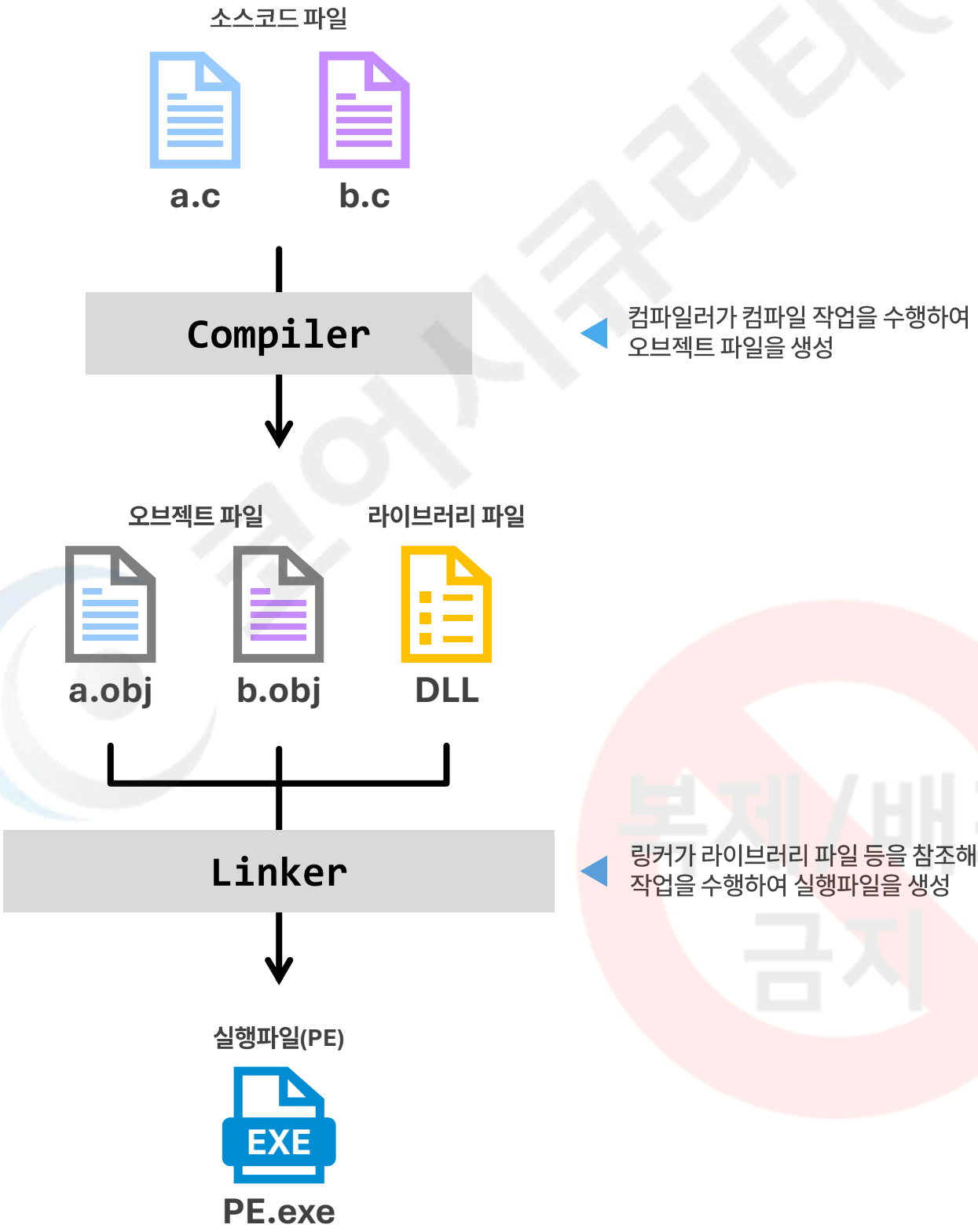
참고 자료

- 리버스 코드 엔지니어링 훈련

윈도우 실행파일의 생성 및 실행과정

윈도우 실행파일의 생성과정

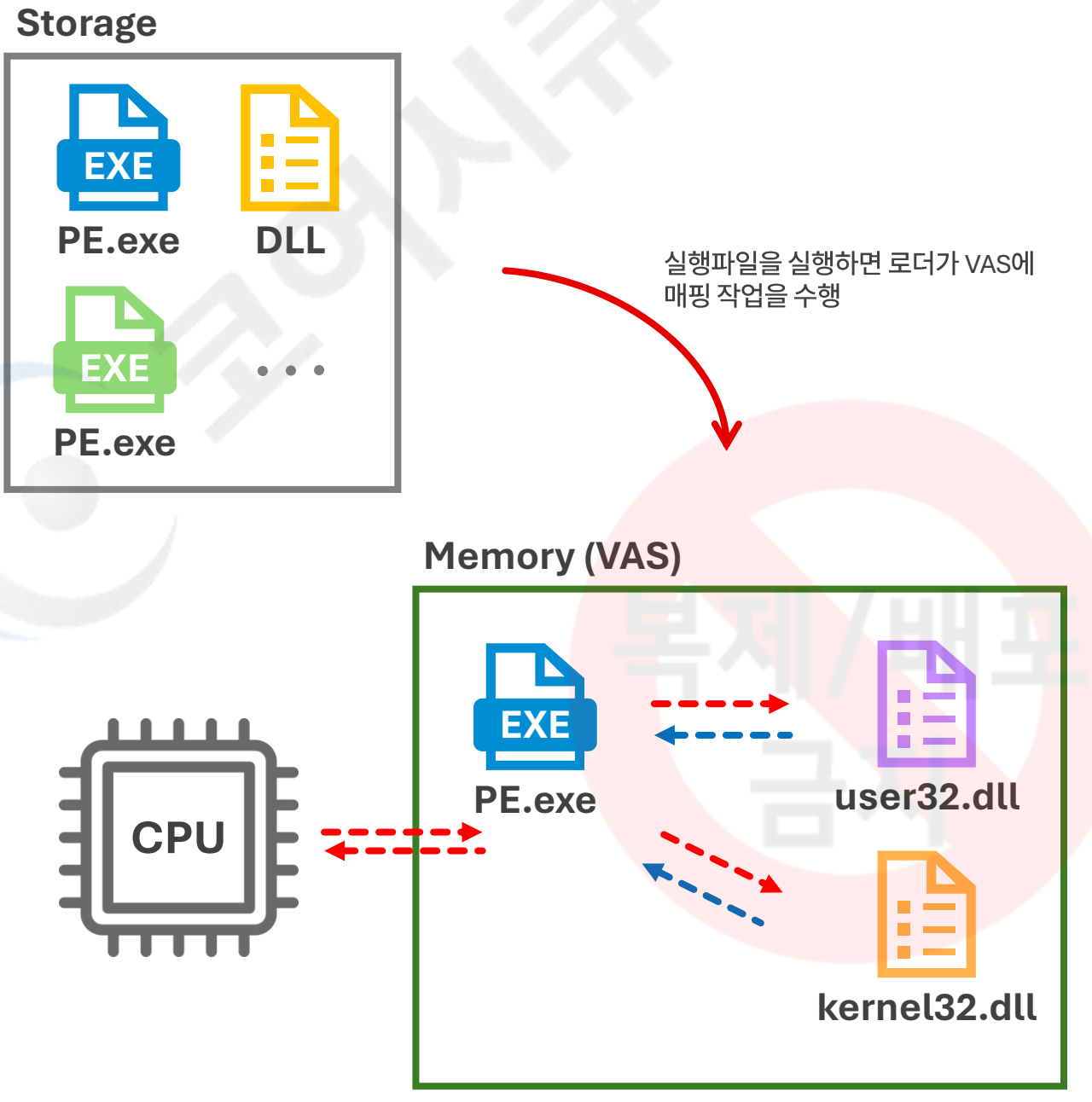
- 윈도우 실행파일은 소스코드 파일, 오브젝트 파일, 실행파일(PE) 순서대로 만들어지며, 이러한 과정에서 각 단계별로 컴파일러(Compiler)와 링커(Linker)가 관여를 합니다.
- 컴파일 단계에서 컴파일러의 주요 업무 중 하나는 소스코드를 아키텍처에 맞는 기계어 코드로 변환하는 것입니다.



윈도우 실행파일의 생성 및 실행과정

윈도우 실행파일의 실행과정

- 폰노이만 아키텍처의 개념에 따라 실행파일은 평소에는 스토리지에 존재, 실행 시 메모리에 로드(매핑), CPU에 의한 코드 실행 흐름을 가집니다.
- 실행파일이 실행될 경우 ntdll.dll 모듈에 정의되어 있는 로더 함수들에 의해 실행파일의 콘텐츠가 메모리에 매핑됩니다.
- 운영체제는 매핑과정에서 메모리 페이지에 대한 프로텍션(실행, 쓰기, 읽기 등)을 부여하여 보호합니다.
- 실행파일이 메모리상에 매핑된 후 함수를 호출하는 데 필요한 DLL을 추가로 메모리상에 매핑한 후 API에 대한 주소값을 조사합니다.
- 이후 실행파일은 필요에 따라 메모리상의 DLL 파일 내에 정의된 API를 호출하고 반환값을 참조하며 기능을 수행합니다.
- 물론 메모리상에 매핑된 EXE 혹은 DLL 파일에 있는 컴파일된 코드들을 실행하는 주체는 CPU 입니다.



운영체제의 프로세스 가상주소공간

가상주소공간의 등장 배경

- **충분하지 못한 메모리 문제 (not enough memory)**
  - 프로세서는 아키텍처에서 제공하는 메모리 영역 전체에 해당하는 주소값 사용 가능
  - 32비트 아키텍처의 경우 4GB 메모리를 지원하므로 0x00000000 ~ 0xFFFFFFFF 범위 내의 주소 사용 가능
  - 시스템에 설치된 물리 메모리의 크기가 아키텍처가 지원하는 메모리의 크기보다 작을 수 있음
  - 물리 메모리의 크기를 넘어서는 주소값을 이용하여 접근 시도하는 경우, 크래시 발생 가능

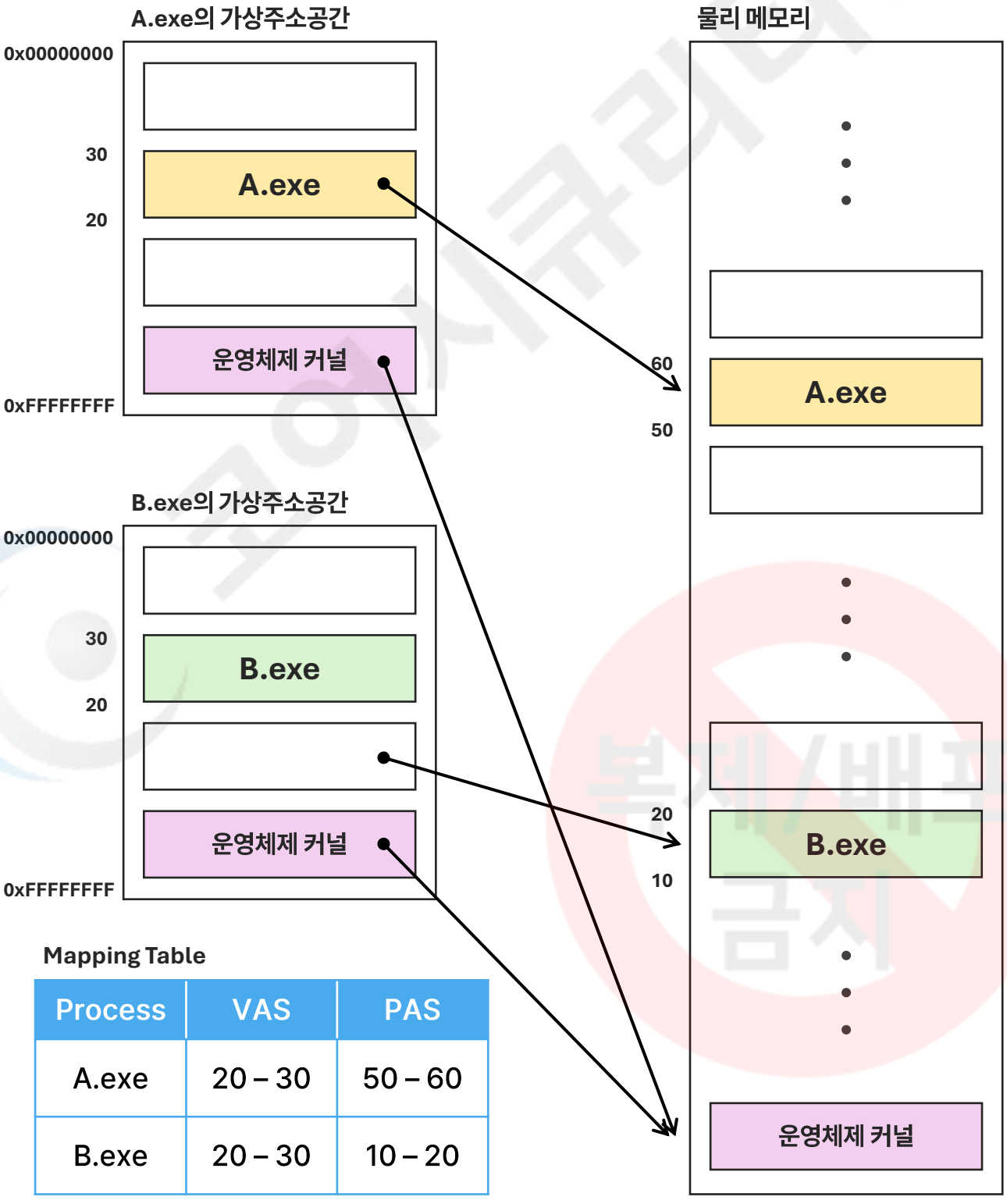
- **프로그램 간 사용 메모리 접근 통제 문제 (keeping program secure)**
  - 실행중인 프로그램들이 동일한 물리 메모리 주소에 접근하여 데이터를 읽고 쓰는 경우, 데이터 오염이나 중요 정보 노출같은 문제점 발생

- **단편화로 인한 메모리 낭비 문제 (holes in memory)**
  - 근본적인 문제는 실행중인 프로그램(프로세스)이 동일한 메모리 공간을 사용한다는 것

운영체제의 프로세스 가상주소공간

윈도우 운영체제의 프로세스 가상주소공간 운용 개념

- 프로세스에게 독립적이고 베타적으로 사용할 수 있는 가상의 주소 공간을 부여
- 프로세스가 사용하는 가상의 주소 공간과 실제 물리 주소 공간의 맵핑은 운영체제가 관리
- 맵핑 정보는 프로세스 별 별도의 맵핑 테이블에 저장
- 프로세스는 물리 메모리 주소 공간을 바라볼 수 없고 직접 접근 불가능

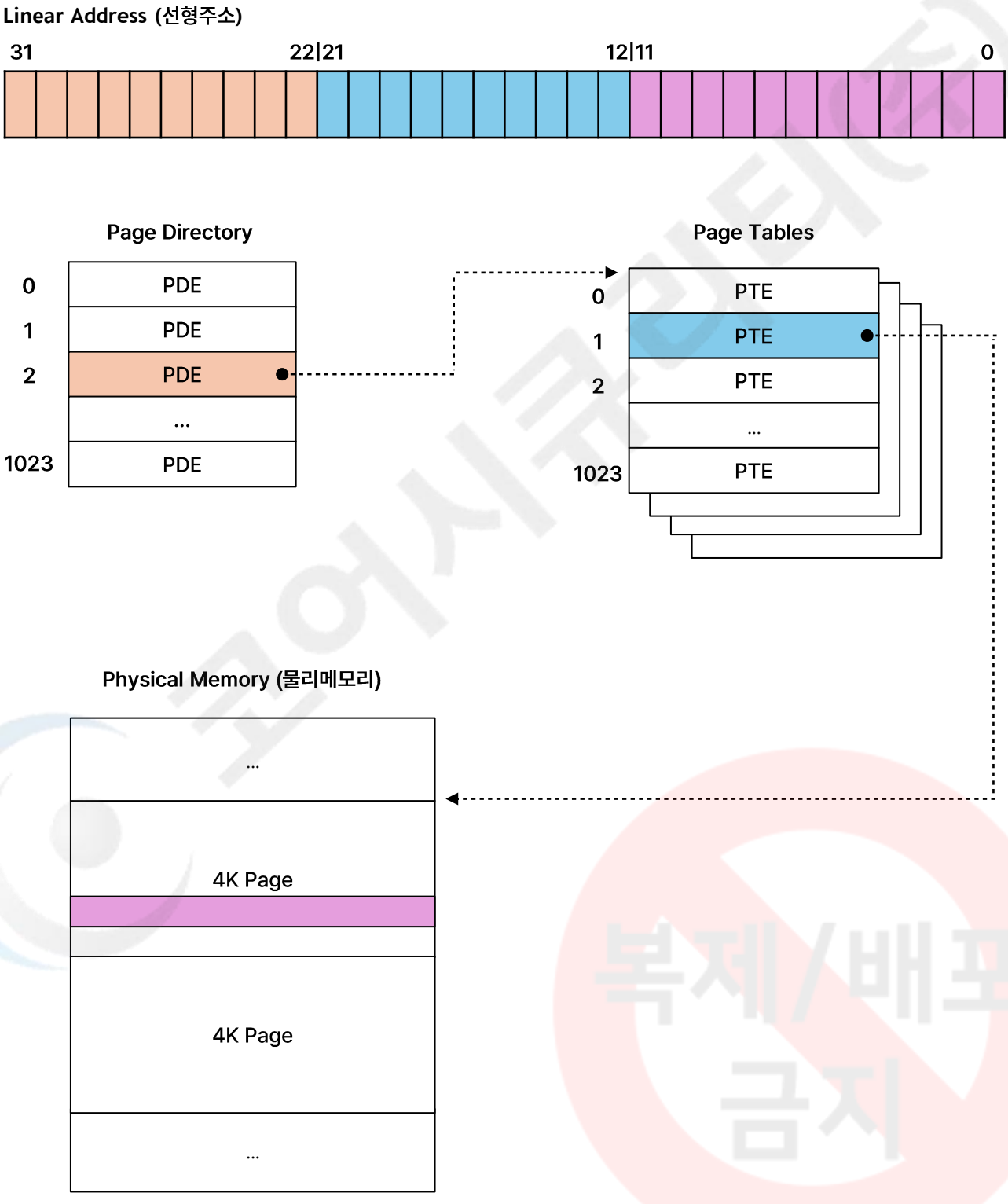


Mapping Table

Process	VAS	PAS
A.exe	20 – 30	50 – 60
B.exe	20 – 30	10 – 20

운영체제의 프로세스 가상주소공간

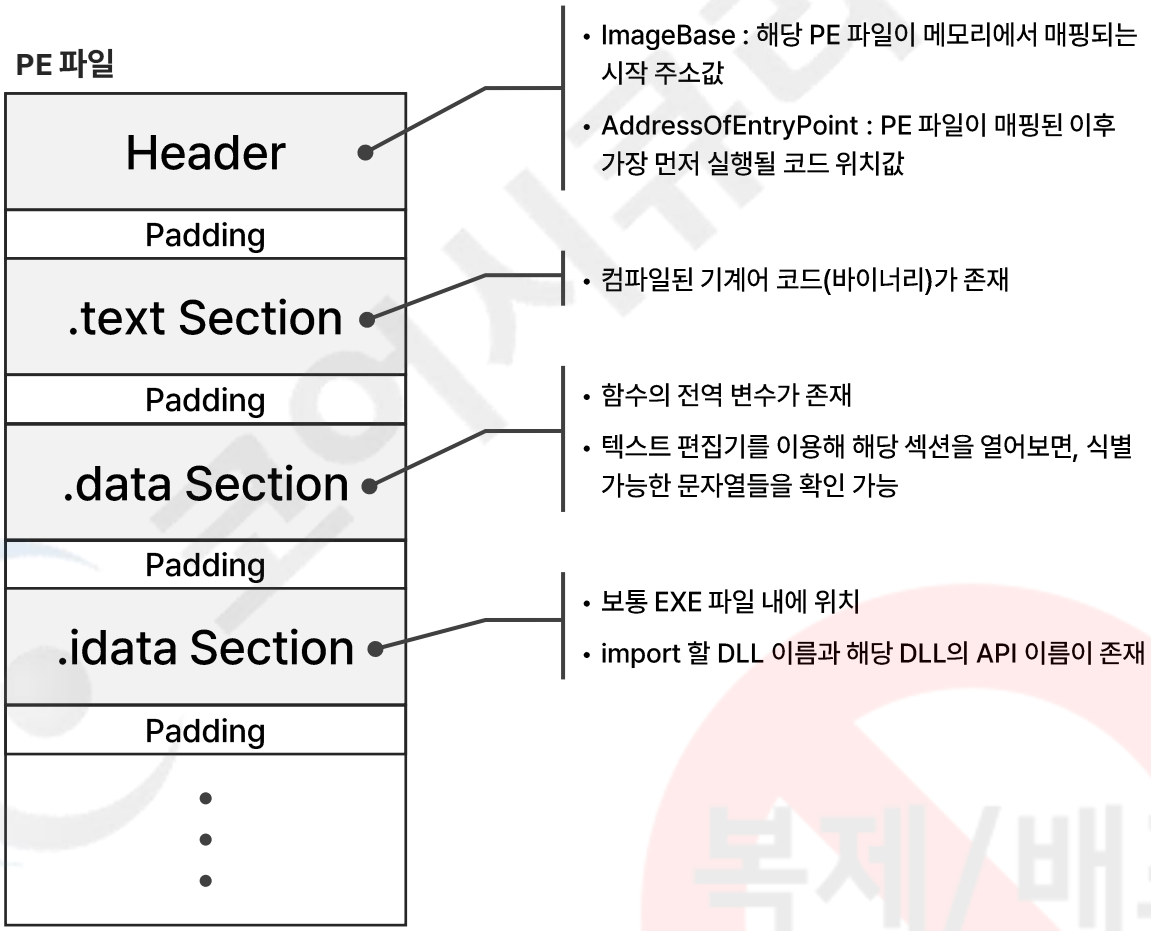
페이징 (Paging)



PE 파일의 개괄적 구조와 실행 전후 변화

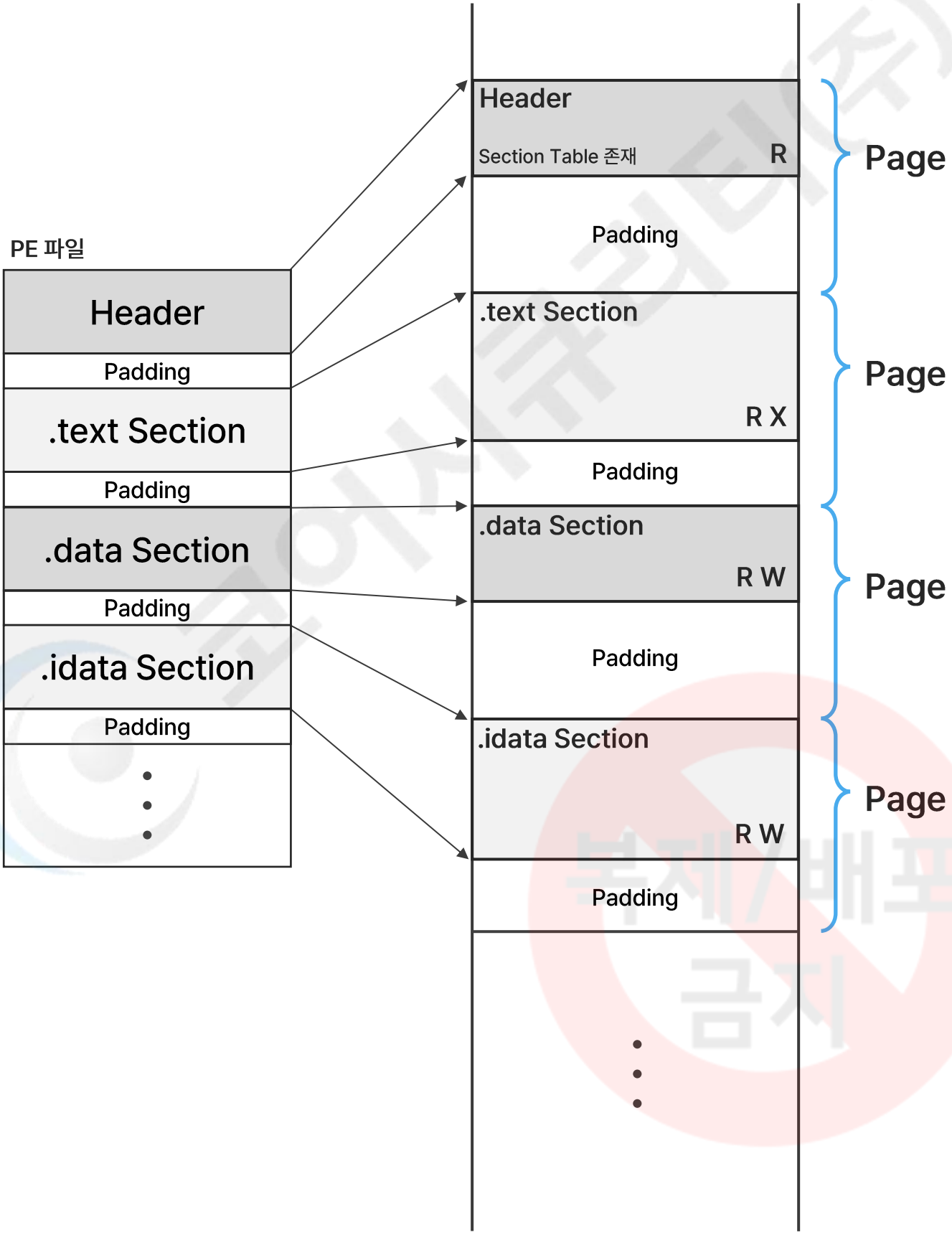
PE 파일의 실행과정

- PE(PE-COFF) 파일은 윈도우 운영체제에서 사용되는 EXE, DLL, SYS 등의 확장자를 가진 파일들에 대한 파일 형식입니다.
- 동적 라이브러리 정보, API 익스포트 및 임포트 정보, 실행을 위한 코드, 실행에 필요한 리소스 정보 등을 구조화 하고 관리하는 방법을 정의하고 있습니다.
- 실행 과정에서 로더에 의해 가상메모리 공간에 매핑된 후에도 파일의 레이아웃은 거의 바뀌지 않습니다.



PE 파일의 개괄적 구조와 실행 전후 변화

PE 파일의 실행 후 과정



DOS 헤더와 DOS Stub 코드

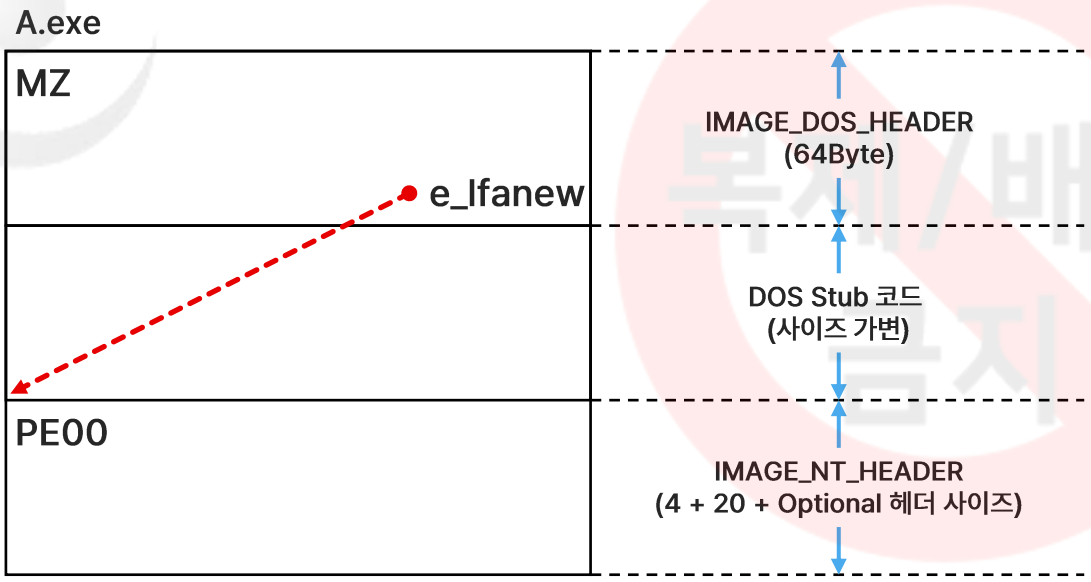
DOS 헤더

- DOS 헤더는 MS-DOS와의 호환을 위해 유지되고 있는 헤더이며 포함된 대부분의 데이터가 사용되지 않습니다.
- PE 파일은 DOS 헤더로 시작하며 디스크 상에서는 파일의 첫 부분이 됩니다.
- MS-DOS 개발자 중 한 명인 'Mark Zbikowski'의 이니셜인 'MZ' 가 DOS 헤더의 시그니처로 사용됩니다.
- DOS 헤더는 항상 64바이트 사이즈로 구성되며, IMAGE_DOS_HEADER 자료구조로 정의되어 있습니다.

```
#define IMAGE_DOS_SIGNATURE                 0x5A4D      // MZ
#define IMAGE_NT_SIGNATURE                  0x00004550  // PE00
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number "MZ"
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
        . . .
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

DOS Stub 코드

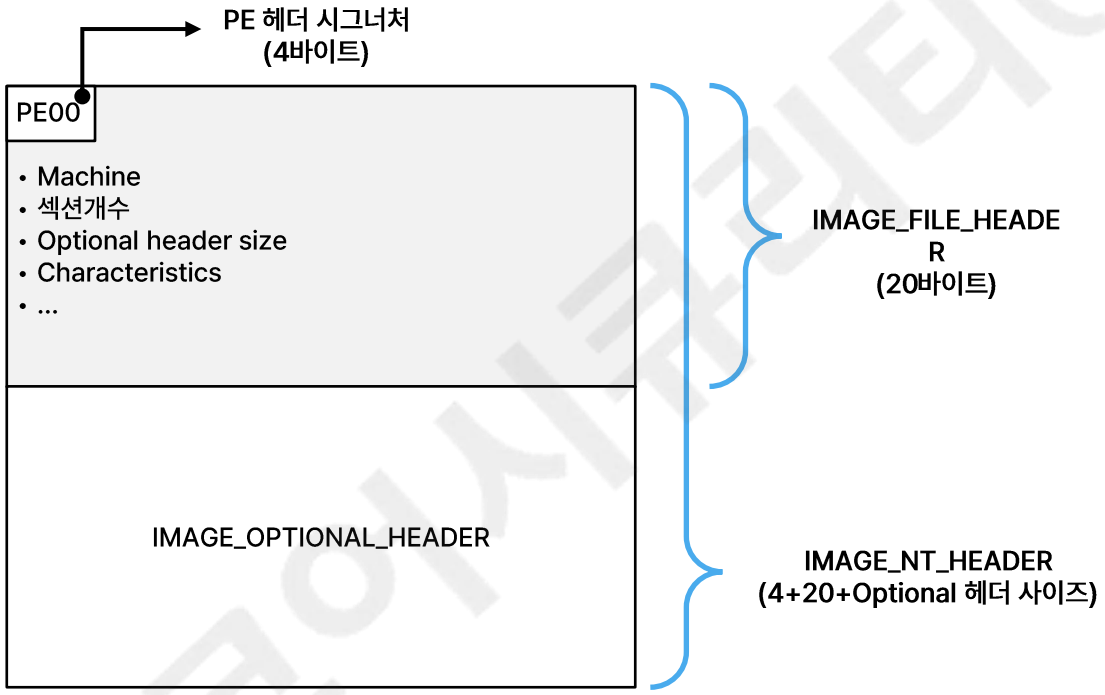
- DOS Stub 코드는 프로그램을 도스 모드로 실행시켰을 때 실행되는 코드를 담고 있습니다.
- DOS Stub 코드는 'This program cannot be run in DOS mode.'라는 문자열과 그 문자열을 화면에 출력해주는 기계어 코드로 이루어져 있습니다.



PE 헤더

IMAGE_NT_HEADER

- PE 헤더의 위치는 IMAGE_DOS_HEADER.e_lfanew 멤버를 통해서 접근이 가능합니다.
- IMAGE_NT_HEADER 구조로 이루어져 있으며 시그너처인 "PE00" 문자열로 시작합니다.
- IMAGE_NT_HEADER는 IMAGE_FILE_HEADER, IMAGE_OPTIONAL_HEADER를 포함합니다.



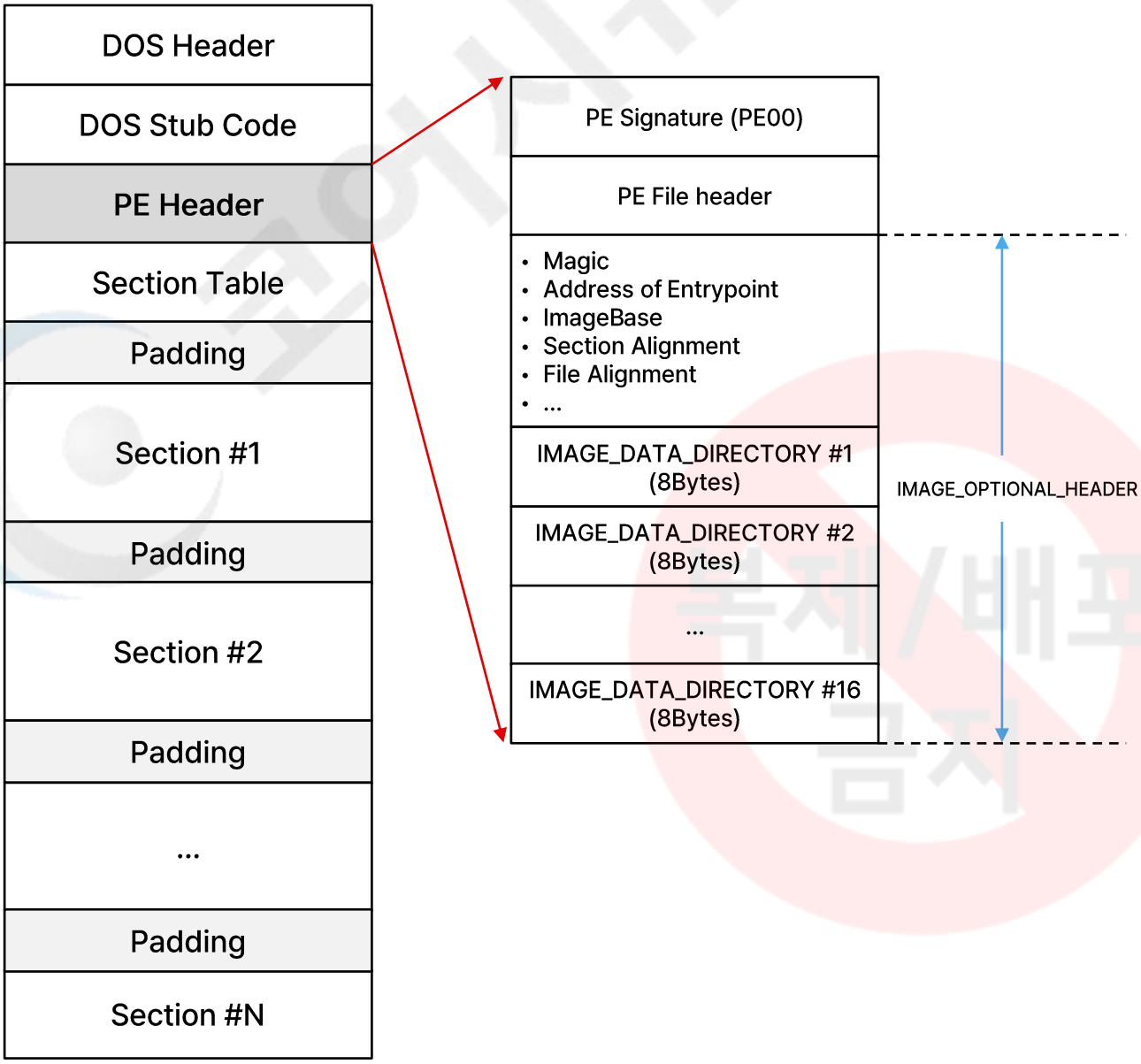

```
typedef struct _IMAGE_NT_HEADERS {
        DWORD Signature;
        IMAGE_FILE_HEADER FileHeader;
        IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADER32, *PIMAGE_NT_HEADERS32;
```

```
typedef struct _IMAGE_NT_HEADERS {
        WORD  Machine;
        WORD  NumberOfSections;
        DWORD TimeDataStamp;
        DWORD PointerToSymbolTable;
        DWORD NumberOfSymbols;
        WORD  SizeOfOptionalHeader;
        WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER
```

PE 헤더

IMAGE_OPTIONAL_HEADER

- IMAGE_OPTIONAL_HEADER 구조로 이루어져 있으며 이름과는 다르게 PE 파일의 논리적인 구조에 대한 매우 중요한 정보들을 담고 있습니다.
- PE 헤더의 마지막 구성요소이며 30개의 필드와 1개의 데이터 디렉토리로 구성되어 있습니다.
- 실행파일 내에서 가장 먼저 실행되는 코드의 가상주소를 찾아야 한다면 ImageBase와 AddressOfEntryPoint을 더한 값을 확인하면 됩니다.
- **ImageBase**
  - 실행파일이 로드될 가상주소공간의 주소
  - 가상주소로 표현됨
- **Address of Entrypoint**
  - 메모리에 로드된 후 가장 먼저 실행될 코드의 위치
  - Offset 값으로 표현됨

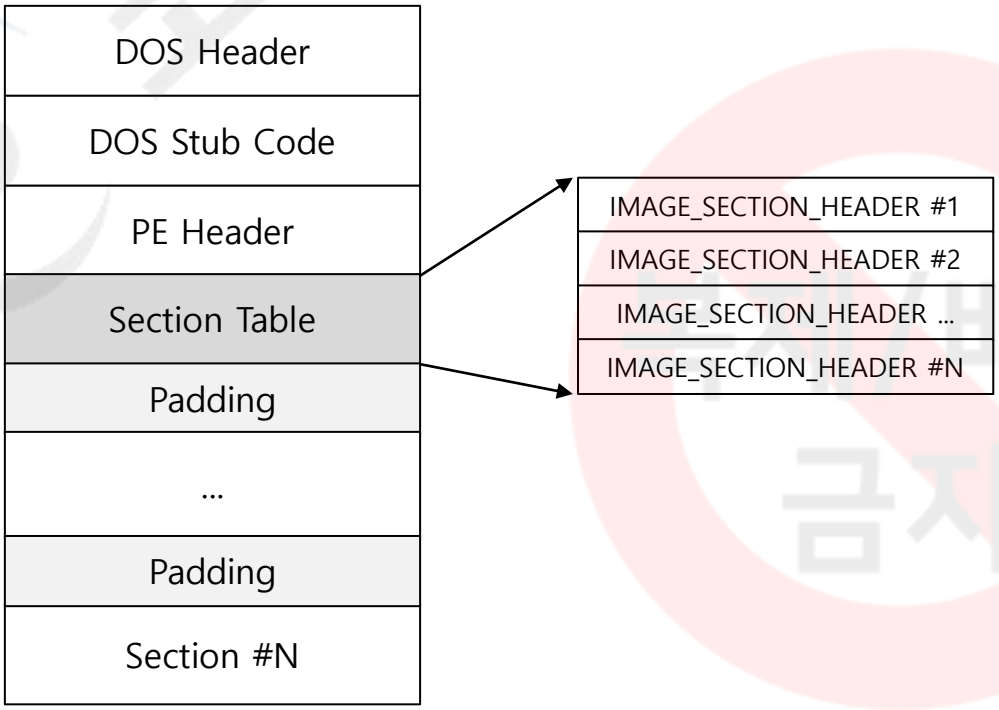


섹션테이블

섹션테이블

- 섹션테이블은 실행파일 내에 존재하는 섹션들의 파일 상의 위치, 가상주소 상의 위치, 가상주소공간에서 가져야 할 프로텍션 정보 등이 포함되어 있습니다.
- 섹션들은 정보가 고정값을 가지지 않고 가변적이기 때문에, 해당 정보가 적힌 섹션테이블을 로더가 참조하여 메모리에 정상적으로 로드할 수 있도록 합니다.
- 섹션테이블은 동일한 자료구조가 배열처럼 연속으로 나열된 형태를 가집니다.

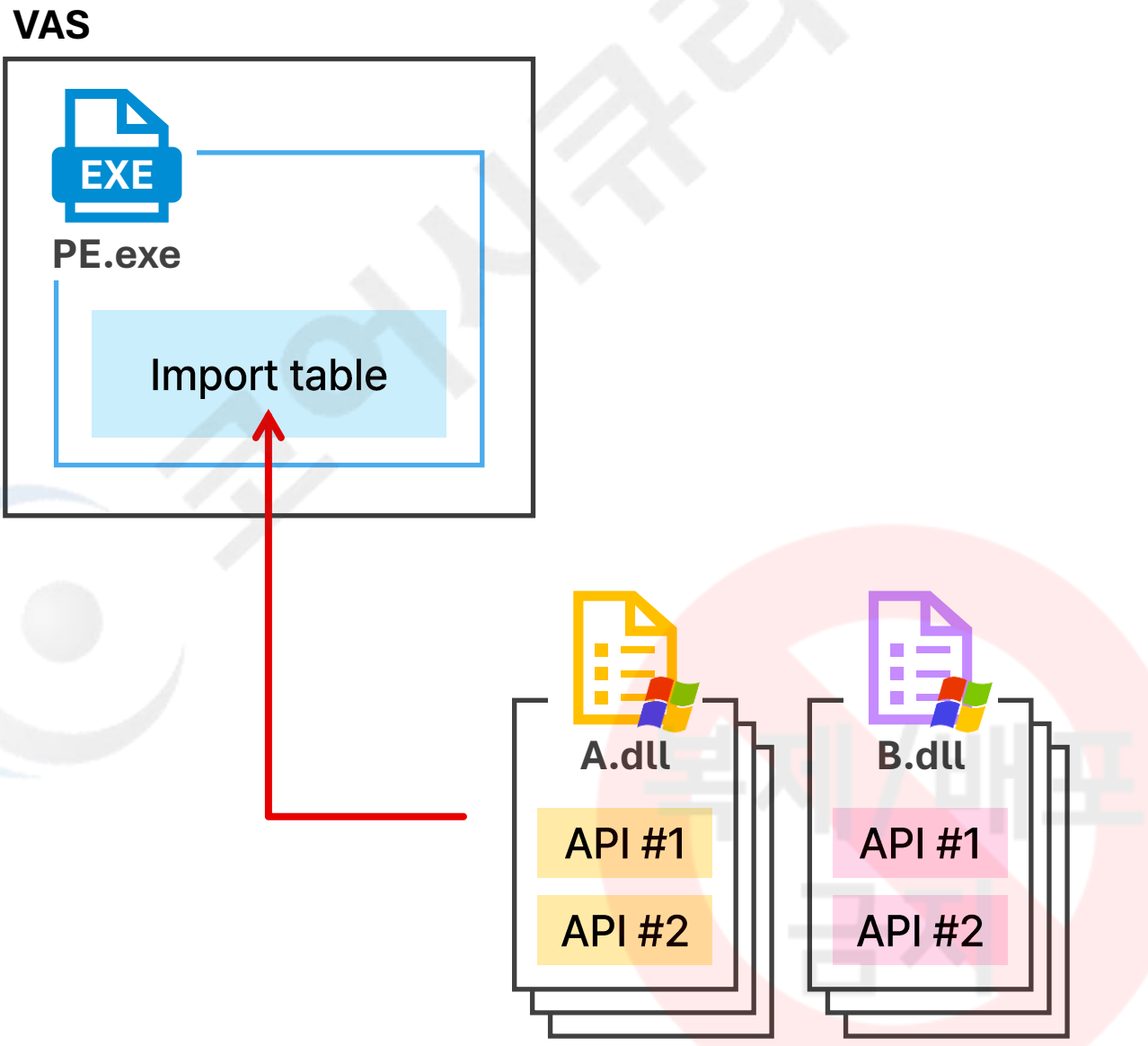
```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
    DWORD PhysicalAddress;
    DWORD VirtualSieze;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```



임포트 메커니즘

개요

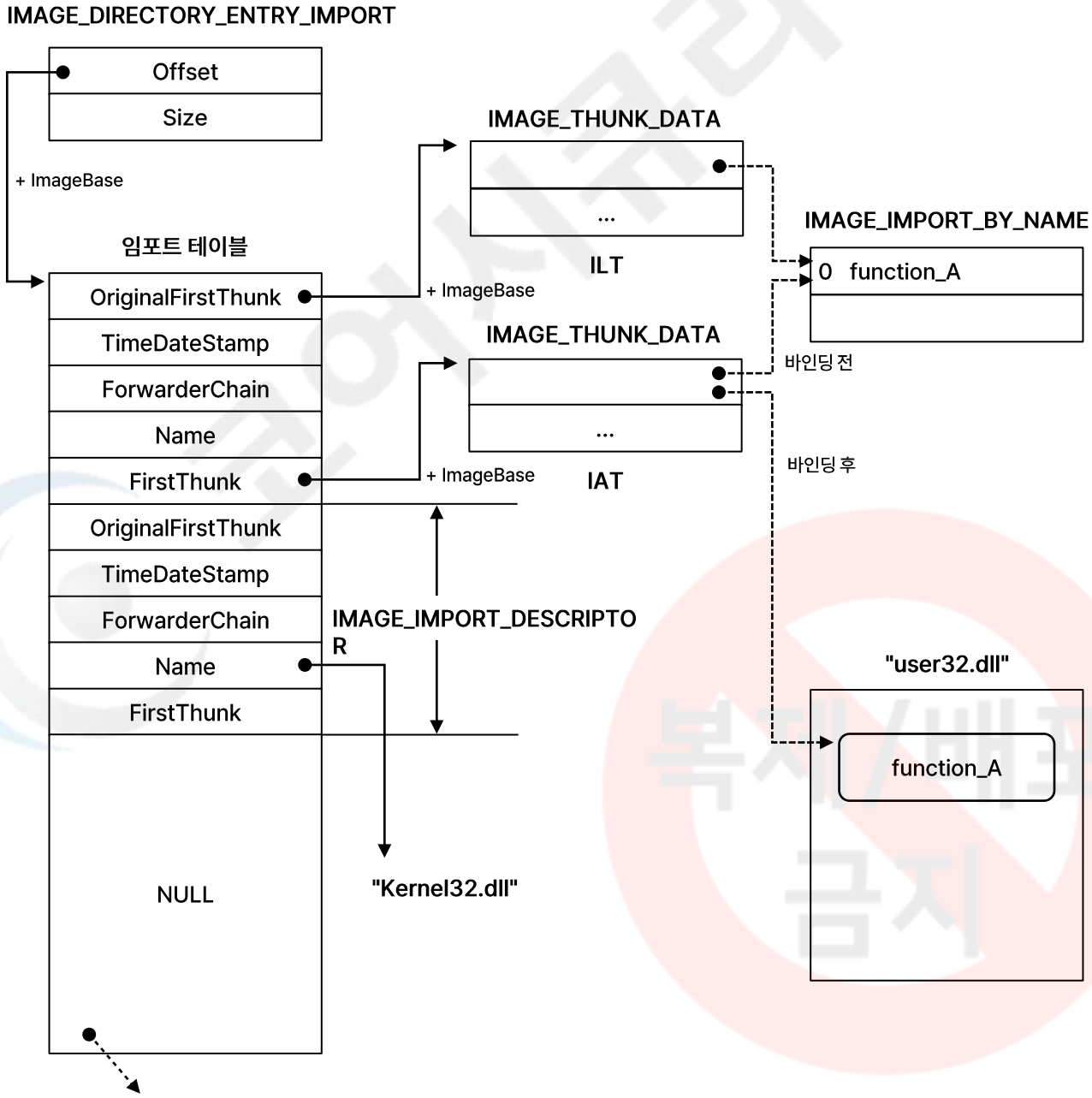
- 윈도우 운영체제의 기능을 사용하기 위해서는 DLL에 정의된 API를 호출해야 하기 때문에, EXE 확장자를 가진 PE-COFF 파일은 가상주소공간에서 매핑되어도 혼자 일을 할 수 없습니다.
- 이로 인해, PE-COFF 포맷에는 실행 과정에서 참조해야 하는 DLL과 API 관련 정보들을 구조화하여 유지하며, 이를 임포트 테이블이라 합니다.
- 바인딩(Binding)에 의해 임포트 테이블은 저장매체 상에 존재할 때와 실행되어 메모리에 존재할 때의 내용이 약간 달라집니다.



임포트 메커니즘

PE 파일의 임포트 매커니즘

- IMAGE_IMPORT_DESCRIPTOR 자료구조로 이루어져 있으며 실행파일이 참조하는 DLL, API와 관련된 정보들을 관리합니다.
- IMAGE_THUNK_DATA, IMAGE_IMPORT_BY_NAME 자료구조들을 직간접적으로 참조하여 호출될 API의 이름과 주소를 관리합니다.
- 로더는 실행파일을 메모리에 매핑한 후 바인딩(Binding)을 수행하여 가상주소공간에 있는 참조할 API의 주소값을 확보합니다.
- 확보된 각 API의 주소는 IAT(Import Address Table)에 기록됩니다.



임포트테이블의 엔트리 개수는 임포트한 DLL 개수 + 1

IA32 주요 레지스터 세트

범용 레지스터 (General-purpose Register)

- 범용 레지스터는 산술/논리 연산에 사용되는 피연산자 정보, 주소 계산을 위한 피연산자 정보, 메모리 포인터 정보 등을 담고 있는 레지스터들을 말합니다.

[General-purpose register]

EAX	Accumulator for operands and results data.
EBX	Pointer to data in the DS segment.
ECX	Counter for string and loop operations.
EDX	I/O pointer.
ESI	Source pointer for string operations.
EDI	Destination pointer for string operations.
EBP	Pointer to data on the stack
ESP	Stack pointer

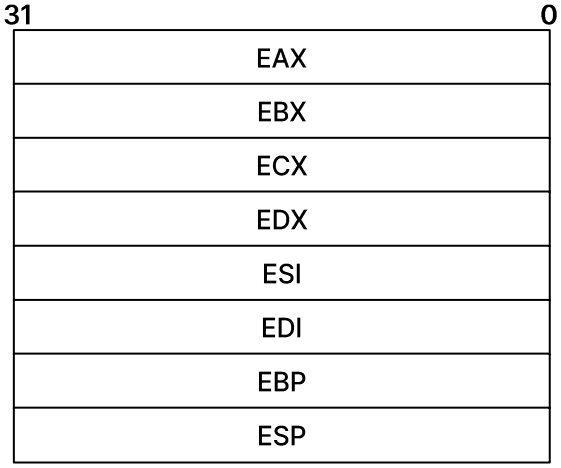
세그먼트 레지스터 (Segment Register)

- 세그먼트 레지스터는 GDT(Global Descriptor Table, 메모리상의 세그먼트 정보를 담은 자료구조)의 인덱스 값을 포함하고 있는 레지스터들을 말합니다.

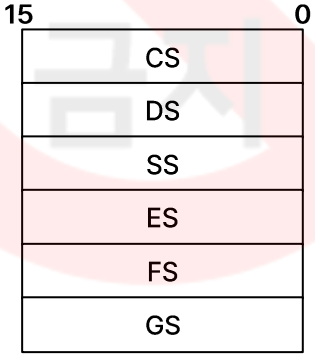
[Segment register]

CS	Code segment register
DS	Data segment register
SS	Stack segment register
ES	Extra segment register
FS	.
GS	.

[General-purpose register]



[Segment register]



IA32 주요 레지스터 세트

EFLAGS 레지스터

- EFLAGS 레지스터는 명령이 실행되는 과정 중 현제 프로세서의 상태정보, 제어정보 등을 보관하고 있으며, 각 비트가 별도의 의미를 가지고 있습니다.

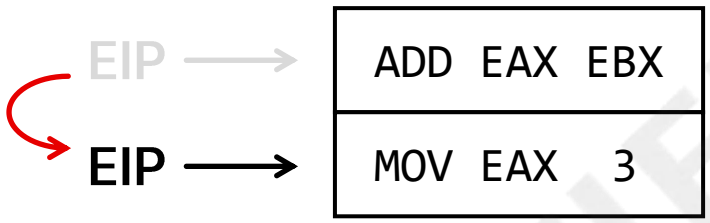
[EFLAGS register]

Bit #	Abbreviation	Description	Category
0	CF	Carry flag	Status
1		Reserved, always 1 in EFLAGS	
2	PF	Parity flag	Status
3		Reserved	
4	AF	Adjust flag	Status
5		Reserved	
6	ZF	Zero flag	Status
7	SF	Sign flag	Status
8	TF	Trap flag (single step)	Control
9	IF	Interrupt enable flag	Control
10	DF	Direction flag	Control
11	OF	Overflow flag	Status
12 - 13	IOPL	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15		Reserved, always 1 on 8086 and 186, always 0 on later models	
16	RF	Resume flag (386+ only)	Sytem
17	VM	Virtual 8086 mode flag (386+ only)	System
18	AC	Alignment check (486SX+ only)	System
19	VIF	Virtual interrupt flag (Pentium+)	System
20	VIP	Virtual interrupt pending (Pentium+)	System
21	ID	Able to use CPUID instruction (Pentium+)	System
22 - 31		Reserved	

IA32 주요 레지스터 세트

EIP 레지스터

- EIP 레지스터는 "Instruction pointer" 혹은 "Program counter"로 불리우기도 하며, 다음에 실행할 명령의 주소 값을 가지고 있습니다.



DR 레지스터 (Debug Register)

- DR 레지스터는 디버깅과 관련하여 주로 브레이크 포인트 지점과 관련된 정보를 담고 있습니다.

IA32 주요 명령어

MOV

- Source 피연산자의 값을 Destination 피연산자로 복사합니다.
- 두 연산자의 사이즈는 반드시 동일해야 합니다.

```
.text:00401023                  mov     DWORD [Address], 0Ah
.text:00401027                  mov     eax, DWORD [Address]
.text:0040102A                  mov     ecx, eax
```

MOZX

- Source 피연산자의 값을 Destination 피연산자로 복사한 후 나머지 비트를 0으로 채웁니다.
- Destination 피연산자는 레지스터만 사용할 수 있습니다.

```
.text:00401016                  mov     DWORD [Address], 11223344h
.text:0040101D                  movzx   eax, WORD [Address]
```

MOZSX

- Source 피연산자의 값을 Destination 피연산자로 복사한 후 나머지 비트를 Source 피연산자의 부호비트로 채웁니다.

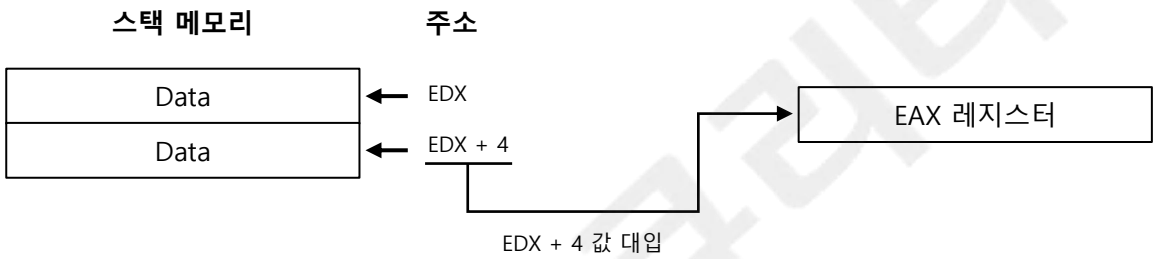
```
.text:00401016                  mov     DWORD [Address], 8877h
.text:0040101D                  movsx   eax, WORD [Address]
```

IA32 주요 명령어

LEA

- Source 피연산자를 참조하여 계산된 메모리 주소가 Destination 피연산자(범용 레지스터)에 저장됩니다.

```
.text:00401027                  lea     eax, [edx+4]
```



ADD

- Source 피연산자를 동일한 크기의 Destination 피연산자에 더합니다.
- Source 피연산자는 변하지 않으며 덧셈한 결과 값은 Destination 피연산자에 저장합니다.

```
.text:00401030                  mov     eax, 0x3
.text:00401034                  add     eax, 0x8
```

SUB

- Destination 피연산자를 Source 피연산자만큼 감소 시킵니다.
- CPU의 ALU 내부에서는 실제 덧셈 연산이 수행됩니다. (Source 피연산의 값에 2의 보수를 취한 후, 두 값을 더함)

IA32 주요 명령어

CALL 명령와 JMP 명령 비교

- IA32에서 EIP 레지스터의 경우 프로그램 흐름에 직접적인 영향을 미치는 레지스터이므로 임의로 값을 수정할 수 없으며 몇 가지 명령을(e.g. JMP, CALL, RET) 이용하여 간접적으로 수정할 수 있습니다.

	JMP	CALL
공통점	피연산자로 지정된 주소로 프로그램의 실행 흐름을 변경시킴 (분기함)	
차이점	분기하기 직전 현재 EIP의 값을 스택에 백업하지 않음	분기하기 직전 현재 EIP의 값을 스택에 백업함
	JZ, JNZ, JA, JB 등의 명령을 통해 조건 분기 가능	조건 분기 불가능

RET

- 의미상 "POP EIP" 명령과 동일합니다.
- 물론 "POP EIP" 명령은 EIP 레지스터 값을 직접 변경하기 때문에 사용할 수 없습니다.

PUSH, POP

- PUSH 는 피연산자를 스택 메모리의 최상단에 입력한 후 스택포인터(ESP)를 변화 시킵니다.
- POP은 스택 메모리의 최상단에 있는 값을 피연산자에 입력한 후 스택포인터(ESP)를 변화 시킵니다.

```
.text:004015CD                  push    ebp
.text:004015CE                  pop     eax
```

IA32 주요 명령어

MUL

- 부호없는 정수에 대한 곱셈을 수행하는 명령입니다.
- AL, AX, EAX 레지스터에 담겨있는 값에 8,16,32비트 값을 곱하는 명령입니다.
- 연산 결과는 AX, DX:AX 혹은 EDX:EAX에 저장합니다.

XOR

- 두 개의 피연산자에서 매칭되는 비트들을 XOR 연산 수행 후 그 결과를 Destination 피연산자에 저장합니다.
- 동일한 레지스터가 두 개의 피연산자로 사용될 경우 레지스터의 값을 0으로 초기화 하는 의미로 사용됩니다.

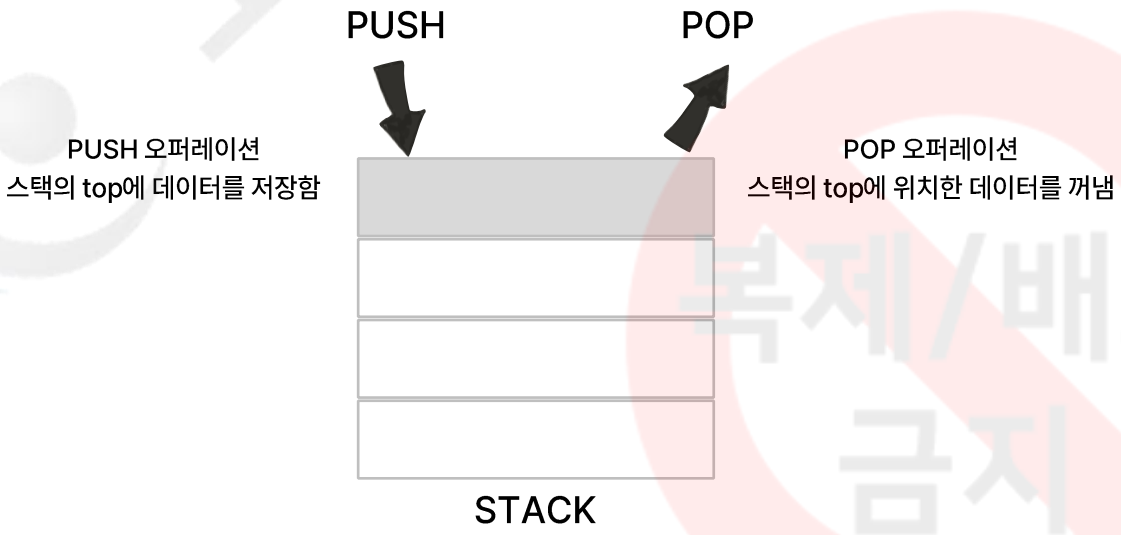
스택 메모리 개요

스택 메모리

- IA32 메모리의 프로세스가 사용하는 가상주소공간(VAS)에서 대표적으로 많이 사용되는 버퍼(Buffer) 메모리로 힙(Heap)과 스택(Stack)이 있습니다.
- 스택에는 함수에게 전달된 파라미터, 지역변수, 반환주소 등이 보관됩니다.
- IA32 기반 스택은 탑 포인터가 마지막으로 저장된 데이터를 가리키며, 낮은 주소 방향으로 자라는 'Full Decending Stack' 방식으로 운용됩니다.

스택 자료구조

- 기본적으로 스택은 2개의 오퍼레이션(PUSH, POP)와 한 개의 탑 포인터(ESP)로 운용됩니다.
- PUSH 오퍼레이션은 스택의 최상층에 데이터를 저장한 후 탑 포인터(ESP) 값을 변화시킵니다.
- POP 오퍼레이션은 스택의 최상층에 있는 데이터를 추출합니다. (보통은 오퍼랜드인 레지스터에 주입)



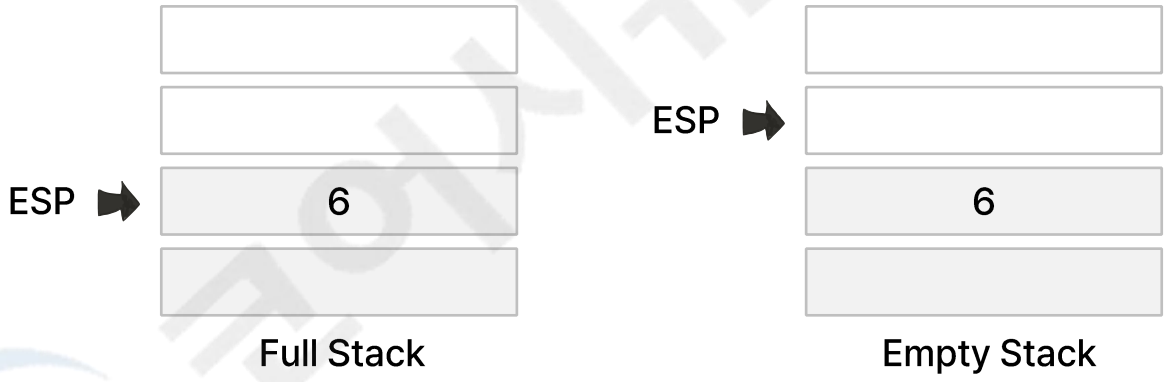
스택 메모리 개요

Call Stack

- IA32 에서 사용되는 스택 버퍼는 콜스택(Call Stack) 이라고 부르며, 콜스택이란 컴퓨터 프로그램에서 현재 실행중인 서브루틴에 관한 정보를 저장하는 스택 자료구조를 의미합니다.
- 콜스택은 스택 자료구조와 비슷한 개념을 갖지만, 스택 자료구조와 다르게 중간에 있는 데이터를 바로 사용할 수도 있다는 점에서 차이점을 갖습니다.

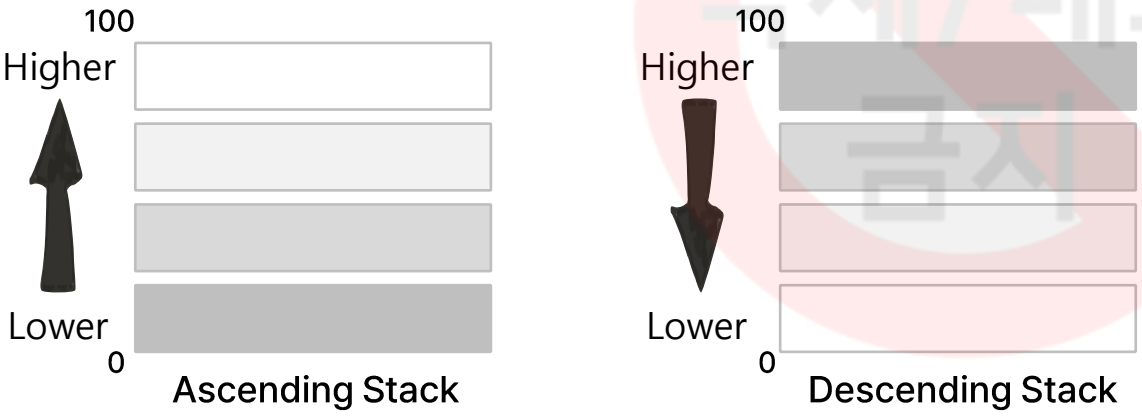
Full Stack과 Empty Stack

- PUSH 명령어를 통해 마지막에 들어온 데이터를 TOP 포인터가 가리킬 경우 Full Stack이라 합니다.
- 이와 달리, 마지막에 들어온 데이터를 TOP 포인터가 가리키지 않고 그 다음 위치를 가리킬 경우 Empty Stack이라 합니다.



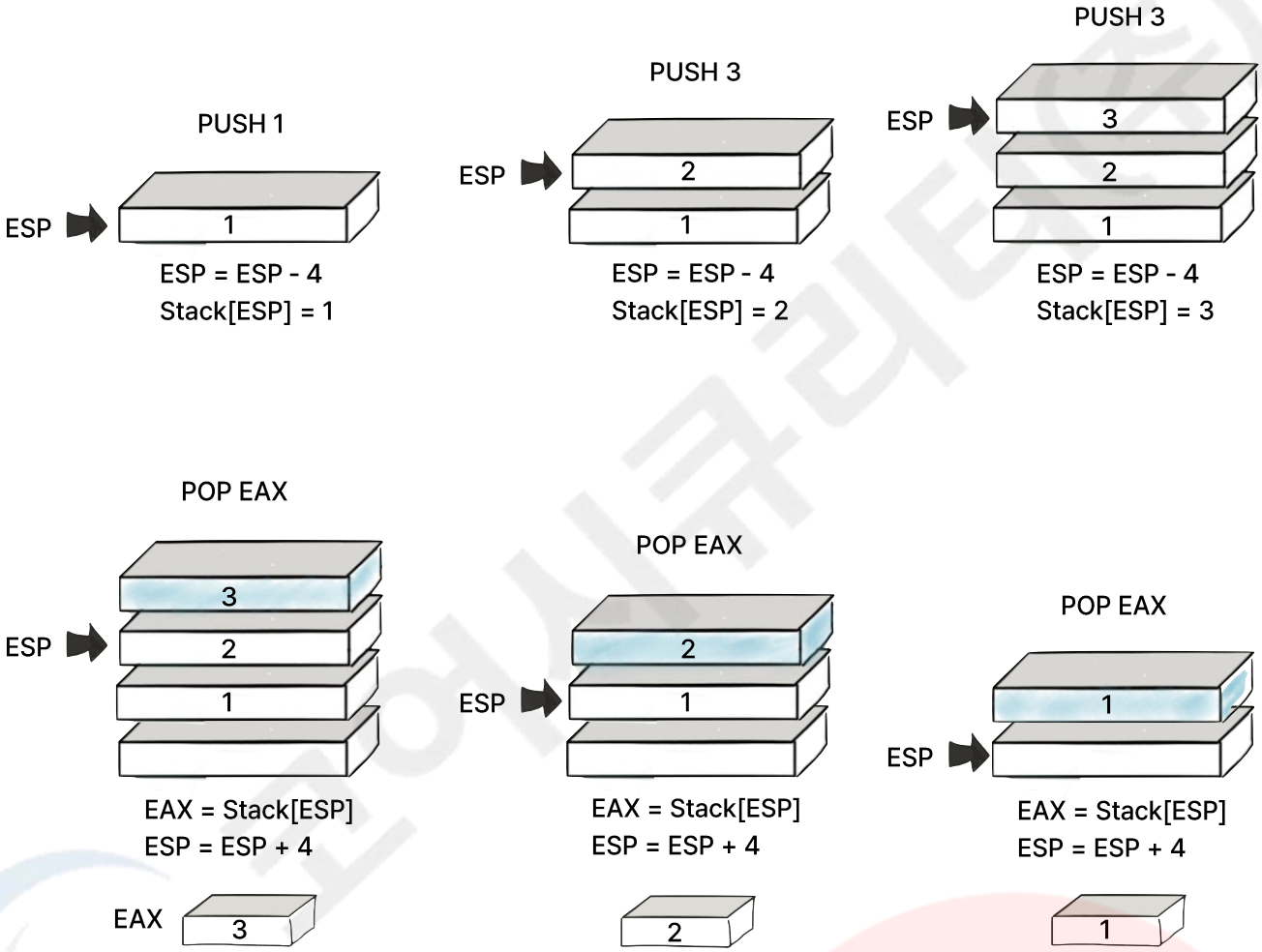
Ascending Stack과 Descending Stack

- 메모리에 데이터가 쌓일 때, 주소가 증가하는 방향으로 쌓일 경우 Ascending Stack이라 합니다.
- 이와 달리, 주소가 감소하는 방향으로 쌓일 경우 Descending Stack이라 합니다.



스택 메모리 개요

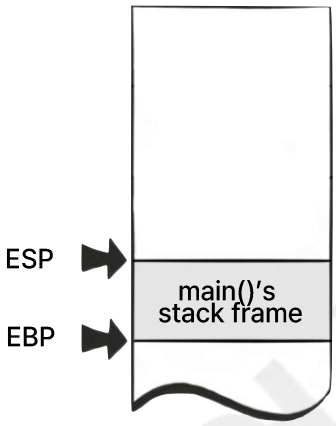
IA32 스택 오퍼레이션



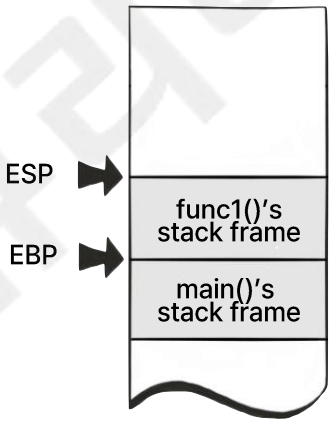
IA32 스택 프레임

스택 프레임

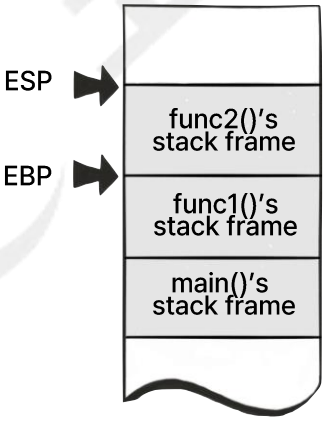
- 함수가 사용하는 스택 영역을 스택 프레임이라고 합니다.
- EBP(Base Pointer)는 현재 실행 중인 함수의 스택 프레임 시작 주소(base)를 가리킵니다.
- ESP(Stack Pointer)는 현재 실행 중인 함수의 스택 프레임 끝 주소(top)를 가리킵니다.
- 함수 실행 중 Base Pointer는 고정되어 있으며, Stack Pointer는 상황에 따라 변합니다.



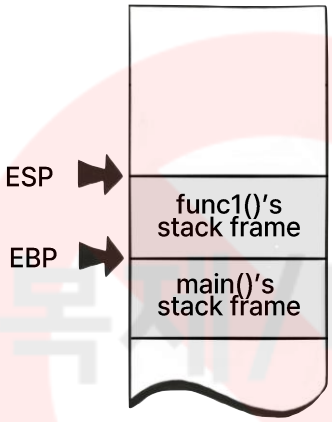
main() 함수 호출



main()함수에서
func1()호출



func()1에서
func2() 호출

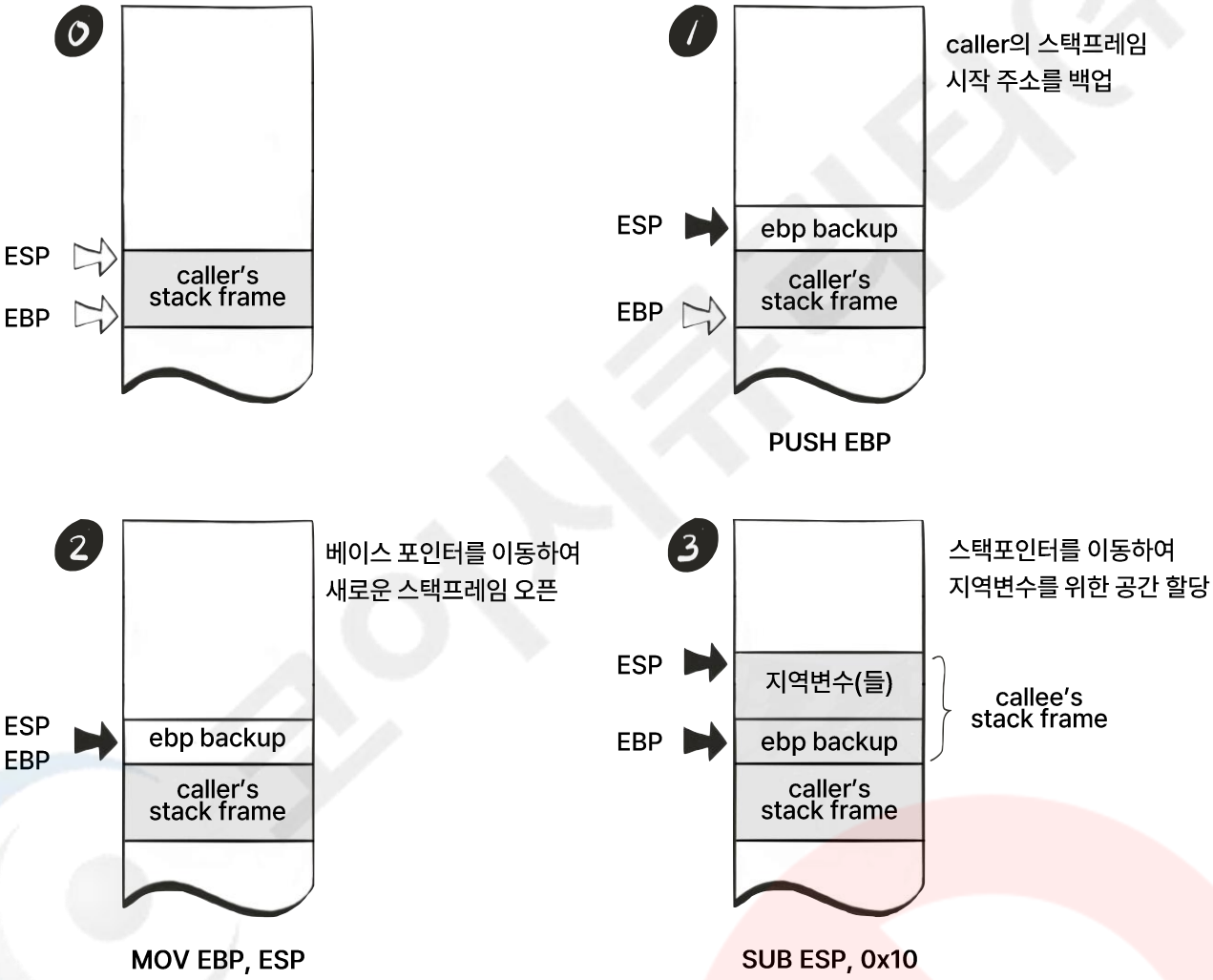


func2() 종료

IA32 스택 프레임

스택 프레임의 생성

- 스택 프레임은 함수가 실행된 직후 함수에 의해 생성됩니다.
- 함수 시작 직후 스택 프레임을 오픈하는 코드를 함수 프롤로그라고 합니다.

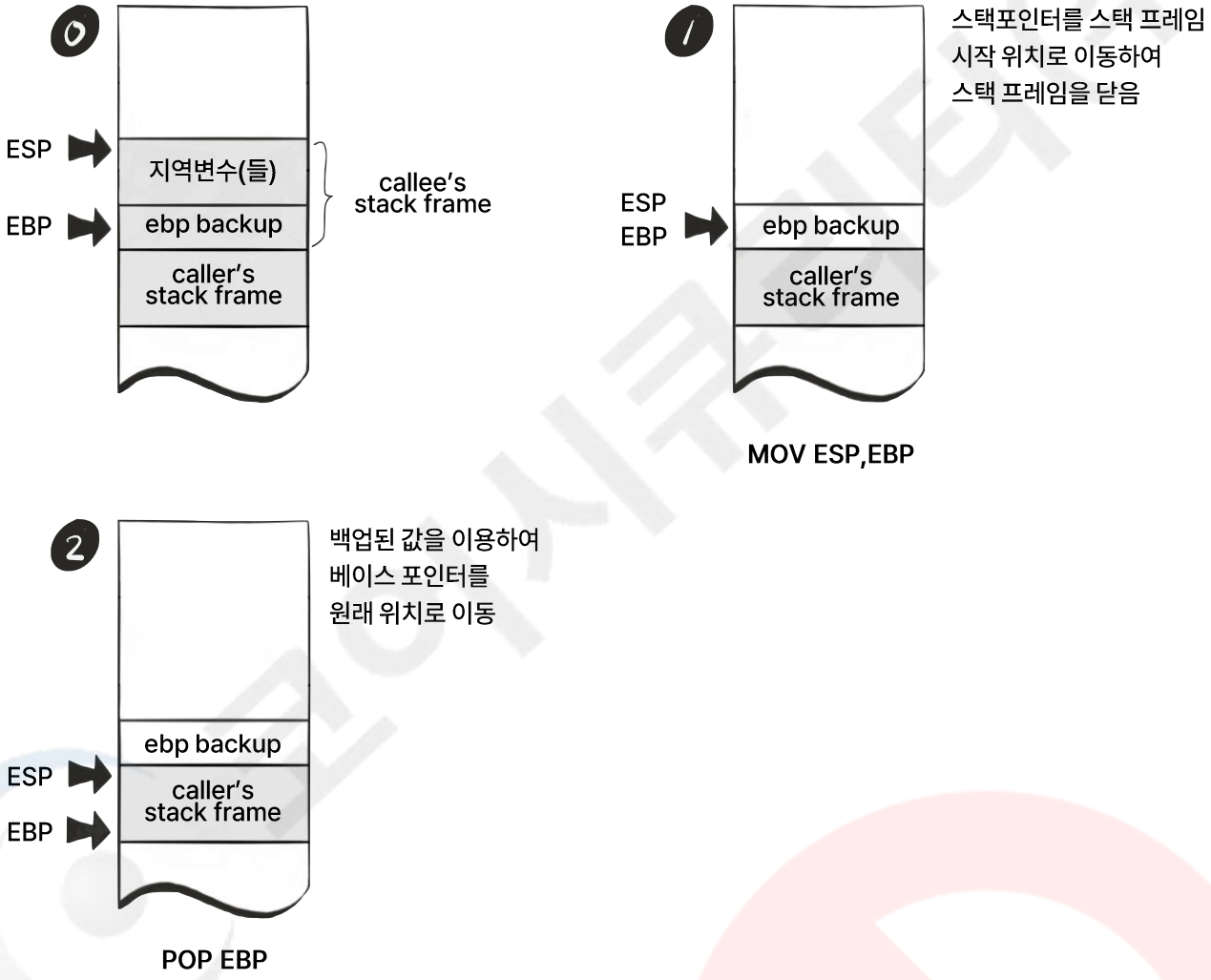



```
.text:0040101E _main    proc near
.text:0040101E
.text:0040101E    push    ebp
.text:0040101F    mov     ebp, esp
.text:00401021    sub     esp, n

              (생략)

.text:00401027    call    _myfunc
---------------------------------
.text:00401000 _myfunc  proc near
.text:00401000    push    ebp
.text:00401001    mov     ebp, esp
```

IA32 스택 프레임

스택 프레임의 소멸

- 스택 프레임은 함수가 종료되기 직전 함수에 의해 소멸됩니다.
- 함수 종료 직전 스택 프레임을 클로징하는 코드를 함수 에필로그라고 합니다.




```
.text:00401000 _myfunc   proc near
.text:00401000
.text:00401000    push    ebp
.text:00401001    mov     ebp, esp
.text:00401003    push    ecx

               (중간 생략)

.text:00401020    call    _printf
.text:00401025    add     esp, 14h
.text:00401028    mov     esp, ebp
.text:0040102A    pop     ebp
.text:0040102B    retn
.text:0040102B _myfunc    endp
```

IA32 스택 프레임

컴파일러

- ‘Stack Frame Pointer Omission’은 EBP 레지스터를 스택의 프레임 포인터로 사용하지 않도록 하는 컴파일 방법 입니다.
- 스택 프레임을 셋업하고 제거하는 코드가 없고, EBP 레지스터를 다른 용도로 사용할 수 있기 때문에 프로그램의 성능 향상에 도움을 줍니다.
- 대부분의 컴파일러에서 최적화 옵션 형태로 제공합니다.
- Visual Studio (/O2, /FPO), GCC(--fomit-frame-pointer)
- 컴파일러는 런타임에 사용되는 주소들을 미리 계산하야 하므로 컴파일 단계에서 부담이 커집니다.
- 스택 프레임 개념이 없어지므로(지역변수와 매개변수 모두 "ESP+N"형태로 표현됨) 분석자 입장에서 디버깅이 번거로워질 수 있습니다.