는 것을 원하는 경우라면 ShellExecute, ShellExecuteEx를 사용하면 된다. 두 함수의 주된 차이는 실행할 프로세스에 대한 제어 옵션을 가지느냐 마느냐로 귀결된다. 따라서 실행된 프로세스에 대한 추가적인 작업이나 설정이 필요하다면 ShellExecuteEx를, 그런 것이 필요 없다면 ShellExecute를 사용하도록 하자. ShellExecute 계열의 함수를 실행할 때 한 가지 알아 두어야 할 사실은 해당 함수들의 실제 실행 주체는 쉘(기본적으로 익스플로러 explorer.exe)이라는 사실이다. 따라서 실행할 수 있는 종류의 확장자는 쉘에 실행 파일로 등록된 것으로 제한된다. a.something과 같은 임의의 확장자를 가진 파일을 실행하려고 하면 실패한다는 뜻이다. something이라는 확장자가 만약 노트패드와 연결되어 있다면 쉘은 노트패드를 통해서 해당 파일을 여는 시도를 수행할 것이다.

이제 실제로 프로세스를 생성하는 코드를 살펴보자. <리스트 1>에는 CreateProcess를 사용하는 방법이 나와 있다. 거의 대부분의 경우에 아래 코드 패턴으로 사용하면 새로운 프로세스를 생성하는데 문제는 없다. 두 번째 인자가 실질적으로 수행할 명령이 들어가는 곳이다. 주의해야 할 점은 CreateProcess 함수 내부적으로 넘어간 값을 수정하기 때문에 반드시 수정 가능한 버퍼를 넘겨야 한다는 점이다. 단순히 문자열 리터럴을 전달하면 크래시가 발생한다. 이 경우에도 예외가 있는데 CreateaProcessA 함수는 내부적으로 넘어간 문자열을 다시 유니코드 문자열로 복사해서 CreateProcessInternalW를 호출한다. 따라서 CraeteProcessA의 경우에는 문자열 리터럴을 넘기더라도 크래시가 발생하지 않는 반면, CreateProcessW 함수는 크래시가 발생한다. 추가적으로 CreateProcess 사용에 주의해야 할 점은 명령어의 경로에 공백이 포함된 경우다. 이 경우에는 반드시 명령어 전체를 큰 따옴표(")로 묶어 주어야 한다.

프로세스 생성이 정상적으로 이루어지면 pi로 생성된 프로세스의 정보가 넘어온다.

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

PROCESS_INFORMATION 구조체에 포함된 핸들은 반드시 필요하지 않은 경우에 닫아 주어야 한다. 그렇지 않으면 프로세스를 생성할 때마다 두 개씩 핸들 릭이 발생한다.

리스트 1 CreateProcess를 사용한 프로세스 생성

```
PROCESS_INFORMATION pi;
STARTUPINFO si;

memset(&si, 0, sizeof(si));
si.cb = sizeof(si);

TCHAR CmdLine[] = _T("notepad.exe");
if(CreateProcess(NULL, CmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
{
    printf("프로세스 생성 성공₩n");
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

ShellExecute는 사용 방법을 설명할 것도 없을 정도로 간단하다. 그저 다음과 같이 사용하면 된다. 가장 주의해야 할 점은 성공한 경우의 리턴 값이다. 성공했다면 32보다 큰 값이 반환된다. ShellExecute 함수는 실행할 파일과 파리미터를 별도의 인자를 통해서 전달 받는다. 세 번째 인자는 실행할 파일 명이고, 네 번째 인자에는 전달할 파라미터를 넣어 주면 된다.

ShellExecute(NULL, _T("open"), _T("notepad.exe"), NULL, NULL, SW_SHOW);

끝으로 ShellExecuteEx의 사용법에 대해서 살펴보자. <리스트 2>에 관련 코드가 나와 있다. <리스트 2>의 코드는 앞선 ShellExecute의 호출과 동일한 일을 한다. ShellExecuteEx를 통해서 추가적인 제어를 할 수 있는 부분은 SHELLEXECUTEINFO의 fMask 옵션을 통해서다. fMask 옵션에 SEE_MASK_NOCLOSEPROCESS 옵션을 지정하면 ShellExecuteEx는 프로세스 생성 후, 생성된 프로세스 핸들을 해당 구조체의 hProcess에 전달해 준다. 해당 핸들을 사용해서 프로세스의 상태를 조사할 수 있다.

리스트 2 ShellExecuteEx를 사용한 프로세스 생성

```
SHELLEXECUTEINFO sei;

ZeroMemory(&sei, sizeof(sei));
sei.cbSize = sizeof(sei);
sei.lpFile = _T("notepad.exe");

ShellExecuteEx(&sei);
```

Copyright © 2011, 신영진

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

프로세스 상태 알아내기

이제 실행된 프로세스의 상태를 알아내는 방법에 대해서 살펴보도록 하자. 프로세스의 상태란 큰 의미에서 실행과 종료라는 두 가지 밖에는 없다. 물론 일시 정지된 프로세스도 있을 수는 있다. 하지만 엄밀히 말해서 Windows는 스레드 기반의 스케줄링을 하기 때문에 프로세스 일시 정지라는 말은 맞지 않다. 실제로 NtSuspendProcess와 같은 함수의 내부를 들여다 보면 해당 프로세스의 모든 스레드를 순차적으로 일시 정지 시키는 작업을 하는 것이 전부다. 이런 관점에서 말하면 앞서 언급한 두 가지 상태인 실행과 종료라는 말도 적당한 말은 아니다. 실행이라는 말은 프로세스의 주소 공간이 아직 유효한 상태를 의미하는 것이고, 종료라는 말은 프로세스의 주소 공간이 더 이상 유효하지 않은 상태를 의미한다.

프로세스의 종료를 알아내는 가장 쉬운 방법은 프로세스 핸들을 WaitForSingleObject에 전달하는 것이다. 프로세스 핸들은 실행 중인 상태일 때에는 비 시그널 상태이며, 종료가 되면 시그널 상태가 된다. 따라서 WaitForSingleObject에 프로세스 핸들을 전달해서 시그널 상태라면 종료된 것이고, 아니라면 계속 실행 중인 것으로 판단하면 된다. 즉, WaitForSingleObject(process, 0)을 해서 리턴 값이 WAIT_OBJECT_0이라면 해당 프로세스는 종료된 것이고, 다른 값이 반환된다면 해당 프로세스는 여전히 실행 중이라는 것으로 판단하면 된다는 것이다. 이를 조금 응용해서 WaitForSingleObject(process, INFINITE)를 하면 해당 프로세스가 종료될 때까지 기다리는 작업을 하게 된다.

Windows는 종료된 프로세스에 대해서 프로세스 종료 코드라는 것을 저장해 둔다. 종료 코드는 해당 프로세스의 시작 함수가 반환한 값으로, 주로 해당 프로세스가 정상적으로 수행이 완료되었는지를 판단하는 기준이 된다. 이러한 종료 코드를 구하는데 이용하는 함수가 GetExitCodeProcess다. GetExitCodeProcess(process, &ExitCode)와 같이 사용하면 해당 프로세스의 종료 코드가 ExitCode로 반환된다. 그렇다면 실행 중인 프로세스 핸들을 GetExitCodeProcess에 집어넣는다면 어떻게 될까? 그 때에는 STILL_ACTIVE라는 미리 정의된 상수 값이 ExieCode로 넘어온다.

GetExitCodeProcess 함수를 사용할 때에는 STILL_ACTIVE라는 값을 통해서 해당 프로세스의 종료 상태를 판단하지 않도록 주의해야 한다. 특정 프로세스의 리턴 값이 STILL_ACTIVE라는 상수 값과 동일한 경우에는 해당 프로세스가 종료된 경우에도 실행 중이라고 판단될 수 있기 때문이다. <리스트 3>에는 그러한 방식이 위험하다는 사실을 보여 주기 위해서 특별히 제작된 간단한 프로그램이 나와 있다.

리스트 3 GetExitCodeProcess를 통해서 프로세스 종료 상태를 판단하는 코드

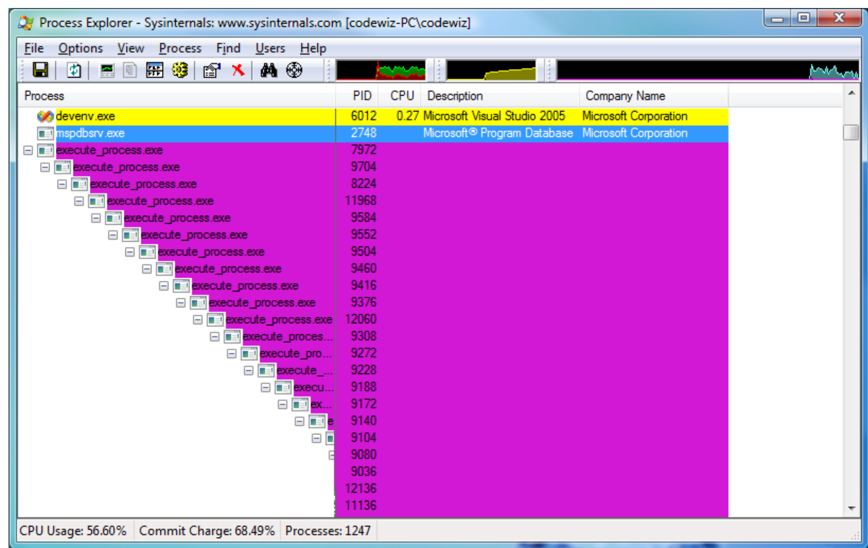
```
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    if(argc > 1)
```


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

```
        return STILL_ACTIVE;

    SHELLEXECUTEINFO sei;

    ZeroMemory(&sei, sizeof(sei));
    sei.cbSize = sizeof(sei);
    sei.fMask = SEE_MASK_NOCLOSEPROCESS;
    sei.lpVerb = _T("open");
    sei.lpParameters = _T("dummy");
    sei.lpFile = argv[0];
    ShellExecuteEx(&sei);

    WaitForSingleObject(sei.hProcess, INFINITE);

    DWORD ExitCode;
    if(GetExitCodeProcess(sei.hProcess, &ExitCode))
        if(ExitCode == STILL_ACTIVE)
            printf("아직 실행 중 입니다.₩n");

    return 0;
}
```

박스 1 황당한 버그 이야기

<리스트 3>의 코드를 작성하고 테스트 하던 중, 실수로 sei.lpParameters를 설정하는 부분을 빼먹었다. 프로그램 타이핑을 마치고 실행 시켜서 결과를 확인하려는데 정상적이라면 "아직 실행 중 입니다."메시지가 나와야 하는데 해당 메시지가 나오지 않는 것이었다.



화면 3 수많은 프로세스가 생성된 화면

직감적으로 잘못된 부분을 눈치챈 저자는 프로세스를 종료하려고 했다. 하지만 이미 너무 많은 프로세스가 생성된 후였다. 그 때 컴퓨터 상황이 <화면 3>에 나와 있다. 프로세스의 생성 구조

Copyright © 2011, 신영진

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

상 가장 아래 부분을 종료 시키면 순차적으로 모든 프로세스가 종료된다는 사실을 알 수 있다. 하지만 어떤 이유에서인지 프로세스의 가장 밑 부분은 생성 종료가 지속적으로 반복되고 있었다. 결국 로그 아웃을 할 수 밖에 없었다.

프로세스 상태를 구하는 두 함수를 소개했다. 하지만 정작 각 함수의 인자로 전달해야 하는 프로세스 핸들을 어떻게 구하는지에 대해서는 언급을 하지 않은 것 같다. 만약 자신이 생성한 프로세스라면 해당 프로세스 생성 함수의 리턴 정보를 통해서 전달 받을 수 있다는 사실을 앞서 배웠다. 자신이 생성하지 않은 프로세스의 정보를 얻기 위해서는 해당 프로세스의 프로세스 ID를 알아야 한다. 만약 해당 프로세스의 ID를 알아냈다면 OpenProcess 함수를 통해서 해당 프로세스의 핸들을 얻을 수 있다. 물론 이 경우에 해당 프로세스에 접근할 수 있는 권한을 가지고 있는 경우에만 정상적으로 핸들이 반환된다.

```
HANDLE process = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, ProcessId);
```

OpenProcess 함수는 일반적으로 위와 같은 형태로 사용한다. 첫 번째 인자는 원하는 프로세스 접근 권한을, 세 번째 인자에는 열고자하는 프로세스 핸들을 전달해 준다. 적절한 권한을 가지고 해당 프로세스에 접근할 수 있는 경우라면 process에 해당 프로세스의 핸들이 반환된다. 만약 그렇지 않다면 NULL이 반환된다.

프로세스 종료하기

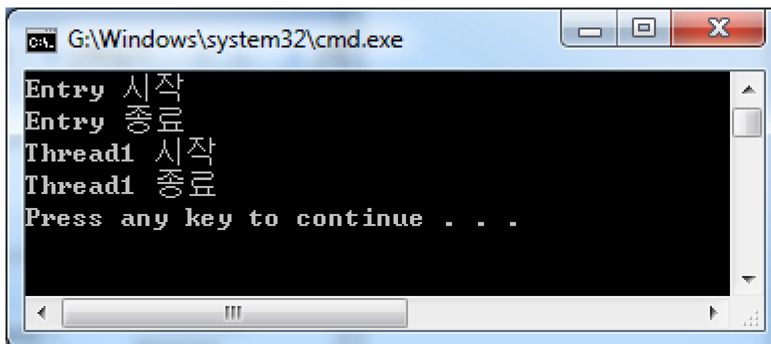
Windows는 해당 프로세스의 모든 스레드가 종료된 경우나 ExitProcess, TerminateProcess가 호출된 경우에 특정 프로세스를 종료 시킨다. 이러한 내용 중에서 보통 개발자들이 가장 많이 오해하고 있는 부분은 모든 스레드가 종료된 경우에 프로세스를 종료 시킨다는 점이다. 일반적으로 많은 Windows 개발자들은 주 스레드(프로세스 시작 시 처음 수행되는 스레드)가 종료되는 프로세스가 종료되는 것으로 알고 있다. 하지만 이는 잘못된 상식으로, <리스트 4>의 코드는 그렇지 않다는 사실을 보여 준다.

리스트 4 주 스레드가 먼저 종료되는 프로그램

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

#pragma comment(linker, " /ENTRY:Entry /NODEFAULTLIB")

void PrintString(LPCTSTR msg)
{
    WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE), msg, lstrlen(msg), NULL, NULL);
}

DWORD CALLBACK Thread1(PVOID param)
{
```

Copyright © 2011, 신영진
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
    PrintString(_T("Thread1 시작\n"));
    for(int i=0; i<3; ++i)
        Sleep(1000);

    PrintString(_T("Thread1 종료\n"));
    return 0;
}


UINT CALLBACK Entry()
{
    PrintString(_T("Entry 시작\n"));
    CreateThread(NULL, FALSE, Thread1, NULL, 0, NULL);
    PrintString(_T("Entry 종료\n"));
    return 0;
}
```



화면 4 주 스레드가 먼저 종료되는 프로그램 실행 결과

<리스트 4>의 프로그램을 컴파일해서 실행하면 <화면 4>과 같은 결과가 출력된다. 일반적으로 많은 개발자가 알고 있는 상식대로라면 Entry 종료가 호출된 직후에 프로세스는 종료되어야 할 것이다. 하지만 결과는 Thread1 종료가 출력된 직후에 프로세스가 종료되는 것으로, Windows가 모든 스레드가 종료된 직후에 프로세스를 종료시킨다는 사실을 말해 준다.

그렇다면 일반적인 Windows 프로그램은 왜 WinMain이 리턴되면 자동적으로 종료되었을까? 그 내용은 CRT(C Runtime Library)라는 부분과 연관된다. 프로세스가 시작되면 실제 진입 함수는 CRT 내의 어떤 함수가 된다. 그 함수는 CRT 초기화 작업을 끝내고 WinMain을 호출한다. WinMain이 리턴되면 해당 함수는 초기화 작업 동안에 생성한 리소스를 해제하고 최종적으로 ExitProcess를 호출한다. 그렇기 때문에 다른 스레드가 동작 중이라도 종료되는 것이다.

모든 스레드가 종료되는 일이 아닌 보통의 경우에 Windows 프로그램은 ExitProcess를 호출 해서 프로세스를 종료시킨다. ExitProcess는 호출한 프로세스 자신을 종료시키는 함수다. 파라미터로는 프로세스 종료 코드가 들어간다. ExitProcess(255)와 같이 호출하면 프로세스를 종료 시키고, 종료 코드를 255로 설정한다.

Copyright © 2011, 신영진

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

ExitProcess와 비슷한 함수로 TerminateProcess라는 함수가 있다. TerminateProcess라는 함수는 ExitProcess와 달리 두 가지 주요한 특징을 가지고 있다. 하나는 다른 프로세스를 종료할 수 있다는 점이고, 다른 하나는 종료에 관한 어떠한 통보도 없이 단숨에 프로세스를 종료 시킨다는 점이다. TerminateProcess(process, 255)와 같이 호출하면 process 핸들이 가리키는 프로세스가 종료되고, 해당 프로세스의 종료 코드는 255로 설정된다.

ExitProcess와 TerminateProcess의 차이에 관해서 많은 개발자들이 오해하고 있는 부분은 앞서 언급한 종료 통보도 없이 단숨에 종료 시킨다는 TerminateProcess의 특징이다. 이러한 점 때문에 많은 Windows 개발 서적은 TerminateProcess 호출을 자제하도록 지시한다. 그리고 그 영향으로 많은 개발자들은 TerminateProcess 함수 호출은 정말 못쓰는 함수처럼 여긴다.

하지만 이는 진실은 아니다. Windows 시스템이 근간을 두고 있는 NT 네이티브 시스템에는 NtTerminateProcess라는 함수는 있으나, NtExitProcess라는 함수는 없다. 즉, ExitProcess든, TerminateProcess든 내부적으로는 NtTerminateProcess를 통해서 종료가 이루어진다는 사실을 말해 준다. 더욱이 프로세스의 주소 공간을 파괴한다던가, 각종 열린 핸들을 닫는 작업은 NT 네이티브 시스템에서 이루어진다는 점을 생각한다면 TerminateProcess를 호출하던지, ExitProcess를 호출하던지 리소스 반납이 안 된다는 주장은 조금 억지스럽다. 어떤 함수를 호출하던지 모든 커널 객체와 프로세스의 메모리 공간은 적절한 방법을 통해서 파괴된다.

그렇다면 ExitProcess가 해 준다는 종료 통보라는 것은 과연 무엇을 의미할까? ExitProcess는 내부적으로 호출이 되면 일단 NtTerminateProcess를 호출해서 프로세스 내의 모든 스레드를 종료 시킨다. 그리고 해당 프로세스 주소 공간에 로드된 DLL들에 대해서 각 DLL의 DllMain을 DLL_PROCESS_DETACH를 인자로 전달해서 순차적으로 호출한다. 이 작업이 모두 완료 되면 최종적으로 csrss.exe에 프로세스 종료를 통보하고 NtTerminateProcess(GetCurrentProcess(), ExitCode)를 호출해서 최종적으로 프로세스를 종료시킨다.

즉, 개발자 입장에서 느끼게 되는 차이는 DllMain 함수가 DLL_PROCESS_DETACH를 통한 호출을 받을 수 있느냐 없느냐 하는 차이가 있는 것이다. 그것 외에는 TerminateProcess와 ExitProcess의 차이는 없는 셈이다. 따라서 많은 개발자들이 생각하는 것처럼 TerminateProcess를 호출하면 커널 객체가 정상적으로 반납되지 않는다거나 할당한 메모리가 영원히 남아 있게 된다거나 하는 걱정은 할 필요가 없는 기우인 셈이다.

현재 프로세스 정보

Windows는 현재 코드가 동작하고 있는 프로세스 정보를 빠르게 구하기 위해서 GetCurrentProcess, GetCurrentProcessId라는 두 가지 함수를 준비해 놓고 있다. 이 두 함수는 각각 현재 실행되는 프로세스 핸들과 현재 실행되는 프로세스 ID를 반환한다. 한 가지 주의해야 할

Copyright © 2011, 신영진

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

사실은 GetCurrentProcess가 반환하는 핸들은 의사 핸들로 항상 0xffffffff을 가진다. 따라서 실제 핸들을 구하기 위해서는 OpenProcess에 현재 프로세스 ID를 전달해서 직접 구해야 한다.

도전 과제

이번 시간에 프로세스를 제어하는 많은 함수에 대해서 살펴보았다. 그 중에 특히 CreateProcess, ShellExecuteEx, OpenProcess 등은 옵션이 많고 사용 방법이 복잡하다. 해당 함수의 완전한 사용법을 MSDN에서 찾아보고 각 옵션들이 어떠한 의미를 가지는지에 대해서 알아보도록 하자. 다음 시간에는 프로세스의 생성과 종료, 실행된 프로세스 정보를 얻는 방법 등에 대한 보다 고급 주제에 대해서 살펴볼 것이다.

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/