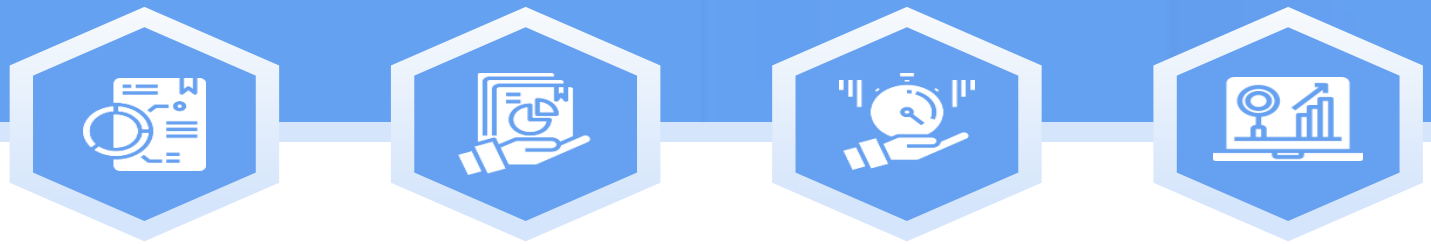
시스템 보안

#4 80x86 시스템 - 2



KOREA UNIVERSITY
SEJONG CAMPUS

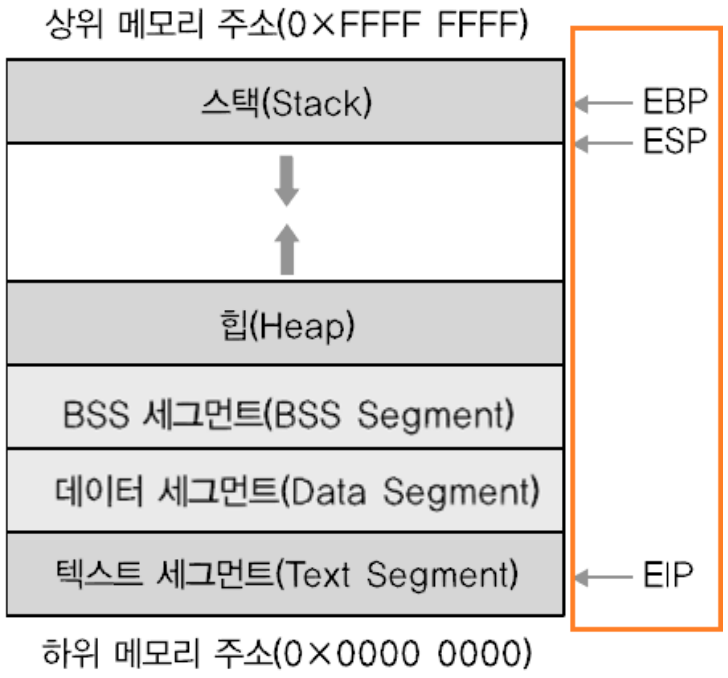
- 목차

- x86 시스템 메모리의 구조와 동작
- 프로세스(Process)와 스레드(Thread)
- DLL(Dynamic Linked Library)

x86 시스템 메모리의 구조와 동작

x86 시스템 메모리의 구조와 동작

- **메모리의 기본 구조 – Linux**



- 스택(Stack)

  함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역

- ESP(Extended Stack Pointer)

  스택 프레임의 끝 지점 주소(스택의 가장 아랫부분, 스택의 마지막)가 저장되며,

  PUSH, POP 명령에 따라 ESP의 값이 4바이트씩 변함

- EBP(Extended Base Pointer)

  스택 프레임의 시작 지점 주소(스택의 가장 윗부분, 스택의 처음)가 저장되며,

  EBP 값은 현재 사용 중인 스택 프레임이 소멸되지 않는 이상 변하지 않음

- 텍스트 세그먼트(Text Segment)

  실행할 프로그램의 코드가 저장되는 영역으로 텍스트(code) 영역

- EIP(Extended Instruction Pointer)

  CPU로 실행되는 머신 코드가 있는 영역으로, EIP가 다음에 실행하는 명령의 위치를 나타냄

x86 시스템 메모리의 구조와 동작

- **메모리의 기본 구조 – Linux**

  - 데이터 세그먼트(Data Segment)
    초기화된 외부 변수나 static 변수 등이 저장되는 영역, 보통 텍스트 세그먼트와 데이터 세그먼트 영역을 합쳐 프로그램이라 함

    데이터 세그먼트(Data Segment)

    ```
    static int  a = 1;
    ```

  - BSS 세그먼트(Block Started by Symbol Segment)
    초기화 되지 않은 데이터 세그먼트(Uninitialized data segment)로 불리며, 외부 변수나 static 변수 중 초기화 되지 않은 변수들이 정의될 때 저장되며 프로그램이 실행될 때 0이나 NULL 포인터로 초기화

    BSS 세그먼트(BSS Segment)

    ```
    static int a;
    ```

  - 힙(Heap)
    Heap은 동적으로 생성되는 data에 대하여 할당을 해주는 메모리 영역으로 bss를 기준으로 stack과의 사이에 위치

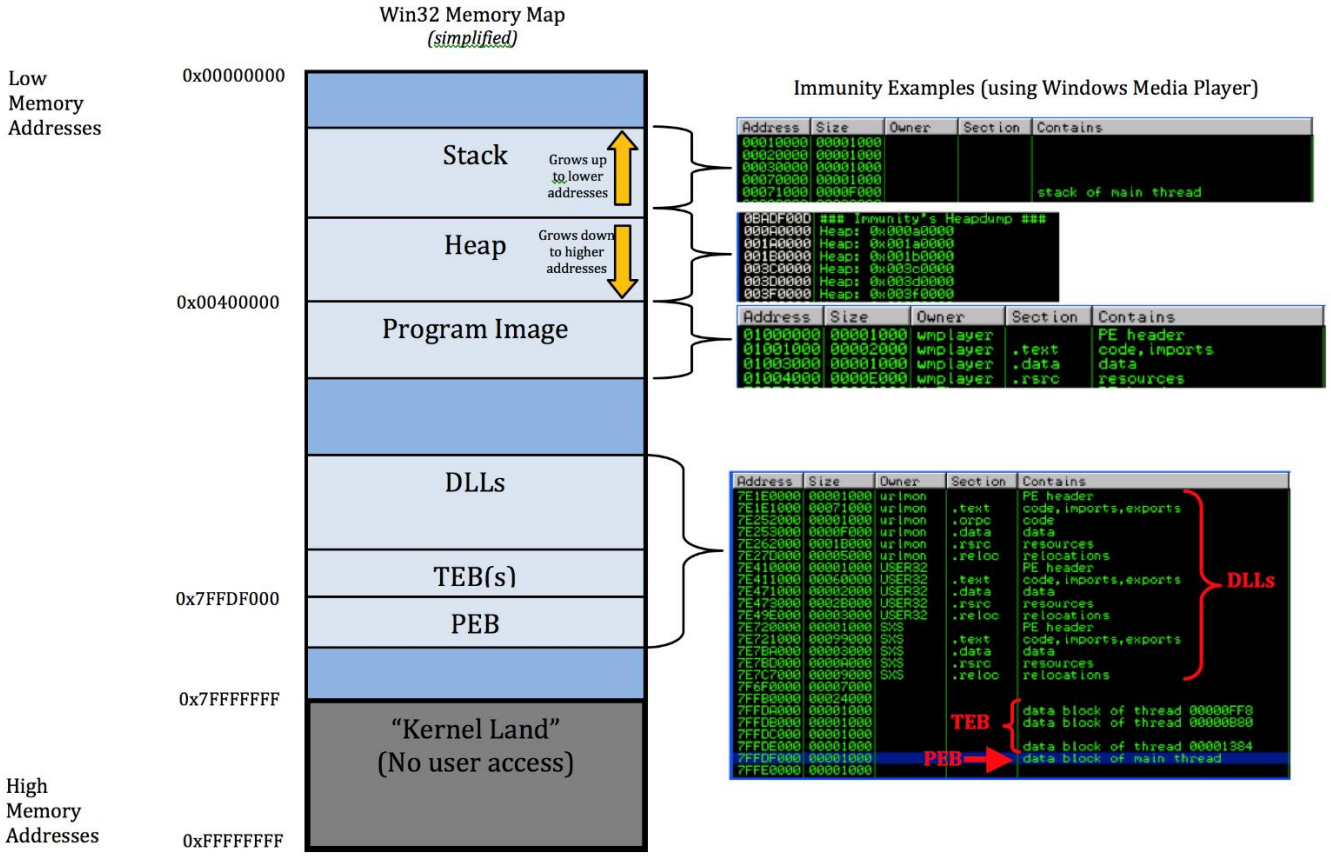
    힙(Heap)

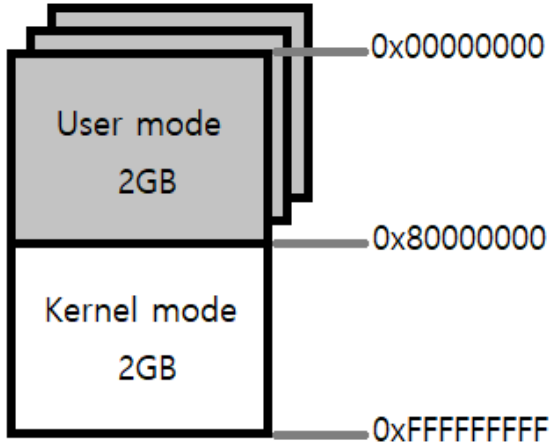
    ```
    malloc(0x80);
    ```

x86 시스템 메모리의 구조와 동작

- ## 메모리의 기본 구조 – Windows

  - Win32 프로세스의 메모리 구조는 리눅스의 구조와 다르며, 운영체제의 종류나 버전에 따라서 달라지는 부분이 많음
  - 리눅스의 경우 커널 영역의 크기가 1G지만, 윈도우는 2G(boot.ini에 지정하는 /3GB 같은 옵션에 의해 사용자 3G, 커널 1G로 변경 가능)

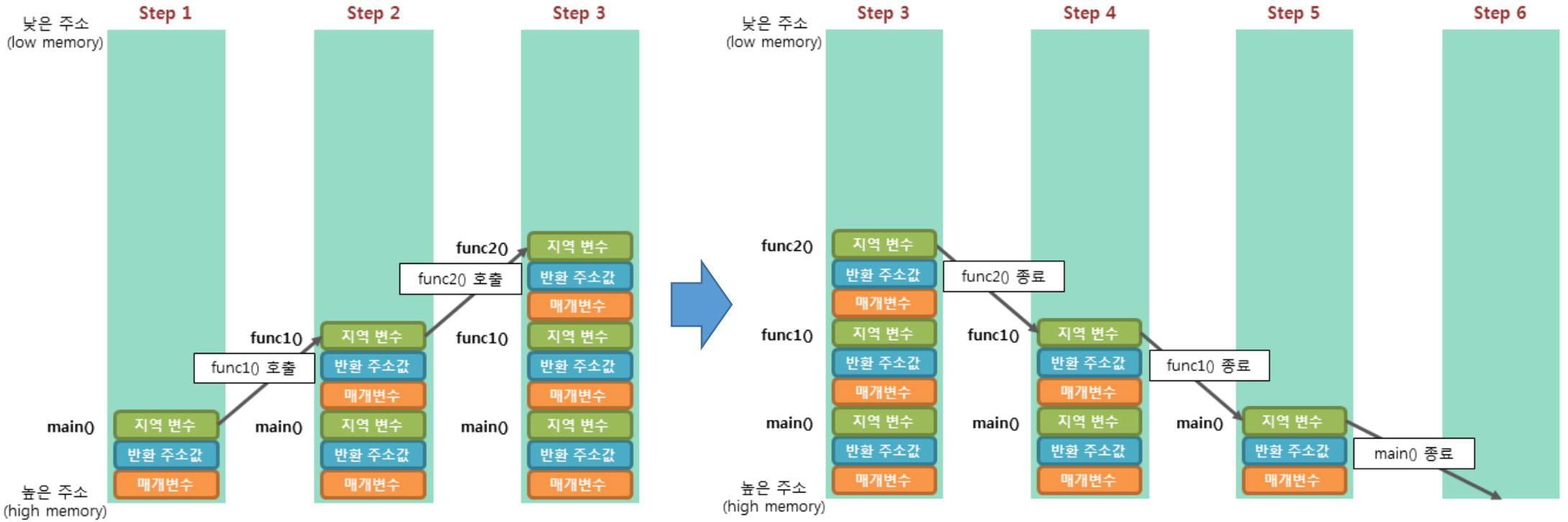




※ 커널 영역은 모든 프로세스가 공유

x86 시스템 메모리의 구조와 동작

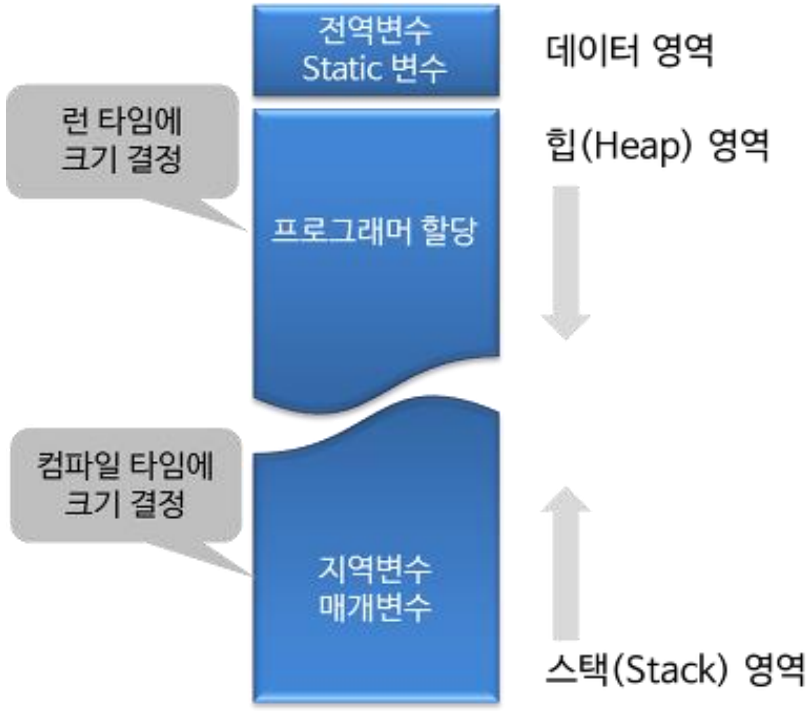
- **메모리의 기본 구조 – 스택(Stack)**
  - 후입선출(LIFO : Last-In, First Out) 방식에 의해 정보를 관리되며 스택의 끝부분에서 데이터의 삽입과 삭제가 발생
  - 프로그램이 실행하고 있는 동안에 만들어지는 데이터(지역변수, 매개변수 등)을 스택처럼 쌓아서 저장하는 영역
  - 함수가 호출될 때마다 스택의 공간이 일정부분 할당되며, 이를 스택 프레임(Stack Frame)이라고 함



x86 시스템 메모리의 구조와 동작

- **메모리의 기본 구조 – 힙(Heap)**
  - 프로그램의 실행 중 필요한 메모리를 할당하기 위해 운영체제에 예약되어 있는 영역
  - 데이터를 저장하기 위해 메모리를 요청하면 운영체제는 힙에 있는 메모리를 프로그램에 할당
  - 메모리가 더 이상 필요 없으면 할당 받았던 메모리를 운영체제에 반납, 운영체제에서는 반납된 메모리를 다시 힙에 돌려줌
  - 힙에 대한 메모리는 포인터를 통해 동적으로 할당되거나 반환되며, 연결 리스트, 트리, 그래프처럼 동적인 특성이 있는 데이터 구조에서 널리 사용

```c
int main(){

    int i = 0;
    scanf("%d", &i);

    int *arr = (int*)malloc(sizeof(int) * i);
    // 데이터가 힙영역에 할당됨.

    free(arr);
    // 데이터가 해제됨.

    return 0;
}
```



x86 시스템 메모리의 구조와 동작

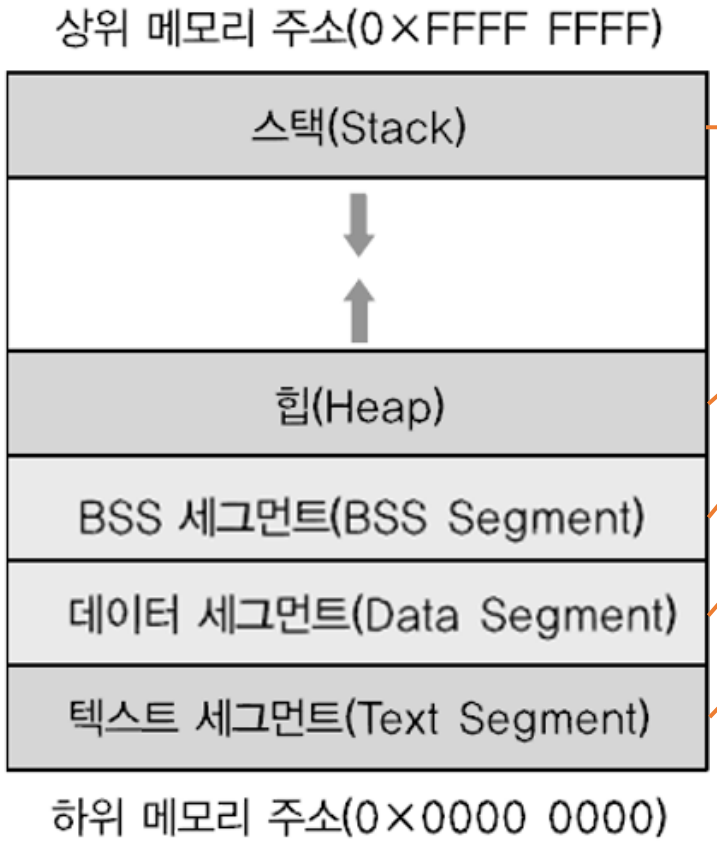
- **(실습) 메모리 구조 확인해보기 (Linux)**

```c
#include <stdio.h>
#include <stdlib.h>

// 초기화된 정적 변수
static int static_init = 70;

// 초기화 되지 않은 정적 변수
static int static_uninit;

// 초기화된 전역 변수
int initial = 30;

// 초기화되지 않은 전역 변수
int uninitial;

// 함수
int function()
{
    return 20;
}
```

```c
int main(int argc, const char* argv[])
{
    // 지역변수
    int localval = 30;

    // 동적 할당 변수
    char* arr = (char*)malloc(sizeof(char) * 10);

    /* 포인터 출력 영역 */
    printf("\n");
    printf("Local Variable                : %p \n", &localval);
    printf("Dynamic Variable              : %p \n", arr);
    printf("Uninitialized Static Variable : %p \n", &static_uninit);
    printf("Uninitialized Global Variable : %p \n", &uninitial);
    printf("Initialized Global Variable   : %p \n", &initial);
    printf("Initialized Static Variable   : %p \n", &static_init);
    printf("Function Address              : %p \n", function);
    printf("\n");

    return 0;
}
```

x86 시스템 메모리의 구조와 동작

- **(실습) 메모리 구조 확인해보기(리눅스)**
  - 실제 각각의 변수 주소를 비교하여 리눅스의 메모리 구조와 같은 지 확인




```
Local Variable                : 0xff97a384
Dynamic Variable              : 0x589f71a0
Uninitialized Static Variable : 0x57b53018
Uninitialized Global Variable : 0x57b53014
Initialized Global Variable   : 0x57b5300c
Initialized Static Variable   : 0x57b53008
Function Address              : 0x57b501bd
```

- 스택 : 함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역
- 힙 : 동적으로 생성되는 데이터에 대하여 할당을 해주는 메모리 영역

- BSS 세그먼트 : 외부 변수나 정적 변수 중 초기화 되지 않은 변수들이 정의될 때 저장
- 데이터 세그먼트 : 외부 변수나 정적 변수 중 초기화 된 변수들이 정의될 때 저장
- 텍스트 세그먼트 : 실행할 프로그램의 코드가 저장되는 코드 영역

x86 시스템 메모리의 구조와 동작

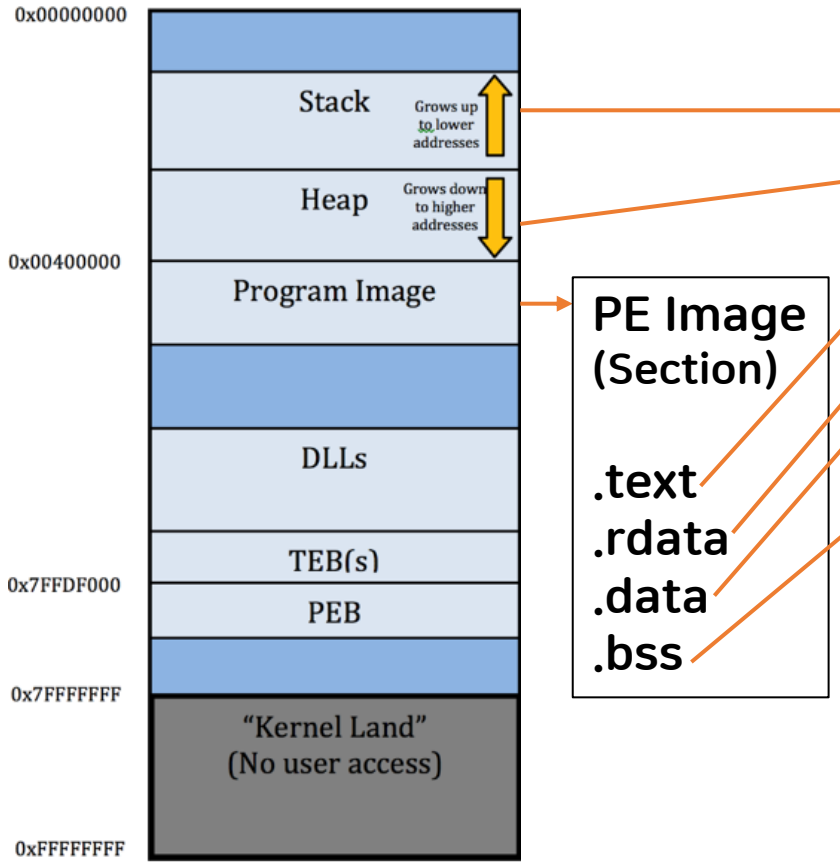
- **(실습) 메모리 구조 확인해보기 (Windows)**

```c
#include <stdio.h>
#include <stdlib.h>

// 상수
const int const_val = 0;

// 초기화된 정적 변수
static int static_init = 70;

// 초기화 되지 않은 정적 변수
static int static_uninit;

// 초기화된 전역 변수
int initial = 30;

// 초기화되지 않은 전역 변수
int uninitial;

// 함수
int function()
{
    return 20;
}
```

```c
int main(int argc, const char* argv[])
{
    // 지역변수
    int localval = 30;

    // 동적 할당 변수
    char* arr = (char*)malloc(sizeof(char) * 10);

    /* 포인터 출력 영역 */
    printf("\n");
    printf("Local Variable                : %p \n", &localval);
    printf("Dynamic Variable              : %p \n", arr);
    printf("Function Address              : %p \n", function);
    printf("Initialized Static Variable   : %p \n", &static_init);
    printf("Initialized Global Variable   : %p \n", &initial);
    printf("Uninitialized Global Variable : %p \n", &uninitial);
    printf("Uninitialized Static Variable : %p \n", &static_uninit);
    printf("Constant Variable             : %p \n", &const_val);
    printf("\n");

    return 0;
}
```

x86 시스템 메모리의 구조와 동작

- (실습) 메모리 구조 확인해보기(윈도우)




```
Local Variable                 : 008FFA94
Dynamic Variable               : 00B2A2B8
Function Address               : 0036100A
Constant Variable              : 00367B5C
Initialized Global Variable    : 0036A000
Initialized Static Variable    : 0036A004
Uninitialized Static Variable  : 0036A1E0
Uninitialized Global Variable  : 0036A514
```

- 스택 : 함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역
- 힙 : 동적으로 생성되는 데이터에 대하여 할당을 해주는 메모리 영역

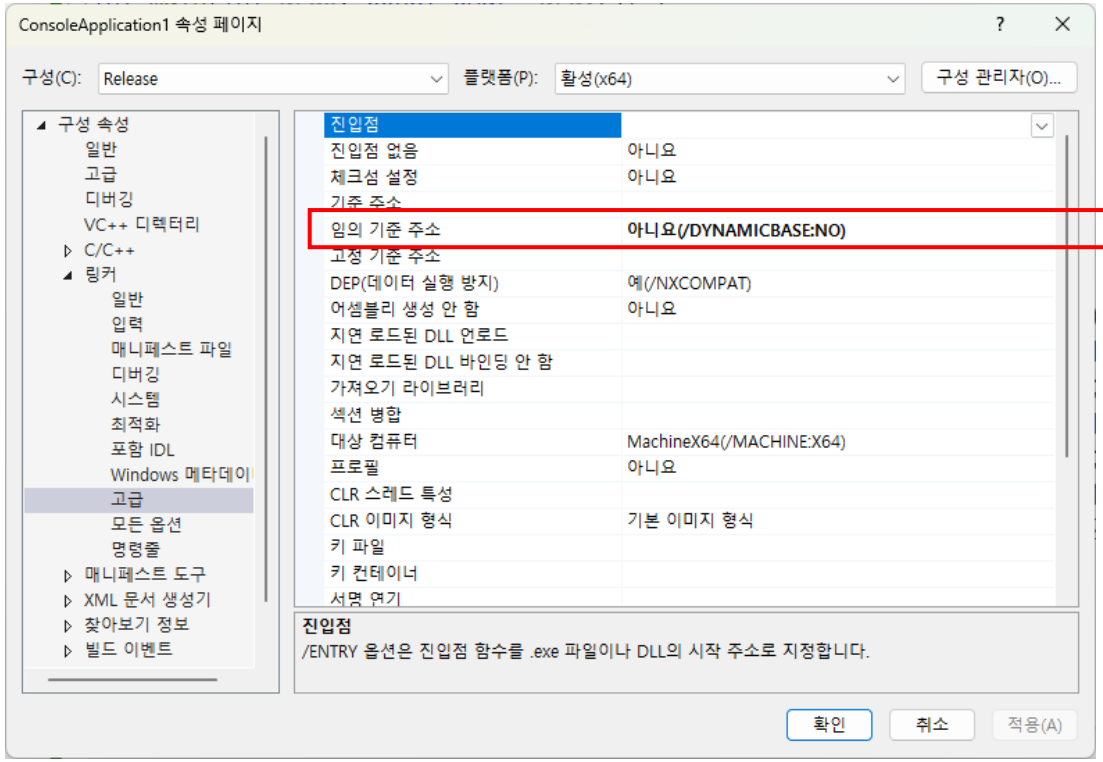
- .text : 실행, 읽기 속성을 지니며 프로그램 실행 코드를 담고 있는 섹션
- .rdata : 읽기 전용 데이터 섹션으로 상수형 변수, 문자열 상수 등이 위치
- .data : 읽고 쓰기가 가능한 데이터 섹션으로 초기화된 전역변수와 정적 변수 등이 위치
- .bss : 초기화되지 않은 전역변수를 담고 있는 섹션
  (컴파일러에 따라서 .data 섹션에 포함될 수 있음)

x86 시스템 메모리의 구조와 동작

- **(실습) 메모리 구조 확인해보기 (윈도우 ASLR)**

ASLR(Address Space Layout Randomization) : 라이브러리, 힙, 스택 영역 등의 주소를 실행될 때마다 랜덤하게 바꿔서 RTL(Return to Libc)과 같이 정해진 주소를 이용한 공격을 막는 보호 기법

※ 리눅스는 커널 설정을 변경해야 함



ASLR off

x86 시스템 메모리의 구조와 동작

- (실습) 메모리 구조 확인해보기 (윈도우 ASLR)

```
c:\Temp>ViewMemory.exe

  Constant Variable Memory Address      : 00007FF771E532F0
  Uninitialized Variable Memory Address : 00007FF771E5ED88
  Initialized Variable Memory Address   : 00007FF771E5D000
  Static Variable Memory Address        : 00007FF771E5D004
  Function Memory Address               : 00007FF771E41000
  Local Variable 1 Memory Address       : 000000C92C12F930
  Local Variable 2 Memory Address       : 000000C92C12F934
  Dynamic Variable Memory Address       : 0000024736B7AE90


c:\Temp>ViewMemory.exe

  Constant Variable Memory Address      : 00007FF771E532F0
  Uninitialized Variable Memory Address : 00007FF771E5ED88
  Initialized Variable Memory Address   : 00007FF771E5D000
  Static Variable Memory Address        : 00007FF771E5D004
  Function Memory Address               : 00007FF771E41000
  Local Variable 1 Memory Address       : 0000004C682FFB00
  Local Variable 2 Memory Address       : 0000004C682FFB04
  Dynamic Variable Memory Address       : 000001FFEBC2B010
```

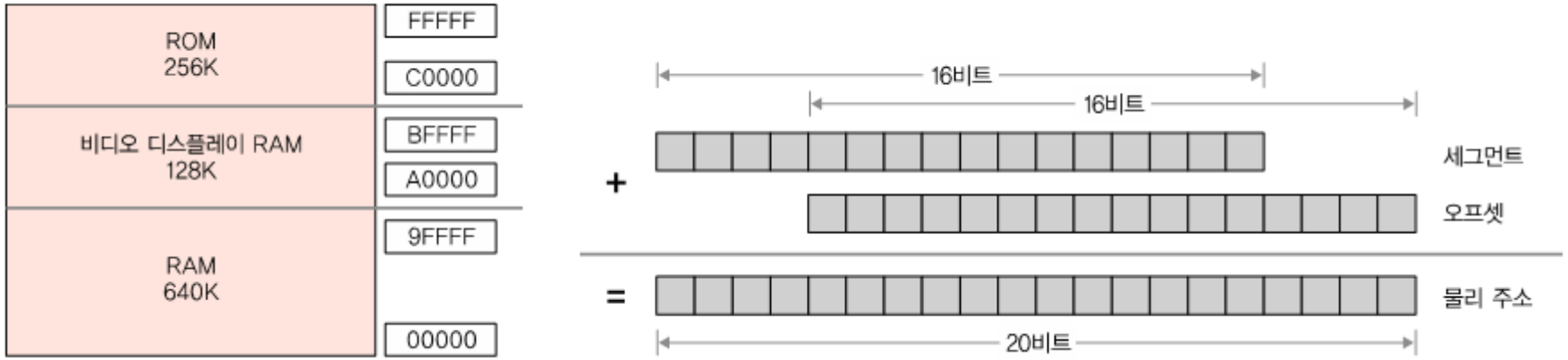
/DYNAMICBASE
(ASLR 사용)

```
c:\Temp>ViewMemory.exe

  Constant Variable Memory Address      : 00000001400132F0
  Uninitialized Variable Memory Address : 000000014001ED88
  Initialized Variable Memory Address   : 000000014001D000
  Static Variable Memory Address        : 000000014001D004
  Function Memory Address               : 0000000140001000
  Local Variable 1 Memory Address       : 000000000014FED0
  Local Variable 2 Memory Address       : 000000000014FED4
  Dynamic Variable Memory Address       : 00000000004FB070


c:\Temp>ViewMemory.exe

  Constant Variable Memory Address      : 00000001400132F0
  Uninitialized Variable Memory Address : 000000014001ED88
  Initialized Variable Memory Address   : 000000014001D000
  Static Variable Memory Address        : 000000014001D004
  Function Memory Address               : 0000000140001000
  Local Variable 1 Memory Address       : 000000000014FED0
  Local Variable 2 Memory Address       : 000000000014FED4
  Dynamic Variable Memory Address       : 000000000041B0B0
```

/DYNAMICBASE:NO
(ASLR 미사용)

x86 시스템 메모리의 구조와 동작

- **메모리 접근 모드와 동작 - 리얼 모드(Real Mode)**
  - 8086 CPU(16bit)에서 사용되던 동작 모드로 16bit 레지스터를 이용하여 20비트 주소로 확장
  - 총 1MB(220 =1,048,567)의 메모리 사용 가능
  - 20비트 주소를 나타내기 위해 세그먼트 레지스터를 도입
  - 16비트의 세그먼트 레지스터와 16비트의 오프셋을 중첩 시켜 20비트의 물리 주소를 생성



초기 8086 시스템 메모리 구조와 20비트 메모리 주소 구성 방법

x86 시스템 메모리의 구조와 동작

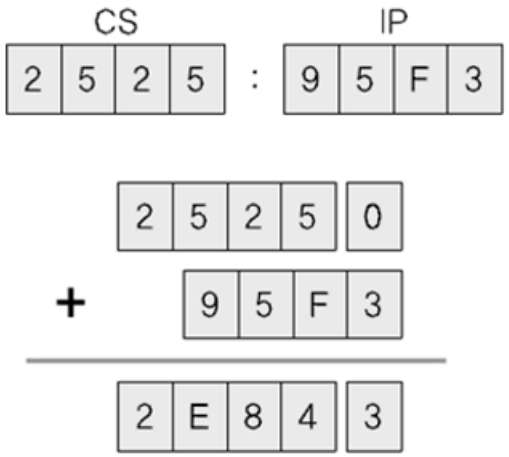
- ## 메모리 접근 모드와 동작 - 리얼 모드(Real Mode)

  - 리얼 모드(Real Mode)에서의 20비트 메모리 주소 구성 방법

    세그먼트 주소인 CS 레지스터가 0x2525h, 오프셋인 IP가 0x95F3h라면

    CS 값 0x2525h뒤에 한 자리의 0x0h를 붙인 다음(4bit 왼쪽으로 shift, x 16, x 10h) IP 값 95F3h를 더한 2E843h가 실제 가리키는 물리 주소

    이를 2525h:95F3h, 또는 [CS]:95F3h로 표현 함

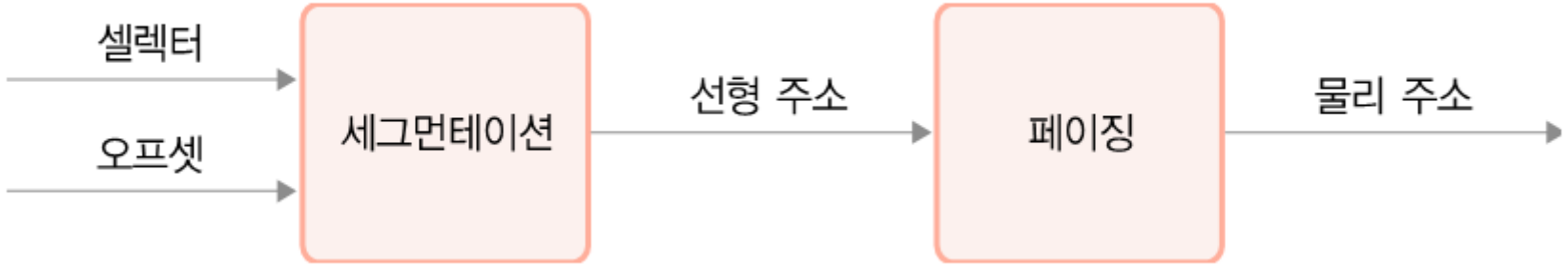


세그먼트 레지스터	오프셋 레지스터
CS	IP
DS	SI, DI, BX
SS	SP, BP
ES	SI, DI, BX

20비트 메모리 주소 구성 예와 세그먼트 레지스터 별 기본 오프셋 레지스터

x86 시스템 메모리의 구조와 동작

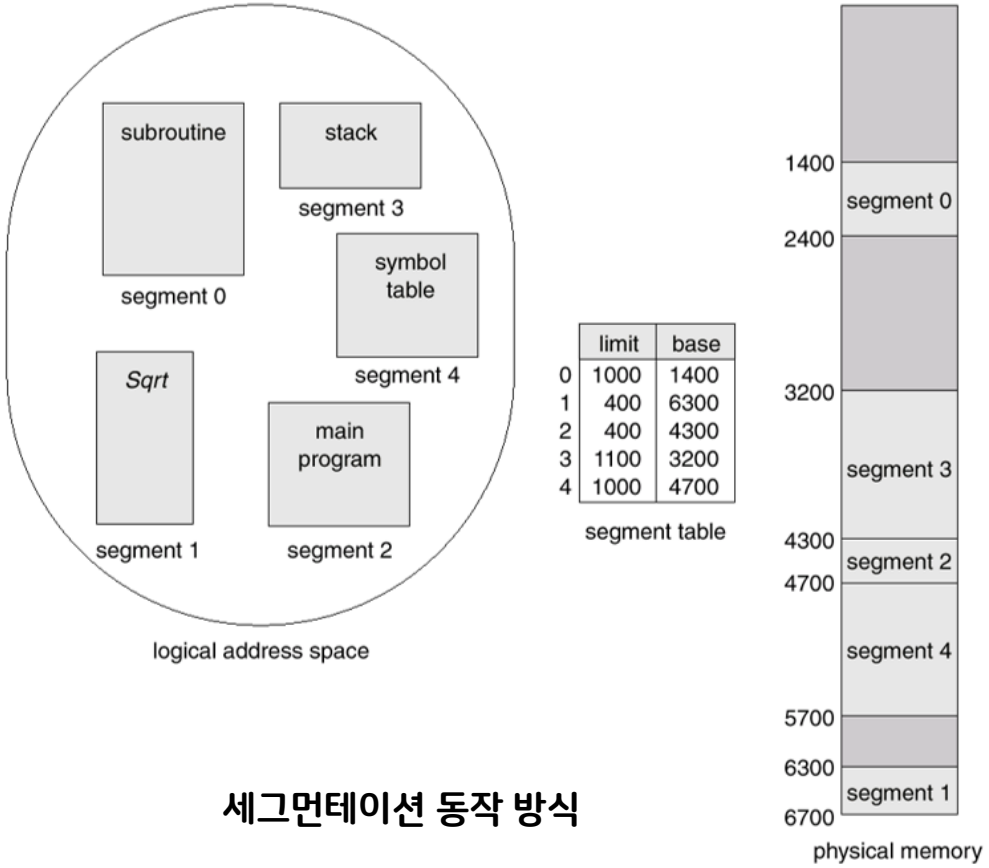
- **메모리 접근 모드와 동작 - 보호 모드(Protected Mode)**
  - 80286부터 도입된 보호 모드(Protected Mode)는 32비트 CPU 80386에 완성
  - 32비트 주소 버스를 통해 4GB 메모리 사용 가능, 메모리 보호 기능과 페이징(Paging) 등으로 가상 메모리를 효율적으로 구현
  - 세그멘테이션(Segmentation)과 페이징(Paging)을 이용 메모리 관리



보호 모드에서의 메모리 변환 과정

x86 시스템 메모리의 구조와 동작

- **메모리 접근 모드와 동작 - 세그멘테이션(Segmentation)**
  - 프로세스가 할당 받은 메모리 공간을 논리적 의미 단위(segment)로 나누어, 연속되지 않는 물리 메모리 공간에 할당하는 메모리 관리 기법
  - 프로세스의 메모리 영역 중 code, data, heap, stack등의 기능 단위로 segment를 정의하는 경우가 많음



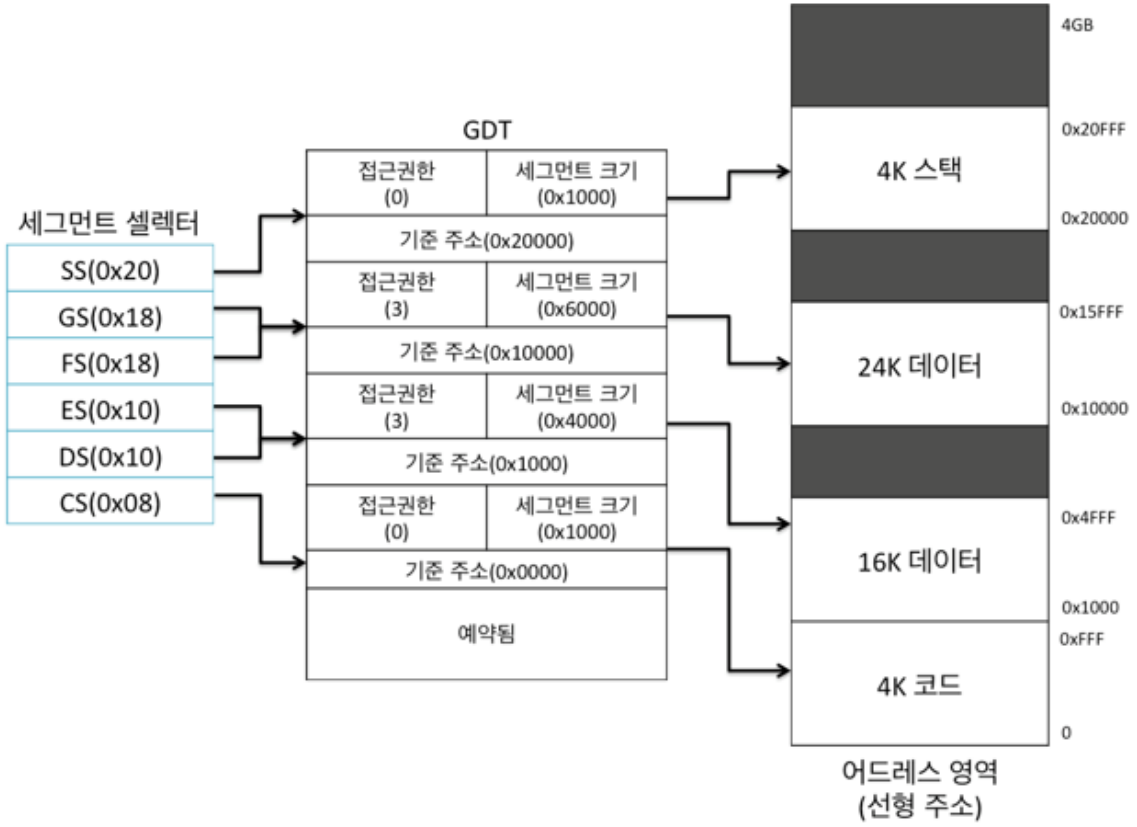
세그먼테이션 동작 방식

- 세그멘테이션은 가상 메모리를 효율적으로 사용할 수 있지만,
  외부 단편화 문제는 해결 안됨

- 외부 단편화는 세그먼트의 크기가 일정하지 않아 물리 메모리에
  사용되지 않는 조각이 발생하게 되며, 이로 인해 메모리 낭비가 발생

x86 시스템의 구조

- **메모리 접근 모드와 동작 - 세그멘테이션(Segmentation)**
  - 세그멘테이션은 4GB의 메모리를 세그먼트 단위로 쪼갠 것으로,
    16비트의 셀렉터와 32비트의 오프셋을 이용해서 4GB 범위의 32비트 선형 주소(linear address)를 만듦



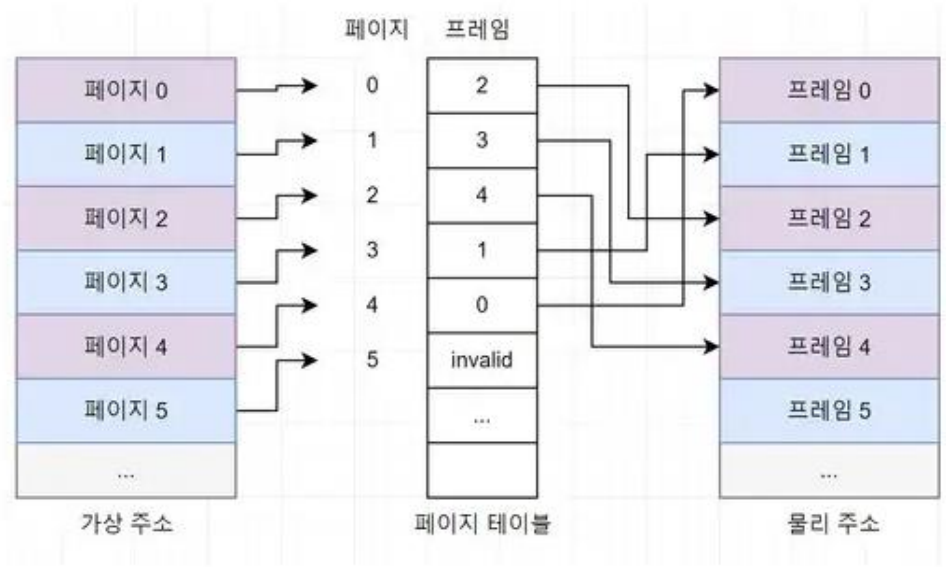
- GDT(Global Descriptor Table)
  GDT는 전역으로 쓰일 디스크립터를 모아 놓은 테이블로
  메모리의 어디든 존재할 수 있지만 그 위치와 크기는
  CPU의 GDTR(GDT register)에 등록시켜 주어야 함

- GDT의 디스크립터 인덱스를 구하는 방법
  세그먼트 셀렉터에 8 * n(10진수) 값을 설정

  셀렉터가 8(0x8)일 경우: 1000 -> 인덱스: 1 -> 10진수: 1
  셀렉터가 16(0x10)일 경우: 10000 -> 인덱스: 10 -> 10진수: 2
  셀렉터가 24(0x18)일 경우: 11000 -> 인덱스: 11 -> 10진수: 3
  셀렉터가 32(0x20)일 경우: 100000 -> 인덱스: 100 -> 10진수: 4

x86 시스템 메모리의 구조와 동작

- ## 메모리 접근 모드와 동작 - 페이징(Paging)
  - 프로세스의 주소 공간을 페이지(page)란 단위의 고정된 사이즈로 나누어 물리적 메모리에 불연속으로 저장하는 방식
  - 실제 메모리는 페이지 크기와 같은 프레임(frame)으로 나누어 사용



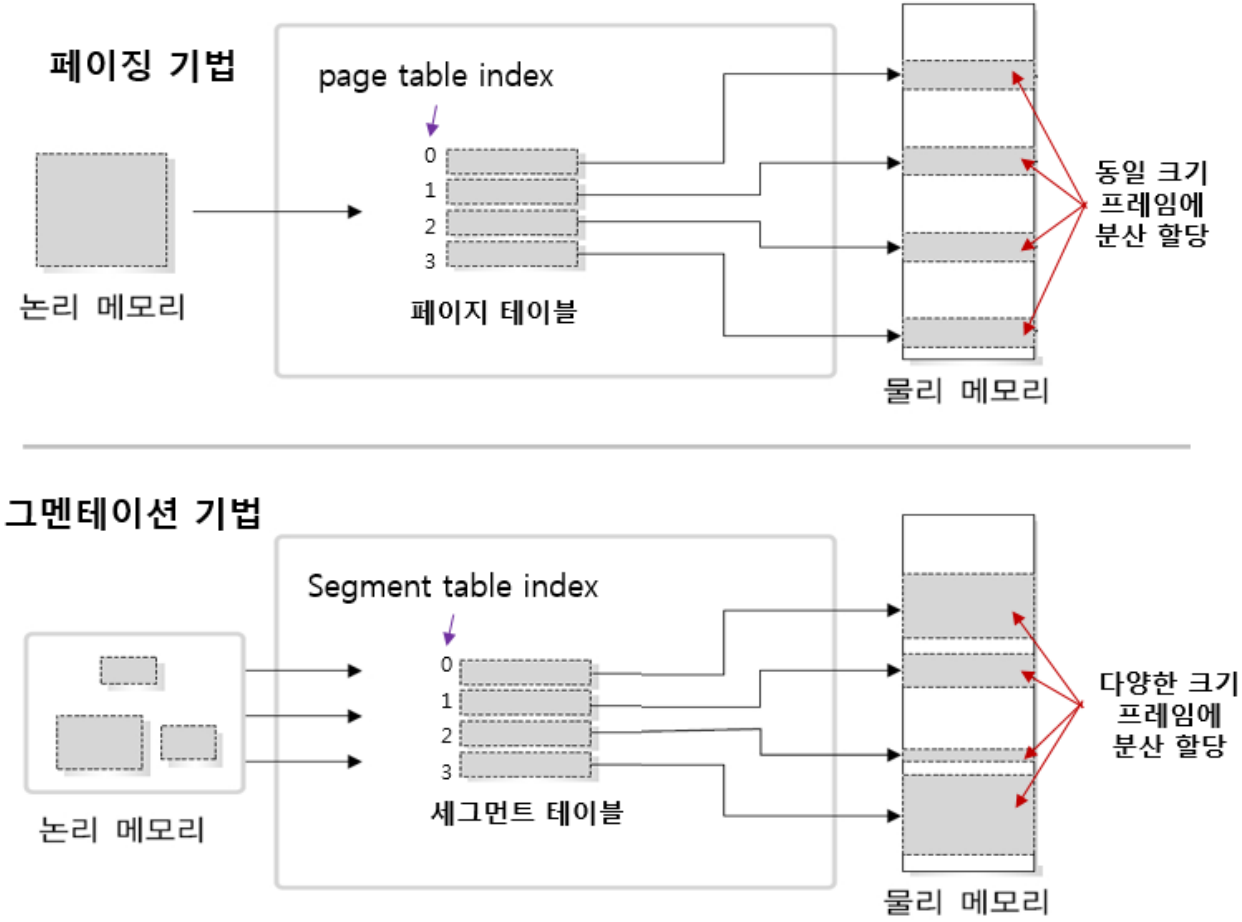
페이징 동작 방식

- 페이징은 물리 메모리를 효율적으로 사용할 수 있지만,
  내부 단편화 문제는 해결 안됨

- 내부 단편화는 프로세스 크기가 페이지 크기의 배수가 아닐 경우,
  마지막 페이지는 한 프레임(페이지)을 다 채울 수 없어서 발생하며
  이로 인해 메모리 낭비가 발생

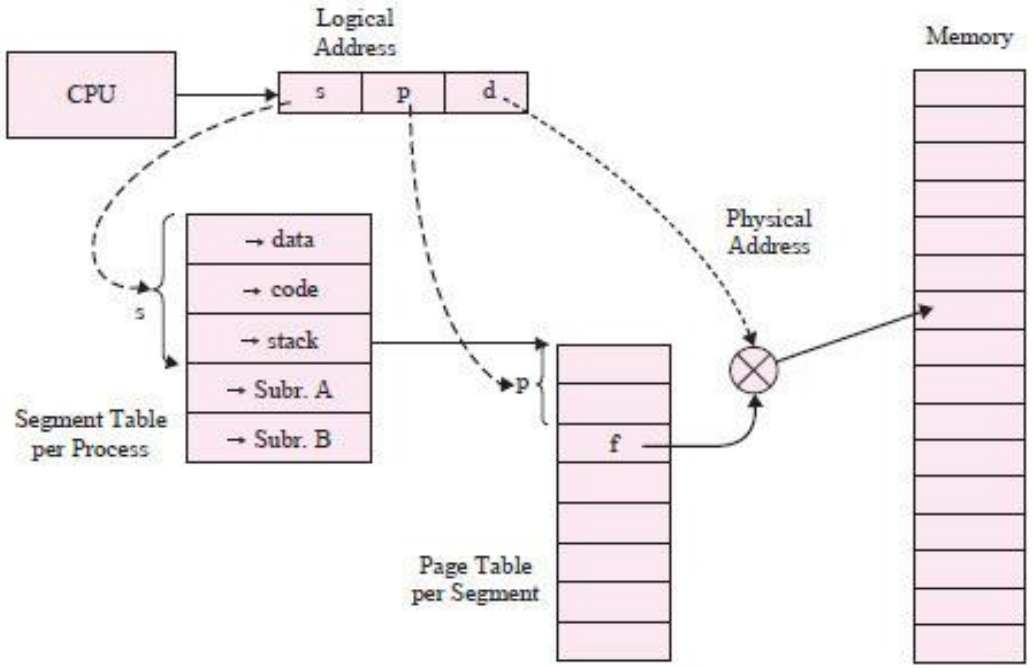
x86 시스템 메모리의 구조와 동작

- **메모리 접근 모드와 동작**
  - 페이징(Paging)과 세그멘테이션(Segmentation) 비교



x86 시스템 메모리의 구조와 동작

- **메모리 접근 모드와 동작**
  - 페이징(Paging)과 세그멘테이션(Segmentation) 혼용



- 프로세스를 처음에 세그먼트 단위로 자름
- 의미 있는 단위로 나누면 보호와 공유를 하는 측면에서 이점을 가질 수 있음
- 세그먼트로 인한 외부 단편화 문제를 해결하기 위해 잘라진 세그먼트를 일정한 간격인 페이지 단위로 자르는 페이징 기법을 적용
- 이후 메모리에 적재하면 페이징의 일정 단위로 다시 잘렸기 때문에 외부 단편화가 발생하지 않음