라이브러리

- Code

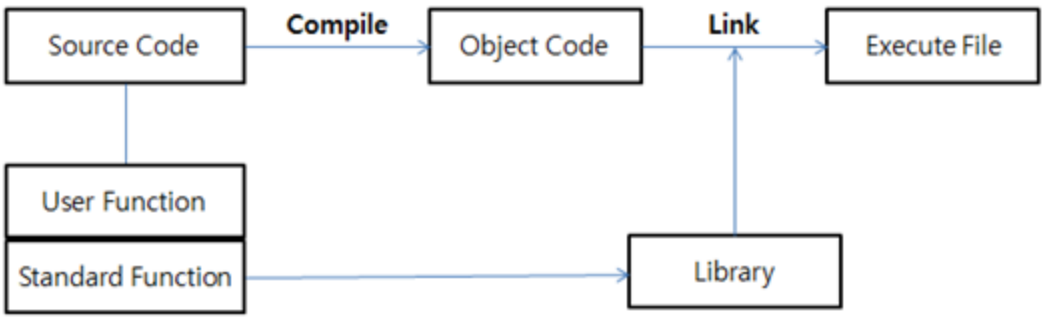
[확장자별 라이브러리 구분]

- .a : 리눅스 / 정적 라이브러리
- .so : 리눅스 / 동적 라이브러리
- .lib : 윈도우 / 정적 라이브러리
- .dll : 윈도우 / 동적 라이브러리

---

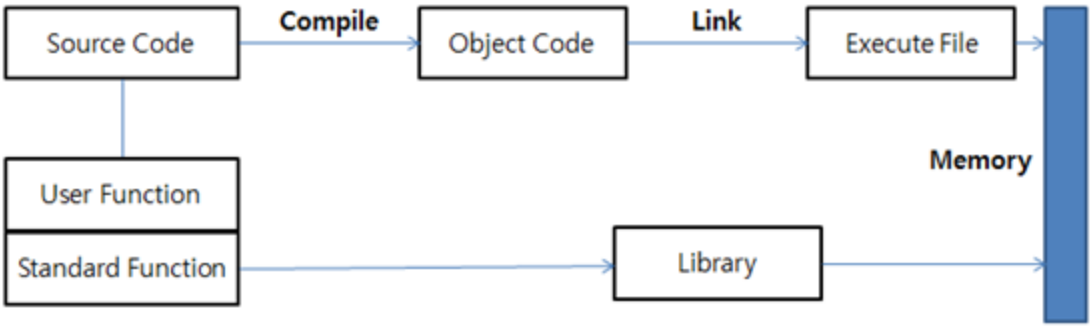
1. Static library
    a. 정의
        i. 특정 기능의 라이브러리를 링크단계에서 라이브러리(*.lib)를 실행 바이너리에 포함
    b. 장점
        i. 라이브러리 동작 코드가 실행 바이너리 속에 포함되어 독립적(실행 바이너리만으로..)으로 라이브러리 함수를 사용할 수 있다.
    c. 단점
        i. 실행 바이너리 사이즈가 커진다.
        ii. 공통 라이브러리가 여러 실행 바이너리에 포함되어 실행될 경우 메인 메모리의 공간 활용 효율이 떨어진다.
            1. multiple-caller program이 존재하는 경우 그 다지 바람직하지 않다.
    d. 형식

```
#pragma comment(lib, "static lib Dir\name.lib")
```



2. Dynamic (linking) library : DLL
    a. 정의

i. 동적으로 라이브러리를 링크하여 해당 라이브러리 함수를 사용한다.
ii. 필요시 사용할 수 있도록 최소한의 정보만 포함하여 링크하거나, 독립적으로 DLL을 로드 / 사용 / 해제 할 수 있다.



a. implicit linking
   i. 라이브러리를 컴파일 => *.lib, *.dll 파일 생성
   ii. *.lib는 static lib.lib와 전혀 다르다.
      1. static lib는 라이브러리 전체 코드를 포함하는 바이너리
      2. dynamic lib는 dll에서 제공하고자 하는 함수 정보(함수명)을 가지는 정보 파일
   iii. DLL의 *.lib를 이용하여 링킹하는 것을 암시적 링킹(implicit linking)이라고 한다.
   iv. 실행 바이너리를 링크 단계에서 실행 바이너리의 *.obj 파일들과 DLL의 *.lib 파일을 함께 링크하여 이 정보를 토대로 runtime에 dll의 함수 코드를 참조하게 되는 것이다.
      1. *.lib => 링크시 필요
      2. *.dll => 실행시 필요
a. explicit linking
   i. 명시적 링킹에서는 *.lib파일이 필요하지 않다.
      1. 실행 바이너리 링크 단계에서 DLL의 함수 정보가 필요하지 않기 때문이다.
   ii. 사용되는 3가지 함수와 역활
      1. LoadLibrary : 필요한 DLL을 프로세스 가상 메모리에 맵핑
      2. GetProcAddress : DLL 함수의 포인터를 획득
      3. FreeLibrary : 프로세스 가상 메모리에서 DLL을 반환한다.
   iii. 프로세스는 내부적으로 DLL의 레퍼런스 카운트를 계산
      1. LoadLibrary 호출시 DLL의 (프로세스) 레퍼런스 카운트는 1씩 증가.
      2. FreeLibrary 호출시 레퍼런스 카운트가 1씩 감소
      3. 래퍼런스 카운트가 0이 될 때 해당 DLL은 프로세스 가상 메모리에서 해제된다.
      4. 주의점

        a. 물리 메모리가 아닌 가상 메모리에서의 반환(해제)
        b. 레퍼런스 카운트는 각 프로세스 내부 호출 회수
            i. 전체 프로세스 간의 호출 회수가 아니다.
    5. 이유
        a. 프로그램 실행 중 DLL을 가상 메모리에 할당, 해제할 수 있도록 하기 위함
iv. 장점
    1. DLL이 필요한 시점에 로딩, 불필요한 시점에 반환 => 메모리 절약
    2. 프로그램 실행 중 DLL 교체 및 선택 가능
    3. 암시적 링킹은 프로그램 실행 전 필요한 모든 DLL을 메모리 로딩 = > 실행까지 시간 걸림
        a. 명시적 링킹은 필요한 시점에 하나씩 DLL 로딩 가능 => DLL 로딩 시간 분산 가능

[동적 라이브러리]

장점

- 메모리를 절약하고 스와핑을 줄인다.
- 프로그램이 **한번 메모리**에 **올려진 것**을 **공유**하므로 메모리 사용 공간이 정적 라이브러리에 비해 적다.
- 여러 프로세스가 메모리에 있는 하나의 DLL 복사본을 공유하여 **하나의 DLL**을 **동시**에 **사용**할 수 있다.
- 반면, 정적 연결 라이브러리를 사용하여 빌드된 응용 프로그램의 경우 Windows는 각 응용 프로그램에 대해 하나의 라이브러리 코드 복사본을 메모리에 로드해야 한다.
- 동적은 메모리에 올라와있는 DLL을 참조하여 사용하기 때문에 1개만 있으면 되지만
- 정적은 빌드 시 라이브러리를 함께 컴파일이 이뤄져야하기 때문에 각 프로그램마다 라이브러리 코드를 갖고있는다.
- DLL을 보다 **쉽게 업그레이드** 할 수 있다.
- DLL의 **함수**가 **변경**되어도 이 함수의 인수 및 반환 값이 변경되지 않았으면 그 함수를 사용하는 응용 프로그램은 **다시 컴파일**하거나 **링크**할 필요가 없다.
- 반면, **정적**으로 링크되는 개체 코드의 경우에는 함수가 변경되면 **응용 프로그램**을 **다시 링크**시켜야 한다.
- 출시 후 지원이 가능하다.
- 예를 들어, 응용 프로그램을 출시할 때 사용할 수 없었던 디스플레이 기능을 지원하도록 디스플레이 드라이버 DLL을 수정할 수 있다.
- **언어 형식**이 **다른 여러 프로그램**을 **지원**한다.

- 서로 다른 프로그래밍 언어로 작성된 프로그램인 경우에도 **함수의 호출 규칙**을 따르기만 하면 **여러 프로그램**에서 **동일한 DLL 함수**를 호출 할 수 있다.
- 이 경우 각 프로그램과 DLL 함수는 여러 가지 면(스택에 해당 함수의 인수가 들어가는 순서, 스택을 정리하는 것이 함수인지 응용 프로그램인지의 여부 및 인수가 레지스터에 전달되는지의 여부)에서 호환 될 수 있어야 한다.
- **프로그램 변경시 변경된 부분**의 **공유 라이브러리**만 **재배포**하면 되므로 유지보수가 쉽다.

단점

- **외부 의존도**가 생기기 때문에 **이식성**이 어렵다.
- **공유 라이브러리**를 **메모리**에 올리려면 찾고 올리는데 시간이 걸리므로 성능저하가 생긴다.

[정적 라이브러리]

장점

- 시스템 환경이 변해도 애플리케이션에 아무런 영향이 없고, **완성된 애플리케이션**을 **안정적**으로 사용할 수 있다.
- **컴파일시** 필요한 라이브러리를 프로그램 내에 적재하기 때문에 **이식성**이 좋다.
- **런타임시** 외부를 참조할 필요가 없기 때문에 **속도**에서 장점이 있다.

단점

- 같은 코드를 가진 여러 프로그램을 실행할 경우 코드가 중복이 되니 그만큼 메모리를 낭비하게 된다.
- **라이브러리 변경**이 필요할 시, 변경된 라이브러리만 재배포하면 안되고 **프로그램**을 다시 **재배포** 해야 한다.

DLL 심화

1. *.exe와 *.dll의 관계

1) exe : import
2) dll : Export

2. DLL Binding

- EXE 파일은 사용하고자 하는 DLL의 함수를 메모리에 같은 메모리상에 올리게 되는데 이때 IAT에는 실제 사용하고자 함수들의 주소가 오게 된다.

(다시 말해, 파일에서 IAT는 실제 함수의 주소를 가리키고 있지 않다. 왜냐하면 사용하고자 하는 함수의 주소를 아직 알 수 없기 때문이다.)
- 메모리에 올라오면서 PE 로더는 IAT에 사용하고자 하는 함수의 실제 주소를 올려주므로 우리는 아무런 의심 없이 사용할 수 있다.
(이러한 작업은 프로그램의 초기화 시간을 지연시키므로 MS는 이러한 제약을 피할 수 있도록 하나의 기능을 제공한다. 바로 IAT에 함수의 주소를 기록하는 작업을 미리 수행하여 로딩 시의 속도 향상을 도모하도록 한다.) => 이 과정을 바로 DLL 바인딩이라고 하며, 바인드 된 실행 파일의 IAT는 실제 함수의 주소를 가리키고 있게 된다.

3. DLL Relocation

하나의 EXE 파일은 여러 라이브러리를 필요로 하는 경우가 일반적이기 때문에, 여러 DLL을 메모리에 올리고자 한다.
이 경우 DLL들의 ImageBase가 중첩된다면 하나의 메모리 주소에 여러 DLL이 존재할 수 없으므로 사용할 수 없게 된다.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
WORD Magic;
BYTE MajorLinkerVersion;
BYTE MinorLinkerVersion;
….
DWORD AddressOfEntryPoint;
….
DWORD ImageBase;
….
```

다행히 DLL 재배치를 통해 원하는 ImageBase에 이미 다른 DLL이 올라와 있다면, 다른 주소에 맵핑될 수가 있다.
하지만 이러한 재배치 작업이 일어나면 PE 로더는 부차적인 작업을 수행해야 한다. DLL의 주소를 바꾸어 올리는 것뿐만 아니라, 해당 DLL의 Code Section의 일부 내용을 수정해야만 한다.
이에 대해 아래의 표를 보자.
아래의 표는 Relocation Section의 내용으로 하단 두 줄에 RVA가 있는 것을 확인할 수 있다.
일반적으로 이 주소가 가리키는 부분은 0x????????과 같은 4 Bytes의 주소를 나타내는 것으로 PE 구조에 기록되어 있는 ImageBase에 맞게 주소가 설정되어 있다.

하지만 ImageBase와 다른 곳에 로드되면 이 주소들은 ImageBase에 로드된 다른 DLL을 가리키게 되는 문제가 발생하므로 값을 수정해주어야 한다.

```
[+] Relocation Section --------------------
Base Address : 0x1000
Size of Block : 0x1069 (Num : 0x830)
Type Value : 0x5708 --- RVA : 0x1708 (Offset : 0xB08)
Type Value : 0x106C --- RVA : 0x106C (Offset : 0x46C)
```

다행히 이러한 주소 값의 수정을 사용자가 직접 하나하나 하는 것이 아니라 PE 로더가 재배치 섹션을 확인하여 알아서 수정해준다.
하지만 여기서 몇 가지 문제점이 존재하게 된다.
우선 재배치 정보가 가리키고 있는 값들은 대개 어떤 주소에 관한 값으로, 이러한 값들이 대개 코드 섹션에 위치하고 있다는 것이다.
따라서 이 값을 PE 로더가 수정하기 위해선 해당 섹션에 Write 속성을 추가한 뒤 수정을 하고, 수정을 마치면 다시 원래의 속성으로 되돌려야 한다는 것이다.
이에 더해 위 예에서는 2개의 값만 나타냈지만, 실제로는 더 많은 경우가 많기 때문에 PE 로더는 그 많은 주소의 값들을 직접 찾아 수정해주어야 한다. 만약 하나의 EXE 파일에 여러 DLL에 대하여 이러한 작업을 수행해야 한다면, 프로그램을 실행하기 위한 초기화 시간이 길어질 수 있다.

4. DLL Delay Loading

상기의 이유들로 초기화 시간이 길어질 수 있다는 것에 대하여 알 수 있었다.
사실 하드웨어의 성능이 상향된 요즘은 별 상관이 없지만, 윈도우는 이러한 초기화 시간을 줄이기 위한 또 다른 방안을 구비해놓았다.
바로 DLL 지연 로딩으로, 단어에서와 같이 DLL을 프로그램 실행 시에 로드하는 것이 아니라 지연하여 로딩하는 것이다.
지연 로딩은 암시적 로딩에서의 간편함과 명시적 로딩에서의 유연함, 이 두 장점을 취하고자 하는 방식으로 EXE 작성에서 DLL 링크 시에는 암시적인 방식으로, 실제 런타임에서 사용 할 때는 명시적인 방식으로 작동하도록 한 것이다.
쉽게 말해 프로그램을 실행 시에 메모리에 매핑되는 것이 아니라 해당 DLL의 Export 함수들 중 하나가 최초로 실행될 때 그 시점에 해당 DLL을 로드해서 가상 주소 공간에 매핑한다는 것이다.

5. DLL Forwarding

DLL의 Export Function Forwarding이란 Export하고자 하는 함수를, 그 기능을 대신하는 다른 DLL 내에 정의된 함수의 호출로 대체하는 것이다.
글로 설명하는 것보다는 직접 코드를 확인 것이 더 좋으므로 일반적인 경우의 DLL의 Export 함수의 주소를 확인해보자.
아래의 표와 같이 Export하는 함수의 주소로 이동을 하면 해당 함수의 내용이 존재하고 있는 것을 확인할 수 있다. 즉, 자신의 DLL 안에 해당 코드를 그대로 잘 가지고 있는 것이다.

```
text:5F923F4B mov edi, edi
.text:5F923F4D push ebp
.text:5F923F4E mov ebp, esp
.text:5F923F50 sub esp, 1Ch
.text:5F923F53 mov eax, [ebp+arg_0]
.text:5F923F56 push ebx
.text:5F923F57 push esi
.text:5F923F58 push edi ; struct CApplnMgr *
.text:5F923F59 mov edi, [ebp+arg_4]
.text:5F923F5C mov [ebp+var_C], eax
```

이번에는 DLL Forwarding이 적용된 DLL의 내용을 확인해보자.
위 표와는 다른 DLL이기는 하지만 Export 하는 함수의 주소로 이동하여 확인해보면 심히 코드가 짧다는 것을 알 수 있다.
Export 하고자 하는 함수의 이름 "LpkEditControl"과 함께 0x1000261C를 호출하는 것을 확인할 수 있다.

```
.text:10002BC8 MemCode_LpkEditControl proc near ; DATA XREF: .
rdata:off_1001E148
.text:10002BC8 ; .data:LpkEditControl
.text:10002BC8 push offset aLpkeditcontr_0 ; "LpkEditControl"
.text:10002BCD call sub_1000261C
.text:10002BD2 jmp dword ptr [eax]
```

0x1000261C에는 다시 아래와 같은 내용이 있으며, 10002634를 호출한 다음, 이전에 인자로 전달 받았던 Export하고자 하는 함수의 이름과 함께 GetProcAddress를 통해 주소를 구하고자 하는 것이다.

```
.text:1000261C call sub_10002634
.text:10002621 push [esp+lpProcName] ; lpProcName : LpkEditCon
trol
```

```
.text:10002625 push hModule ; hModule : lpk.dll
.text:1000262B call ds:GetProcAddress
.text:10002631 retn 4
```

해당 10002634를 따라가다 보면 시스템 디렉터리의 경로를 구한 뒤, 해당 Export 함수를 가진 대상(포워딩 대상)을 LoadLibarary API를 통해 로드하는 것을 확인할 수 있다.
이를 통해 해당 DLL이 로드되고 위의 과정에서와 같이 GetProcAddress를 통해 해당 함수가 로드되는 것이다.

```
.text:1000265B push esi ; uSize
.text:1000265C push eax ; lpBuffer
.text:1000265D call ds:GetSystemDirectoryA
.text:10002663 lea eax, [ebp+Buffer]
.text:10002669 push offset String2 ; "\\lpk.dll"
.text:1000266E push eax ; lpString1
.text:1000266F call ds:lstrcatA
.text:10002675 cmp hModule, 0
.text:1000267C pop esi
.text:1000267D jnz short loc_10002691
.text:1000267F lea eax, [ebp+Buffer]
.text:10002685 push eax ; lpLibFileName
.text:10002686 call ds:LoadLibraryA ; lpk.dll
```

6. DLL의 메모리 상주

? : “DLL은 메모리에 한 번 올라가면, 이 DLL을 공유하는 프로세스가 모두 종료될 때까지 메모리에 존재한다.”
1) Process가 DLL를 Load 한다. → Process 가상 메모리에 DLL이 맵핑되었음을 의미
2) Process의 가상 메모리는 Process 소유의 가상 메모리로써 Process 별 독립적 존재 영역
! : “DLL은 물리 메모리에 한 번 올라가면, 이 DLL을 공유하는 프로세스가 모두 종료될 때까지 물리 메모리에 존재한다.”

1. 프로세스 A가 실행되며 X.dll을 로드한다.
   따라서 X.dll은 A의 가상 메모리 영역에 맵핑되면서 물리 메모리에 할당된다.

2. 프로세스 B가 실행되며 X.dll을 로드한다.
   이미 1단계에서 X.dll이 물리 메모리에 올라 있으므로 그대로 참조할 수 있도록 프로세스 B의 가상 메모리 영역에 맵핑만 한다.

   이 상황에서 유의할 것이 X.dll 이 할당된 가상 메모리 주소이다. 프로세스 A와 B가 동일한 주소에 할당한다.

   이것이 DLL 공유가 가능한 이유이다. 다시 말해서 두 프로세스가 동일한 DLL을 동일한 가상 주소에 맵핑했기 때문에,
   페이지 단위로 공유가 가능하다는 것이다.

3. 프로세스 A가 종료되었다. 그러나 B가 종료되지 않았기 때문에,
   X.dll은 여전히 물리 메모리에 남아있다.

4. 프로세스 B가 종료되었다. 이제 X.dll을 참조하는 프로세스가 하나도 존재하지 않기 때문에 X.dll도 할당된 물리 메모리에서 반환된다.

7. DLL 중복 Loading

DLL이 빌드될 때 DLL이 할당되어야 할 가상 메모리 주소가 Linker에 의해 결정됨
→ 만약 X.dll을 0x20000000번지에 맵핑하기로 함 → 모든 Process는 해당 번지에 X.dll의 주소를 맵핑

1. 프로세스 A가 X.dll을 로드하면서 가상 메모리 주소 0x10000000에 맵핑하였다.
2. 프로세스 B가 Y.dll을 로드하면서 가상 메모리 주소 0x10000000에 맵핑하였다.
3. 프로세스 B가 X.dll을 로드하게 되면, 이미 0x10000000에 Y.dll이 존재하고 있으므로,다른 주소의 메모리 영역에 X.dll을 올리게 된다.

Therefore

- 위와 같은 상황이 발생하면 두 개의 X.dll이 물리 메모리에 올라가게 된다.
  즉, DLL도 물리 메모리에 중복적으로 올라갈 수 있음

[Windows vs Linux]

무슨 일이 있어도:

- 각 프로세스에는 자체 주소 공간이 있습니다. 즉, 일부 프로세스 간 통신 라이브러리 또는 확장을 사용하지 않는 한 프로세스간에 메모리가 공유되지 않습니다.
- ODR (One Definition Rule)이 여전히 적용됩니다. 즉, 링크 타임에 전역 변수의 정의를 한 번만 볼 수 있습니다 (정적 또는 동적 링크).

여기에서 중요한 문제는 실제로 가시성입니다.

모든 경우에 정적 전역 변수 (또는 함수)는 모듈 (dll / so 또는 실행 파일) 외부에서 결코 볼 수 없습니다. C 표준은 내부 링크를 필요로합니다.
즉, 정의 된 변환 단위 (오브젝트 파일이됩니다) 밖에서는 볼 수 없습니다. 그래서, 그 문제를 해결합니다.

복잡 해지는 곳은 extern 전역 변수가있을 때입니다.

여기서 **Windows**와 **Unix** 계열 시스템은 완전히 다릅니다.

Windows (.exe 및 .dll)의 경우 extern 전역 변수는 내 보낸 심볼의 일부가 아닙니다.
즉, 다른 모듈은 다른 모듈에 정의 된 전역 변수를 전혀 알지 못합니다.
예를 들어, DLL에 정의 된 extern 변수를 사용하도록되어있는 실행 파일을 만들려고하면 허용되지 않기 때문에 링커 오류가 발생합니다.
extern 변수의 정의와 함께 객체 파일 (또는 정적 라이브러리)을 제공하고 실행 파일과 DLL 모두에 정적으로 링크해야하므로 두 개의 별개의 전역 변수가 생깁니다 (하나는 실행 파일에 속하고 다른 하나는 DLL에 속합니다).
Windows에서 전역 변수를 실제로 내보내려면 함수 내보내기 / 가져 오기 구문과 유사한 구문을 사용해야합니다.
즉,

```
#ifdef COMPILING_THE_DLL
#define MY_DLL_EXPORT extern "C" __declspec(dllexport)
#else
#define MY_DLL_EXPORT extern "C" __declspec(dllimport)
#endif

MY_DLL_EXPORT int my_global;
```

그렇게하면 전역 변수가 내 보낸 기호 목록에 추가되고 다른 모든 기능과 마찬가지로 연결할 수 있습니다.
유닉스와 같은 환경 (Linux와 같은)의 경우, 확장자가 “shared objects”인 동적 라이브러리는 모든 extern 전역 변수 (또는 함수)를 내 보냅니다.
이 경우 어디서나 공유 객체 파일에 대한로드 타임 링크를 수행하면 전역 변수가 공유됩니다.
즉, 전역 변수가 하나로 링크됩니다.
기본적으로 유닉스 계열 시스템은 정적 라이브러리 또는 동적 라이브러리와의 연결에 차이가 없도록 설계되었습니다. 다시 말하면 ODR은 보드 전체에 적용됩니다.
extern 전역 변수는 모듈간에 공유됩니다.
즉, 로드 된 모든 모듈에서 하나의 정의 만 있어야합니다.
마지막으로 Windows 또는 Unix 계열 시스템의 경우 LoadLibrary () / GetProcAddress () / FreeLibrary () 또는 dlopen () / dlsym () / Dlsym ()을 사용하여 동적 라이브러리의 런타임 링크를 수행 할 수 있습니다. dlclose ().
이 경우, 사용하고자하는 심볼 각각에 대한 포인터를 수동으로 가져와야하며 사용하고자하는 전역 변수가 포함되어 있어야합니다.
전역 변수의 경우 전역 변수가 내 보낸 기호 목록의 일부인 경우 (앞 단락의 규칙에 따라) 함수에 대해 수행하는 것과 똑같은 방식으로 GetProcAddress () 또는 dlsym ()을 사용할 수 있습니다.

필요한 최종 메모로서 :

전역 변수는 피해야합니다. 그리고 당신이 인용 한 텍스트 (“명확하지 않은”것에 대해)가 방금 설명한 플랫폼 별 차이점을 정확히 참조하고 있다고 생각합니다 (동적 라이브러리는 실제로 C 표준에 의해 정의되지 않았으며 이는 플랫폼 별 영역입니다. 안정성 / 휴대 성이 훨씬 떨어짐).

[DLL과 프로세스 주소 공간]

DLL 파일의 이미지는 실행 파일이나 다른 DLL이 DLL 내 포함되어 있는 함수를 호출하기 전에
반드시 **프로세스의 주소 공간에 매핑**되어 있어야 한다.

이를 위해 다음 두 가지 방법 중 하나를 선택할 수 있다.

- 암시적 로드타임 링킹
- 명시적 런타임 링킹

위 방법들에 대해선 아래 챕터에서 각각 자세히 설명하도록 하겠다.

DLL 파일 이미지가 프로세스의 주소 공간에 매핑되고 나면,
DLL이 가지고 있는 모든 함수들은 프로세스 내의 모든 쓰레드에 의해 호출될 수 있게 된다.

사실 이렇게 로드가 완료되고 나면, DLL 고유의 특성은 거의 없어진다고 볼 수 있다.

프로세스 내의 쓰레드 관점에서는 DLL이 가지고 있는 코드와 데이터들은
단순히 프로세스의 주소 공간에 로드된 추가적인 코드와 데이터들로 여겨질 뿐이다.

쓰레드가 DLL에 포함되어 있는 함수를 호출하게 되면,
호출된 DLL 함수는 호출한 쓰레드의 스택으로부터 전달된 인자 값을 얻어내고,
호출한 쓰레드의 스택을 이용하여 지역변수를 할당하게 된다.

뿐만 아니라 DLL 함수 내부에서 생성하는 모든 오브젝트들도
DLL 함수를 호출하는 쓰레드나 프로세스가 소유하게 되며, DLL 자체가 소유하는 오브젝트는 존재하지 않는다.

예를 들어, DLL 내의 특정 함수가 VirtualAlloc 함수를 호출하게 되면,
해당 함수를 호출한 쓰레드가 속해 있는 프로세스의 주소 공간 내에 영역이 예약된다.

만일 DLL이 프로세스의 주소 공간으로부터 내려간다 하더라도
앞서 프로세스의 주소 공간에 예약했던 영역은 그대로 남게 되는데,
이는 시스템이 해당 영역이 DLL로부터 예약되었다는 사실을 특별히 관리하지 않기 때문이다.

하지만, **단일의 주소 공간은 하나의 실행 모듈과 다수의 DLL 모듈로 구성**되어 있음을 반드시 알아두어야 한다.

이 중 일부 모듈은 C/C++ 런타임 라이브러리를 정적으로 링크하고 있을 수도 있으며,
또 다른 모듈은 C/C++ 런타임 라이브러리를 동적으로 링크하고 있을 수도 있다.

따라서, **단일의 주소 공간 내에 C/C++ 런타임 라이브러리가 여러 번 로드될 수 있다**는 사실을 잊어버리면 곤란한다.
아래 예제를 살펴보자.

```
1. -- 실행 파일의 함수 --
2. void EXEFunc()
```

```
3. {
4.     void* pv = DLLFunc();
5.
6.     // pv가 가리키는 저장소를 사용한다.
7.
8.     // pv가 EXE의 C/C++ 런타임 힙 내에 있을 것이라고 가정
한다.
9.     free(pv);
10. }
11.
12. -- DLL의 함수 --
13. void* DLLFunc()
14. {
15.     // DLL의 C/C++ 런타임 힙으로부터 메모리를 할당받는다.
16.     return (malloc(100));
17. }
```

이 코드가 정상적으로 동작할 것인가?
DLL 함수 내에서 할당받은 메모리 블럭을 EXE 함수 내에서 정상적으로 해제할 수 있는가?

위 예제는 제대로 동작할수도 있고, 그렇지 않을 수도 있다.

만일 EXE와 DLL이 모두 DLL로 구성된 C/C++ 런타임 라이브러리를 사용하고 있다면, 위 예제는 정상 동작한다.

하지만, 둘 중 하나라도 C/C++ 런타임 라이브러리를 정적으로 링크하고 있다면, free 호출 과정에서 문제가 발생할 것이다.

이러한 문제는 애초에 습관을 제대로 들이면 된다.
DLL 내 메모리를 할당하는 함수가 있다면, 해제하는 함수도 DLL에 만들고 그걸 사용하는 것이다.

```
1. -- 실행 파일의 함수 --
2. void EXEFunc()
3. {
```

```
    // DLL 함수에서 할당한 메모리 블럭의 주소를 얻어온다.
    void* pv = DLLAllocFunc();

    // pv가 가리키는 메모리 블럭을 사용한다.

    // DLL 함수를 이용해 pv를 메모리로 반환한다.
     DLLFreeFunc(pv);
}

-- DLL의 할당 함수 --
void* DLLAllocFunc()
{
    // DLL의 C/C++ 런타임 힙으로부터 메모리를 할당받는다.
    return (malloc(100));
}

-- DLL의 해제 함수 --
void DLLFreeFunc(void* p)
{
    // DLL의 C/C++ 런타임 힙에서 메모리를 해제한다.
    free(p);
}
```

그리고, 실행 파일 내에 전역으로 선언된 정적 변수는 동일한 실행 파일이
여러 번 실행될 경우라도 Copy-on-write 메커니즘에 의해 공유되지 않는다.

DLL 파일 내에 전역으로 선언된 정적 변수 역시 이와 동일한 메커니즘이 적용된다.

프로세스가 DLL 이미지 파일을 자신의 주소 공간 내에 매핑하는 경우
실행 파일의 경우와 동일하게 전역으로 선언된 정적변수의 새로운 인스턴스가 생성된다.

이를 공유하게 하는 방법에 대해선 PE/COFF의 Section 문서의 챕터 4. 공유 섹션을 보기 바란다.

[Quote]

1. https://yonghello.tistory.com/entry/Dll%EC%9D%B4%EB%9E%80
2. https://kali-km.tistory.com/entry/DLL%EC%9D%B4%EB%9E%80
3. http://www.sck.pe.kr/c-cgi/whatisdll.htm
4. https://sinun.tistory.com/99
5. https://codeday.me/ko/qa/20190314/48403.html
6. http://egloos.zum.com/sweeper/v/2991664 : 꼭 다시 읽어보기