GeekOS
2. ELF, User Process

March, 2015

Daeyeon Son

Dept. of Software, Dankook University

E-mail : sonn2@daum.net

Introduction

- 우리는 생활 속 주위의 모든 Object에 식별이 가능한 정보를 붙이고 이러한 정보를 이용하여 Object에 대한 Identity를 부여한다.
- 그리고, 'Data에 대한 정보를 담는 Data'를 뜻하는 Metadata를 접하게 됨으로서 각각의 Data를 구별이 가능하게 된다.

Metadata → Address → Object

Introduction

- Disk에 존재하는 File은 Program을 구성하는 Object 중 하나로서, File을 구성하는 Object로 Section, Segment 등이 존재한다.
- 그리고, Section과 Segment등의 정보를 나타내는 Metadata 역할을 수행하는 것이 File Format이라고 이야기 할 수 있다.

- 이번 시간에 배울 내용
  - ✓ Linux에서 사용되는 ELF File Format에 대한 내용
    - Program의 Section, Segment 정보를 Parsing 하여 Memory에 적재하는 과정
  - ✓ Memory에 적재된 Program을 Process로 활용하는 내용
    - CPU Register, Segmentation, Paging
    - GDT, LDT, Context Switching

GeekOS

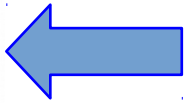
- GeekOS에서는 이러한 자료구조와 함수를 사용합니다.
  - ✓ Part 1 : ELF
    - Main()
      - GeekOS의 초기화 작업을 수행한다.
    - Spawner()
      - Program을 읽고 실행하는 역할을 수행한다.
    - Read_Fully()
      - File Path와 File Size를 읽어 들여 임시 메모리에 적재한다.
    - Parse_ELF_Executable()
      - ELF Header에 대한 정보를 읽어 들인 뒤 실행하고자 하는 Program에 대한 정보를 읽어 들인다.
    - Spawn_Program()
      - Program에서 Segment정보를 추출하여 Memory에 적재하고 Program을 실행하게 된다.

- GeekOS에서는 이러한 자료구조와 함수를 사용합니다.
  - ✓ Part 2 : User Process
    - struct User_Context
      - Process의 Context에 관한 구조체로서 Program의 Memory 시작 주소와 크기를 비롯하여 구성되어 있는 Segment Descriptor에 대한 정보를 담아 놓는다.
    - Load_LDTR()
      - LDT Descriptor를 LDTR Register에 읽어 들인 뒤 해당 Process의 Segment에 대한 정보에 접근한다.
    - Load_User_Program()
      - Program의 Context에 Segment, Stack, Argument, Memory 시작 주소, Memory 크기에 대한 정보를 복사한다.
    - Create_User_Context()
      - GDT에서 LDT Descriptor를 할당 받은 뒤 User_Context 구조체 변수의 초기화 과정을 진행한다.
    - Setup_User_Thread()
      - Thread에 Process의 Context를 연결 시킨 뒤 Process의 Stack Segment에 현재 Process에 관한 모든 State를 저장 시키고 정상적으로 작동하도록 만든다.

File

- File
  - ✓ Computer 저장과 처리 목적을 위해 Binary 형식으로 Encoding된 Data
    - Kernel : Binary File Format 사용
    - Application : ELF File Format 사용

Disk



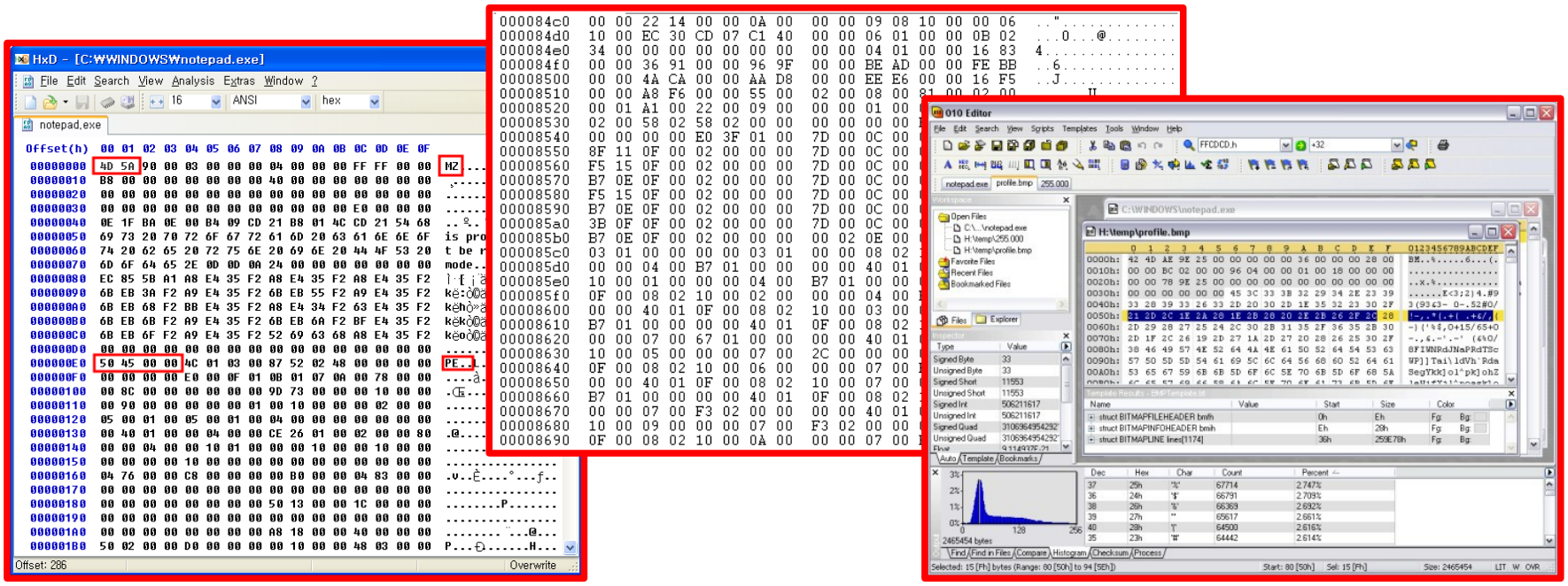
```
01001010101100
10011111010100
00011001010100
11010011
```

```
0100
01010011010100
10101001010100
00100111101010
```

File

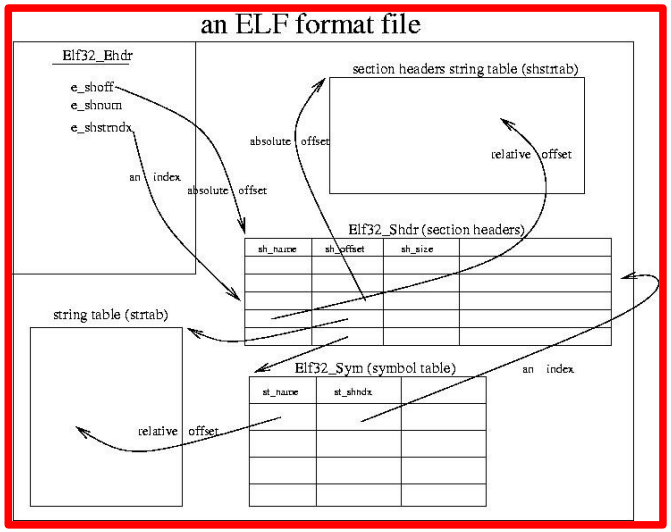
File Format

- File Format
  - ✓ 일반적으로 OS는 File을 실행하기 전에 File Header에 대한 정보를 먼저 읽어 들인 뒤 Memory에 Segment, Section과 관련된 정보를 적재시킨다.
  - ✓ 이러한 정보는 OS마다 각기 다르며, Windows의 경우는 PE 포맷, Unix와 Linux의 경우는 ELF 포맷, MacOS의 경우는 Mach-O 포맷을 사용한다.



ELF

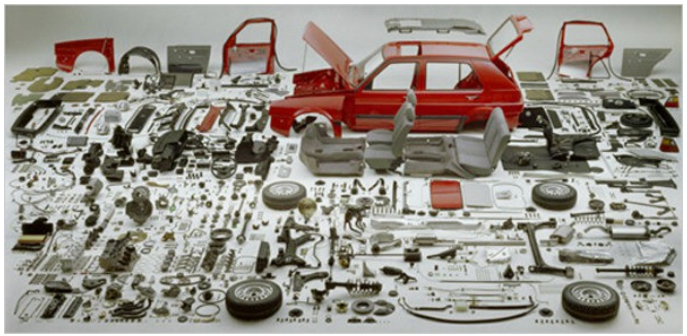
- ELF
  - ✓ Executable and Linkable Format
  - ✓ Code, Data, Relocation 정보, Symbol 정보, Debugging 정보 등을 Section으로 구분하여 저장
    - Section의 정보를 담고 있는 Section Header를 정의하여 테이블 관리 하는 것이 특징
  - ✓ ELF 표준은 이진 인터페이스를 프로그래머에게 제공하여 소프트웨어 개발에 연계성을 주기 위해 만들어 짐



ELF

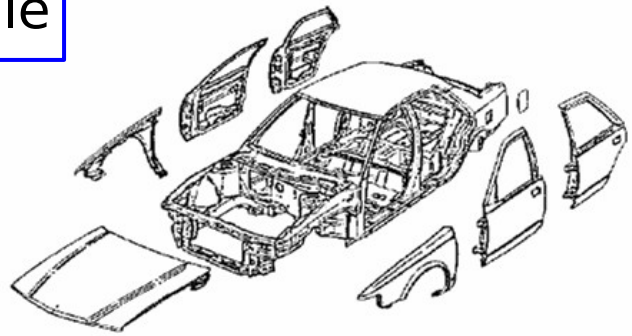
- ELF 파일
  - ✓ ELF File Format이 존재하는 Data

File



AAA.c

ELF File

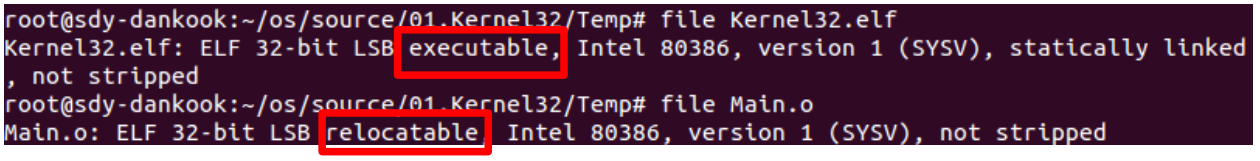


AAA.o

AAA.elf

ELF

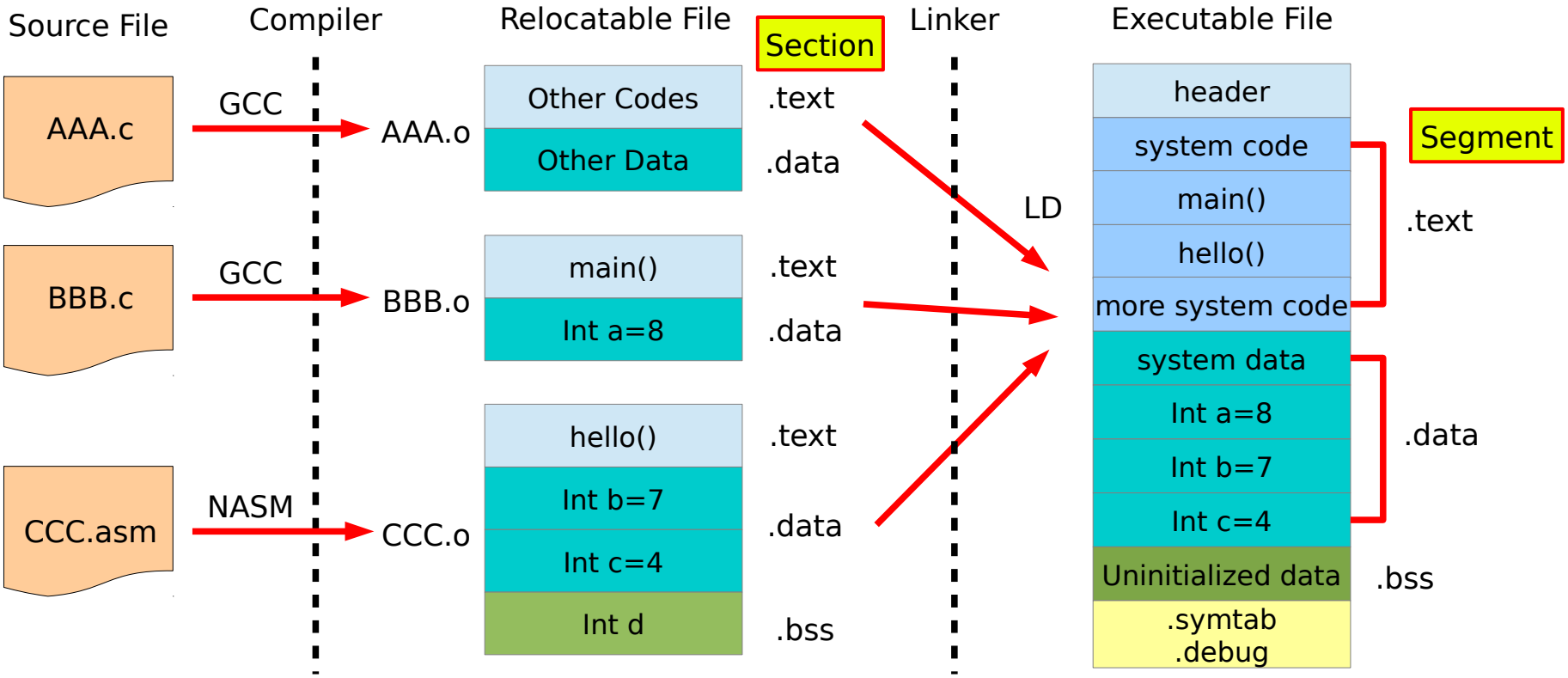
- ELF 파일 타입
  - ✓ Relocatable File (* .o)
    - 다른 Object 파일과 Linking 됨으로서 실행 프로그램이나 공유 목적 파일을 생성할 수 있는 Code와 Data를 가지고 있는 파일
  - ✓ Executable File
    - Code와 Data를 Target OS에서 실행 될 수 있도록 하는 파일
  - ✓ Shared Object File (* .so)
    - Linker를 이용하여 Executable File과 다른 Shared Object File을 연결하여 프로그램을 실행시키는 파일



Linux상에서 ELF 파일 타입 확인 가능함

ELF

- ELF 파일 구조
  - ✓ 여러 개의 Relocatable Object File 내용이 합쳐져서 Executable Object File의 내용을 생성
  - ✓ Compile 과정 이후 Linking 과정을 수행하는 이유는 Executable File을 생성하기 위한 것



ELF

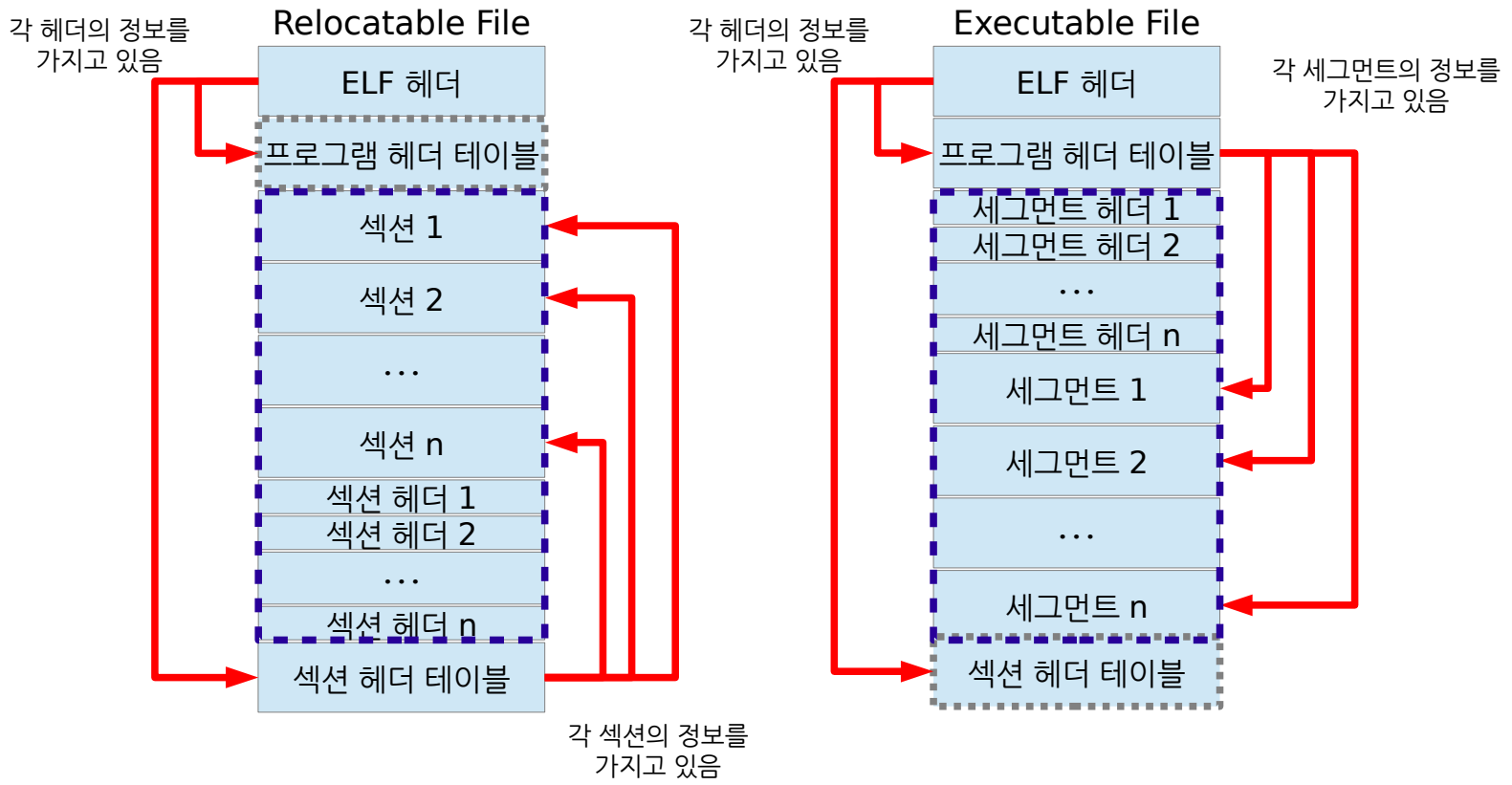
- ELF의 Section과 Segment
  - ✓ Section : Linking 과정에서 필요한 정보를 담고 있음
    - Object File(* .o)의 모든 정보 포함
  - ✓ Segment : Program 실행 시에 필요한 정보를 담고 있음
    - Executable File(* .elf)의 정보 포함
    - 같은 속성의 Section을 묶어 놓은 것이 Segment
    - Segment는 0개 이상의 Section들로 구성

```
root@sdy-dankook:~# readelf -l test
```

```
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .ctors .dtors .jcr .dynamic .got
root@sdy-dankook:~#
```

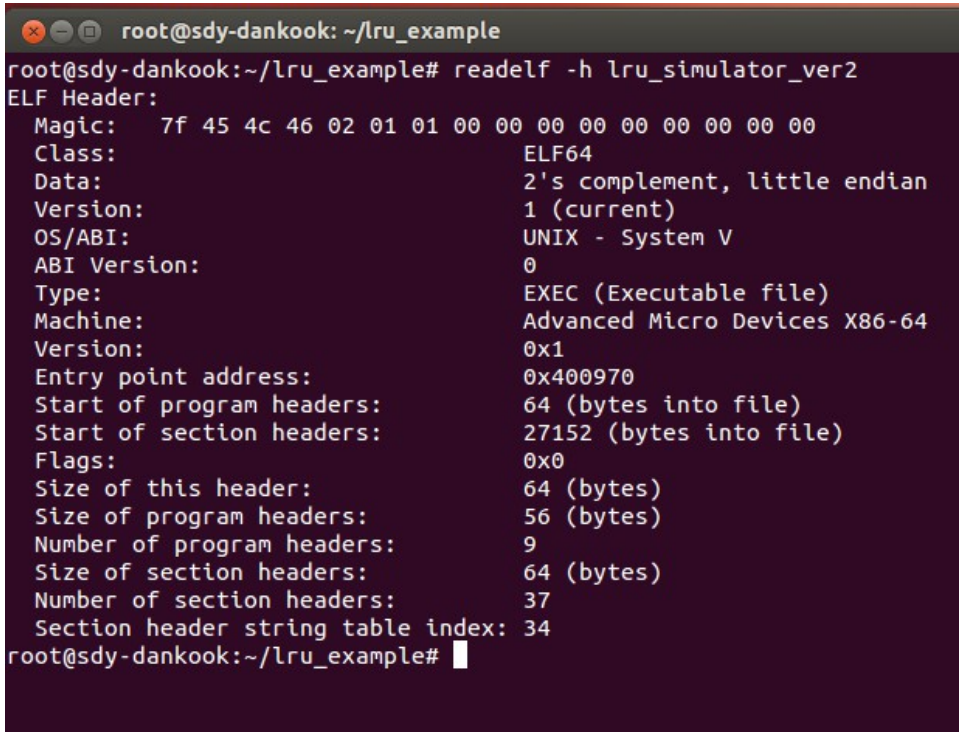
ELF

- ELF 파일 구조
  - ✓ ELF 파일은 Relocatable File, Executable File로 나뉘는데 실제 구조는 File을 구성하는 Header Table과 Relocation 정보의 유무 외에는 큰 차이가 없음.



ELF

- ELF Header 자료구조
  - ✓ ELF Header는 총 14개의 Field로 구성
    - Linux에서는 readelf 명령어를 통하여 ELF Header 내용 볼 수 있음



ELF

- ELF Header 자료구조
  - ✓ Linux Kernel에서는 (kernel dir)/include/uapi/linux/elf.h에 자료구조가 기술되어 있음
  - ✓ 32bit와 64bit 모두 동일한 자료구조 사용하며, 유일한 차이점은 Data Type의 Length가 다르다는 것

```
#define EI_NIDENT   16

typedef struct elf32_hdr{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;
  Elf32_Half    e_machine;
  Elf32_Word    e_version;
  Elf32_Addr    e_entry;  /* Entry point */
  Elf32_Off e_phoff;
  Elf32_Off e_shoff;
  Elf32_Word    e_flags;
  Elf32_Half    e_ehsize;
  Elf32_Half    e_phentsize;
  Elf32_Half    e_phnum;
  Elf32_Half    e_shentsize;
  Elf32_Half    e_shnum;
  Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

32bit의 ELF Header

```
typedef struct elf64_hdr {
  unsigned char e_ident[EI_NIDENT]; /* ELF "magic number" */
  Elf64_Half e_type;
  Elf64_Half e_machine;
  Elf64_Word e_version;
  Elf64_Addr e_entry;       /* Entry point virtual address */
  Elf64_Off e_phoff;        /* Program header table file offset */
  Elf64_Off e_shoff;        /* Section header table file offset */
  Elf64_Word e_flags;
  Elf64_Half e_ehsize;
  Elf64_Half e_phentsize;
  Elf64_Half e_phnum;
  Elf64_Half e_shentsize;
  Elf64_Half e_shnum;
  Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

64bit의 ELF Header

ELF

- ELF Header 자료구조
  - ✓ Header의 내용만 파악하더라도 Program의 기본 정보 수집 가능

```
#define Elf32_Addr      unsigned long
#define Elf32_Off       unsigned long
#define Elf32_Half      unsigned short int
#define Elf32_Word      unsigned int
#define Elf32_Sword     int
#define Elf32_Xword     unsigned long
#define Elf32_Sxword    long

typedef struct {
        unsigned char   e_ident[16];     // ELF 식별자
        Elf32_Half      e_type;          // 오브젝트 파일 형식
        Elf32_Half      e_machine;       // 머신 타입
        Elf32_Word      e_version;       // 오브젝트 파일 버전
        Elf32_Addr      e_entry;         // 엔트리 포인트 어드레스
        Elf32_Off       e_phoff;         // 파일 내에 존재하는 프로그램 헤더 테이블의 위치
        Elf32_Off       e_shoff;         // 파일 내에 존재하는 섹션 헤더 테이블의 위치
        Elf32_Word      e_flags;         // 프로세서 의존적인 플래그
        Elf32_Half      e_ehsize;        // ELF 헤더의 크기
        Elf32_Half      e_phentsize;     // 프로그램 헤더 엔트리 한 개의 크기
        Elf32_Half      e_phnum;         // 프로그램 헤더 엔트리의 개수
        Elf32_Half      e_shentsize;     // 섹션 헤더 엔트리 한 개의 크기
        Elf32_Half      e_shnum;         // 섹션 헤더 엔트리의 개수
        Elf32_Half      e_shstrndx;      // 섹션 이름 문자열이 저장된 섹션 헤더의 인덱스
} Elf32_Ehdr;
```

Header size : 52 byte

ELF

- ELF Header 자료구조
  - ✓ Linux Kernel의 경우 init_module()이 초기에 ELF Image를 Kernel Space로 넣어주는 역할을 수행
  - ✓ (kernel dir)/include/linux/module.h

```
INIT_MODULE(2)                    Linux Programmer's Manual                    INIT_MODULE(2)

NAME
       init_module - initialize a loadable module entry

SYNOPSIS
       #include <linux/module.h>

       int init_module(const char *name, struct module *image);

DESCRIPTION
       init_module()  loads  the  relocated module image into kernel space and runs the module's
       init function.

       The module image begins with a module structure and is  followed  by  code  and  data  as
       appropriate.  The module structure is defined as follows:
```

ELF

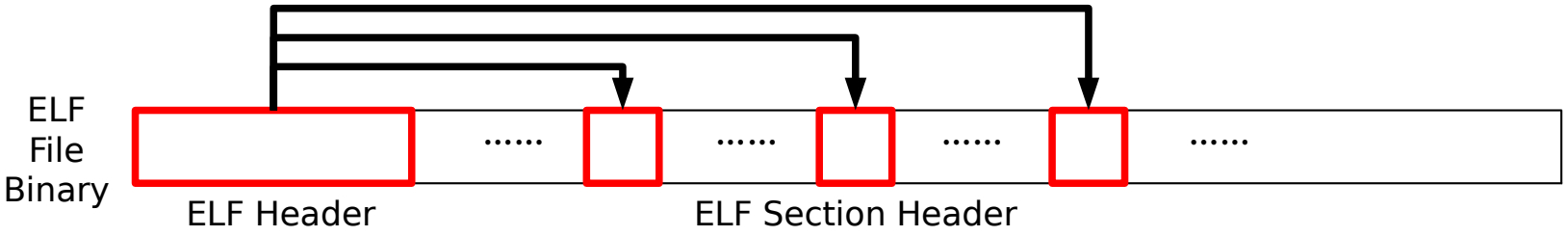
- ELF Section Header 자료구조
  - ✓ ELF Header와 동일하게 32bit, 64bit의 차이점은 Data Type의 Length가 다르다는 것
  - ✓ ELF Section Header 역시 File을 구성하는 Section에 대한 기본 정보를 수록
  - ✓ (kernel dir)/uapi/linux/elf.h

```
typedef struct elf32_shdr {
  Elf32_Word    sh_name;
  Elf32_Word    sh_type;
  Elf32_Word    sh_flags;
  Elf32_Addr    sh_addr;
  Elf32_Off sh_offset;
  Elf32_Word    sh_size;
  Elf32_Word    sh_link;
  Elf32_Word    sh_info;
  Elf32_Word    sh_addralign;
  Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

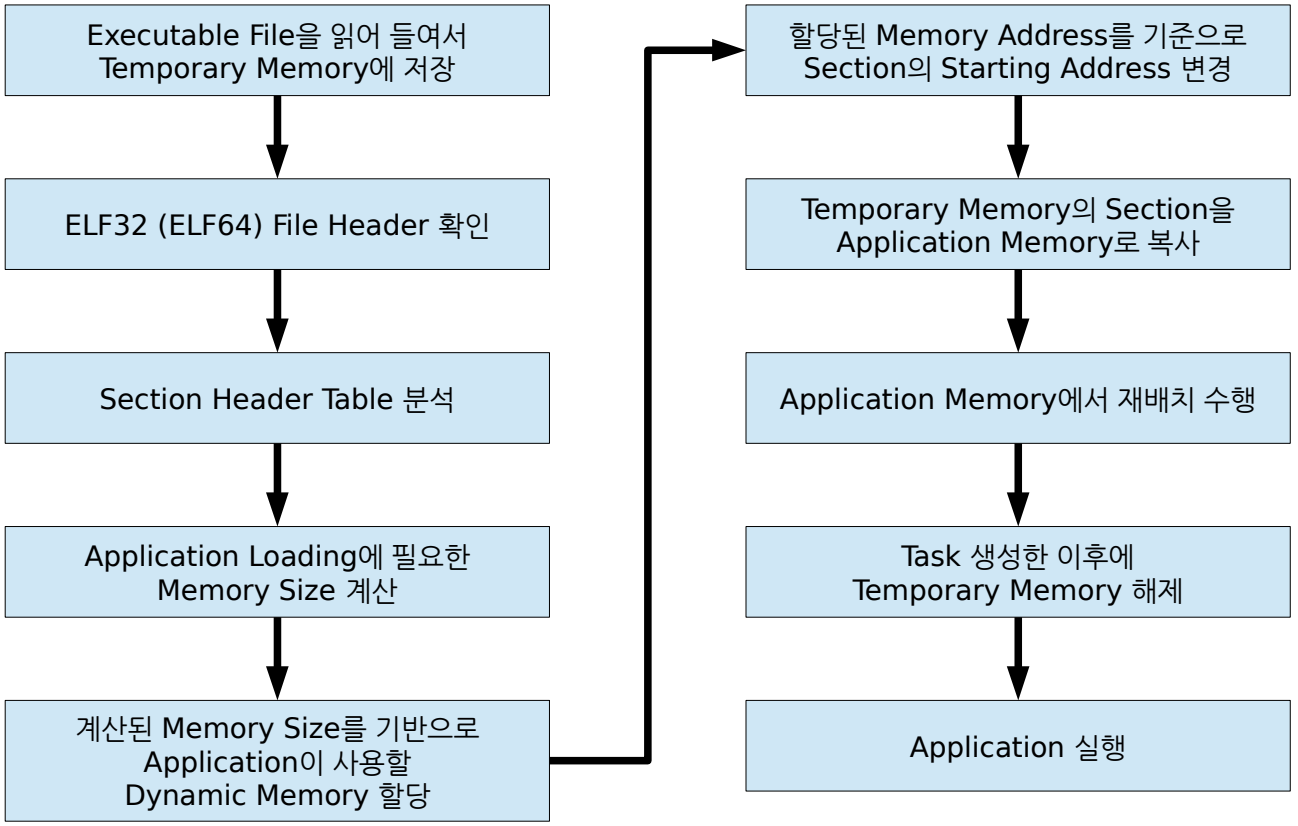
32bit의 ELF Section Header

```
typedef struct elf64_shdr {
  Elf64_Word sh_name;       /* Section name, index in string tbl */
  Elf64_Word sh_type;       /* Type of section */
  Elf64_Xword sh_flags;     /* Miscellaneous section attributes */
  Elf64_Addr sh_addr;       /* Section virtual addr at execution */
  Elf64_Off sh_offset;      /* Section file offset */
  Elf64_Xword sh_size;      /* Size of section in bytes */
  Elf64_Word sh_link;       /* Index of another section */
  Elf64_Word sh_info;       /* Additional section information */
  Elf64_Xword sh_addralign; /* Section alignment */
  Elf64_Xword sh_entsize;   /* Entry size if section holds table */
} Elf64_Shdr;
```

64bit의 ELF Section Header

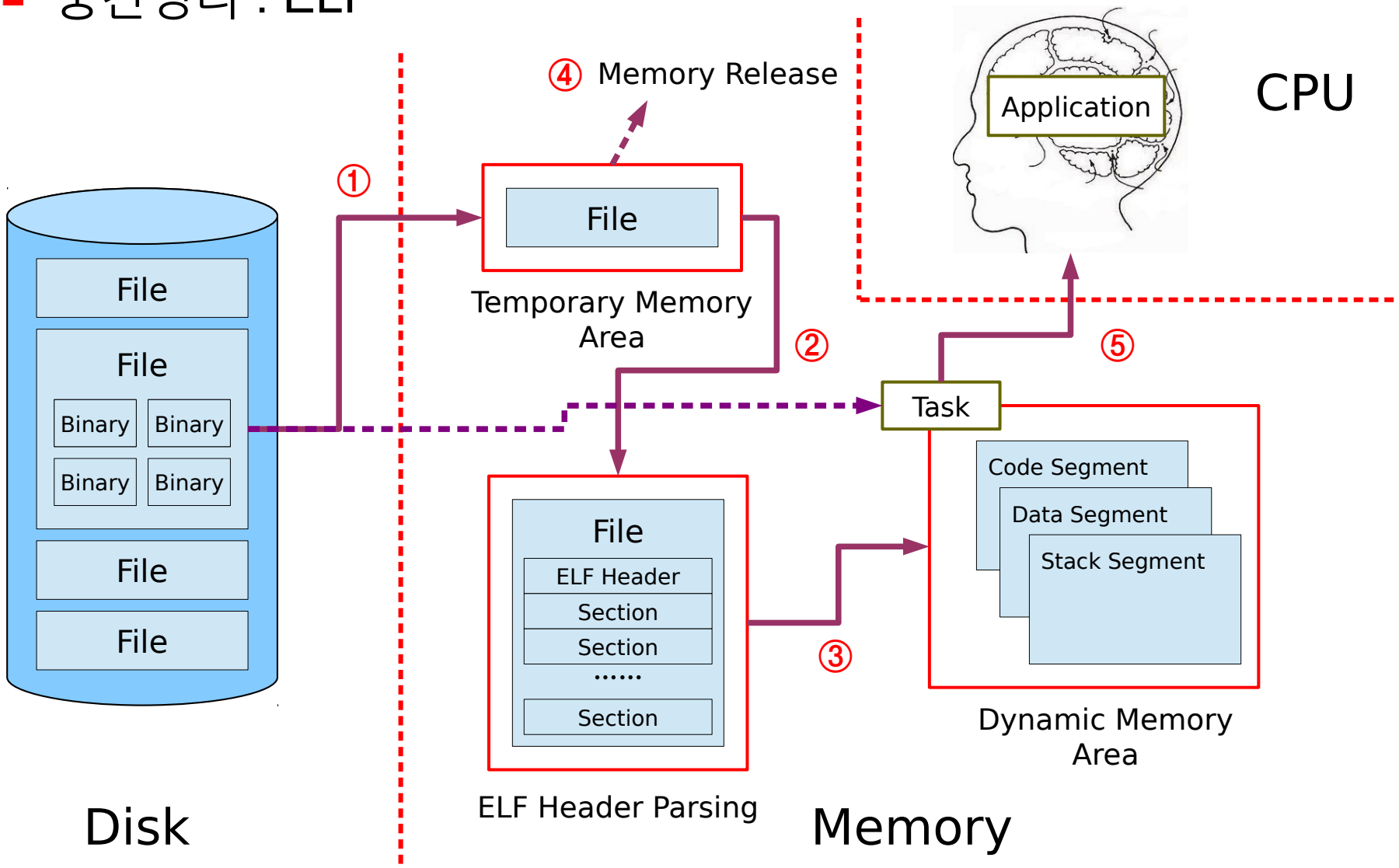


- ELF 파일의 실행 과정 정리



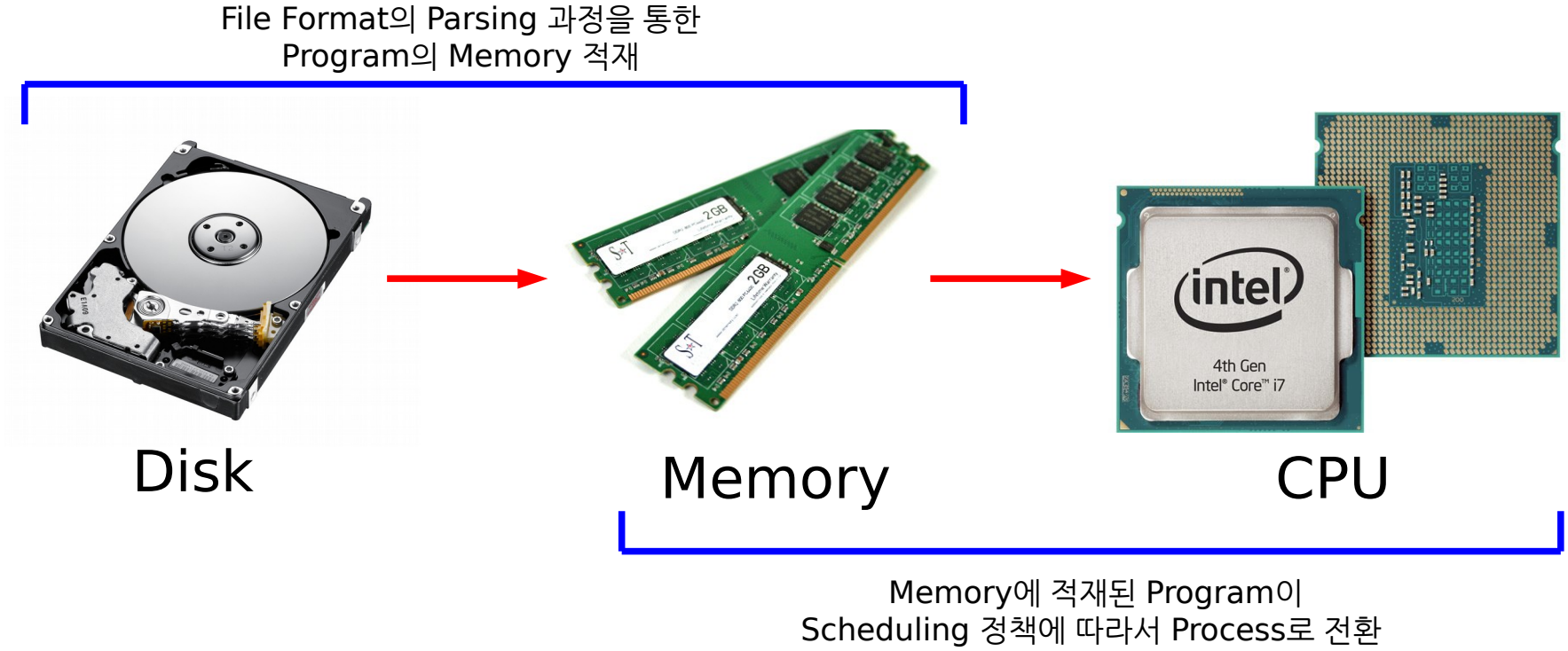
ELF

- 중간정리 : ELF


④ Memory Release
CPU
Application
①
File
Temporary Memory Area
②
⑤
Task
File
File
Binary
Binary
Binary
Binary
File
File
Code Segment
Data Segment
Stack Segment
File
ELF Header
Section
Section
.....
Section
③
Dynamic Memory Area
ELF Header Parsing
Disk
Memory

User Process

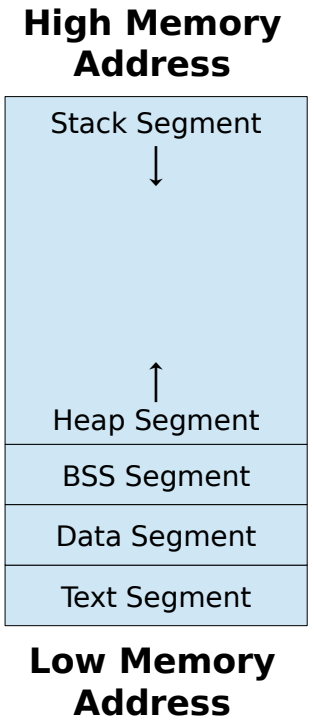
- User Process
  - ✓ File Format 분석 과정 이후 각 Segment의 정보를 통하여 Memory에 적재가 완료되면 Process로서의 역할 시작



User Process

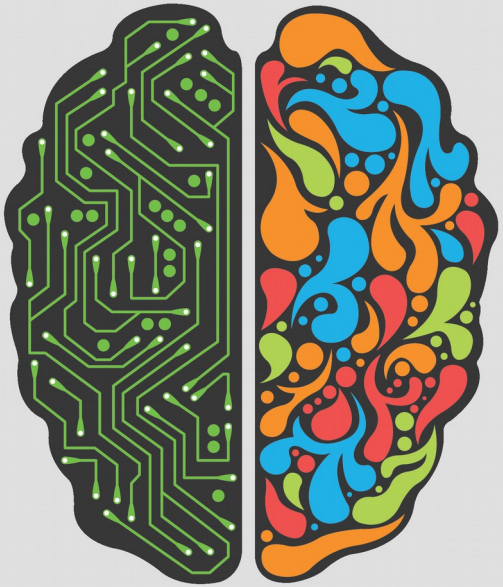
Segment Type

- ✓ Segment는 Memory의 거의 어느 곳에나 위치 할 수 있음.
- ✓ Text Segment (Code Segment)
  - Program의 Instruction이 포함되어 있으며 Code만 저장되어 있기 때문에, 읽기 전용의 Segment에 해당 됨.
- ✓ Data Segment
  - 초기화 된 Global 변수와 Static 변수가 저장 됨.
    - Static 변수 → static 키워드를 사용하여 선언된 변수
- ✓ BSS(Block Started Symbol) Segment
  - 초기화 되지 않은 Global 변수와 Static 변수가 저장 됨.
- ✓ Heap Segment
  - 직접 할당한 Memory가 저장되는 영역이며, 일반적으로 malloc() 함수를 사용하여 Memory를 할당 할 수 있음.
- ✓ Stack Segment
  - Local 변수와 각종 정보들이 저장되는 영역이며, Function의 Local 변수를 저장하고 사용하는데 Stack 영역이 사용 됨.



User Process

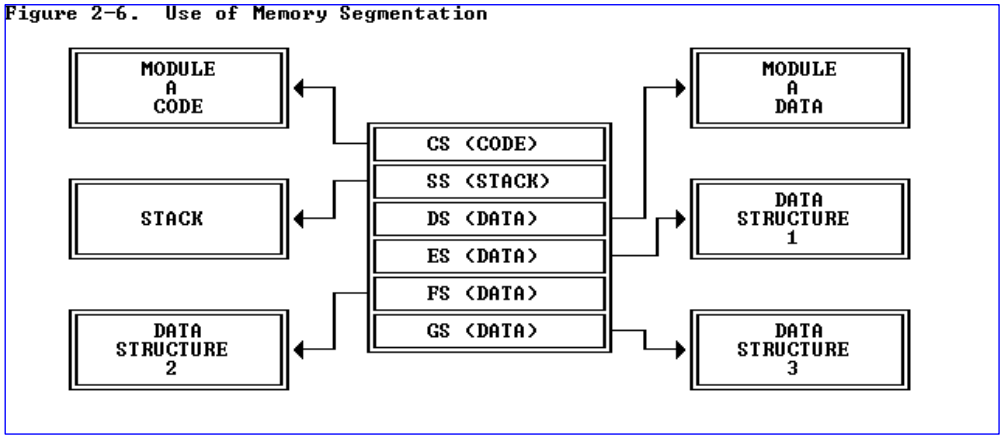
- CPU
  - ✓ Central Processing Unit
  - ✓ Software 명령의 실행이 이루어지는 컴퓨터의 부분, 혹은 그 기능을 내장한 칩
  - ✓ Computer 안의 중앙 처리 장치(CPU)는 외부에서 정보를 입력 받고, 기억하고, Program의 명령어를 해석하여 연산하고, 외부로 출력하는 역할


사람의 두뇌와 같이 CPU도 내부적으로 처리하는 부분이 따로 존재하지 않을까?

User Process

- Process & CPU Register
  - ✓ Process의 Data는 Context라는 개념의 상태 값으로 나타낼 수 있는데, 이러한 Context를 구성하는 단위는 CPU의 Register이다.
  - ✓ 일반적으로 CPU는 4가지 종류의 Register를 포함하고 있다.
    - Data Register
      - AX, BX, CX, DX
    - Segment Register (Segment Selector)
      - CS (Code Segment)
      - DS (Data Segment)
      - SS (Stack Segment)
      - ES (Extra Segment)
      - FS, GS
    - Index Register
      - BP, SP, SI, DI
    - Special Register
      - IP, FLAGS



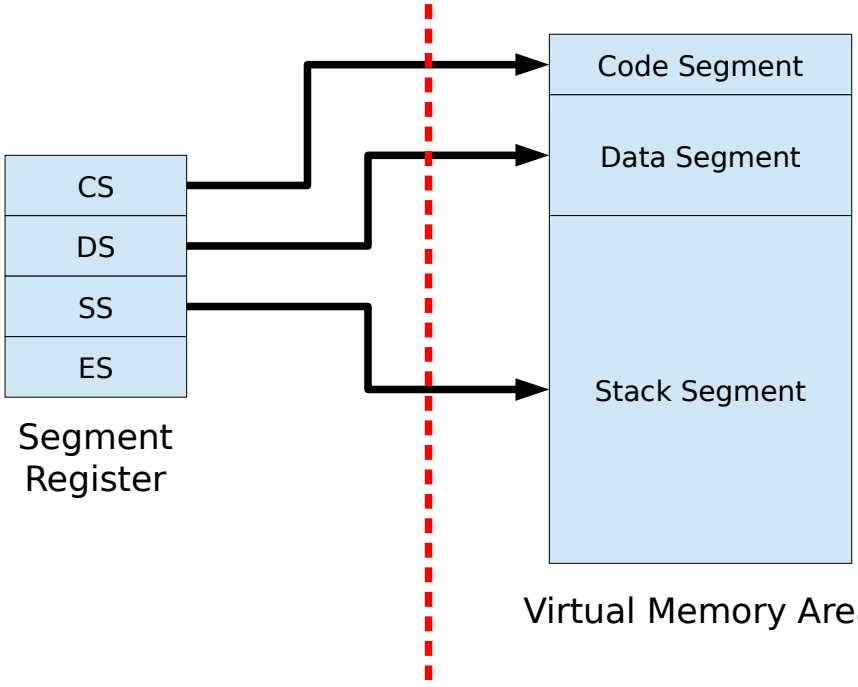
참고 : **http://intel80386.com/386htm/s02_03.htm**

  - ✓ Segment Register의 경우 CPU의 Virtual Memory 할당 방식이 Segmentation, Paging으로 나뉘게 된다.

User Process

- Segmentation
  - ✓ Segment = 동일 속성을 지닌 Section들의 집합
  - ✓ Segmentation = Program을 구성하는 각각의 Segment 크기에 따라서 Virtual Memory 영역을 분할하여 사용하는 기법
  - ✓ 각각의 Segment 크기가 다르기 때문에 해당 크기 만큼 비어있는 Memory 공간을 찾아서 할당

CPU



Memory

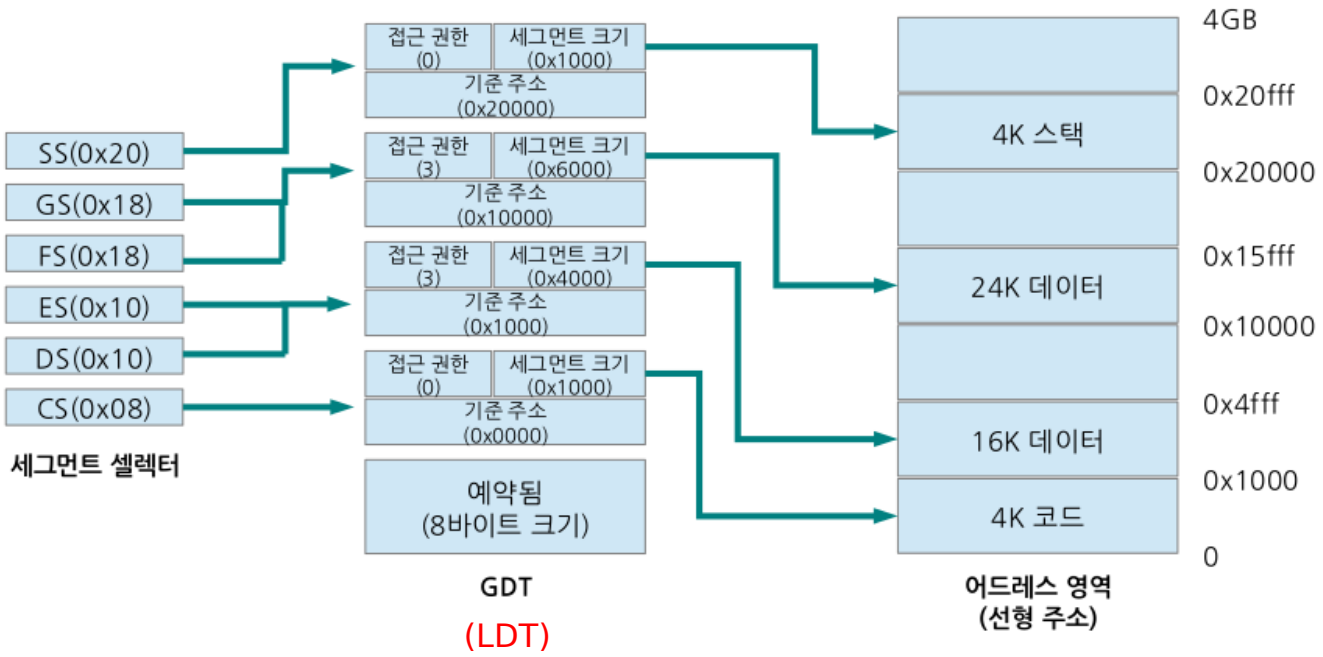
User Process

- Segmentation
  - ✓ Segmentation은 Process의 Memory 할당 영역이 크기에 따라서는 다른 Process의 Segment와 중복이 될 가능성이 존재한다.

- 실습 조교가 소개하는 코끼리를 냉장고에 넣는 법 -
A. 코끼리를 잘게 잘라주세요.
B. A방법이 힘들면 엄청나게 큰 냉장고를 준비해주세요.

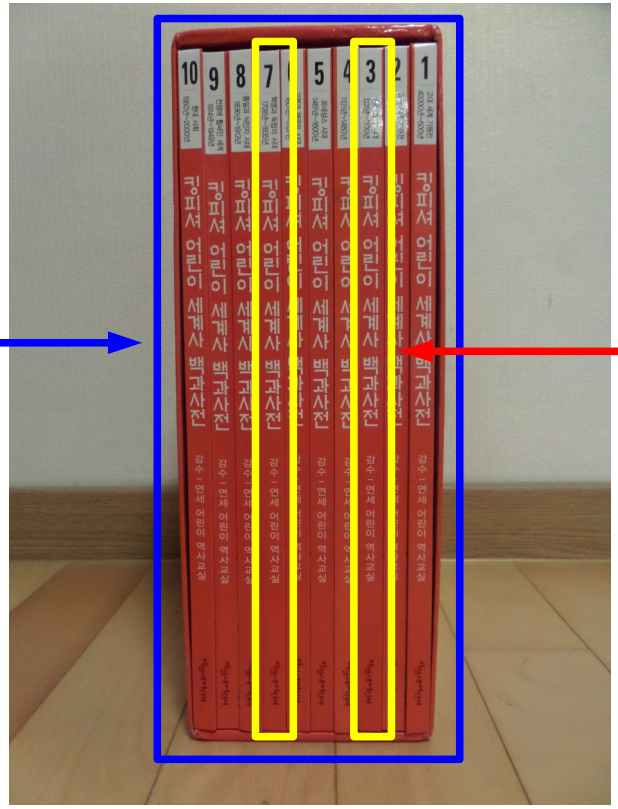
User Process

- Segmentation
  - ✓ 각각의 Segment마다 효율적인 Virtual Memory 할당을 위해서는 Register에 등록된 특정 Memory Address를 기점으로 정하고 Page 단위로 잘게 나누어 할당하는 것이 통상적인 방법이다.
  - ✓ 그리고, 32bit 이상의 OS에서 Segment Register에는 Segment에 관한 LDT의 Descriptor Number가 들어가게 되어 Segment Selector라는 명칭으로 바뀌게 된다.



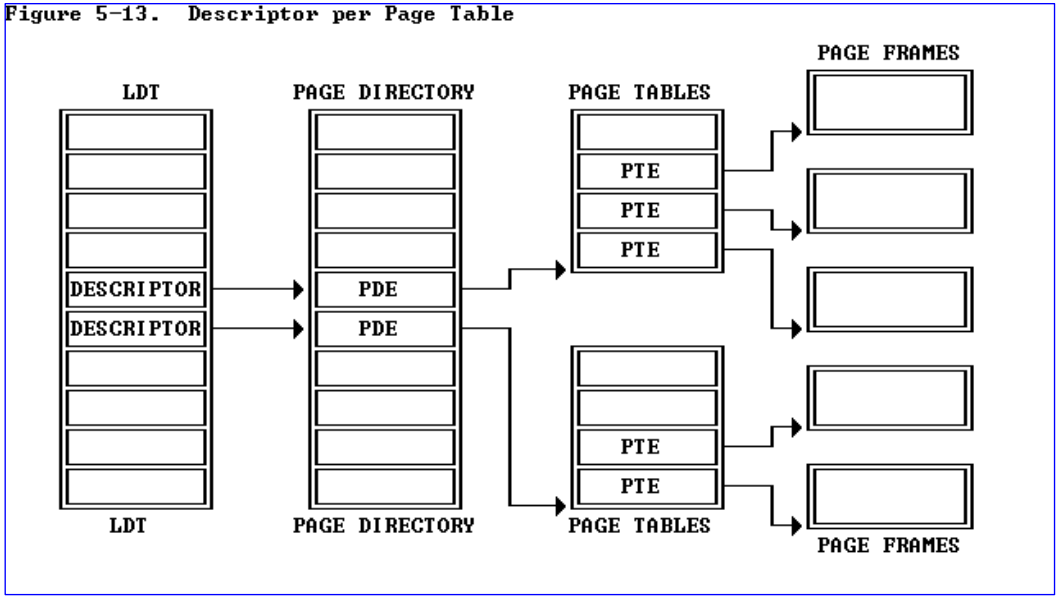
User Process

- Paging
  - ✓ Paging은 Segmentation이 가지고 있던 단점을 보완하고자 등장하였고, Process의 Segment가 사용하는 Memory 영역을 잘게 나누게 되었다.



User Process

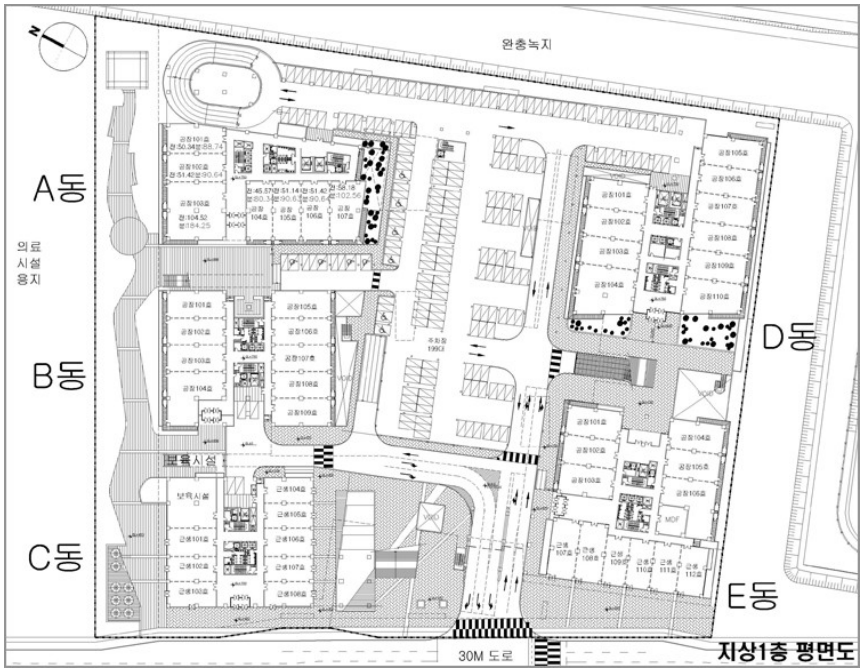
- Paging
  - ✓ Virtual Memory를 논리적 의미와는 관계 없이 모두 같은 크기의 Block으로 나누어 사용하는 기법
  - ✓ 이렇게 나뉘어진 일정한 크기의 Block을 Page라고 부른다.
  - ✓ 일반적으로 많이 쓰이는 Page의 크기는 4KB
  - ✓ 여러 개의 Page를 Group 별로 묶어서 사용하며 이에 따라서 Frame, Table, Directory 등의 분류로 나뉘게 된다.



참고 : **http://intel80386.com/386htm/s02_03.htm**

User Process

- GDT
  - ✓ Booting 이후 Computer의 Memory는 어떤 Software가 얼마만큼 사용할 지 정해지지 않는다.
  - ✓ OS는 자신의 Memory 사용 영역을 어떻게 할당 하는가?



OS는 자신을 비롯하여 Process가 적재될 Memory 공간에 대한 기본적인 설정을 해줄 의무가 있다.

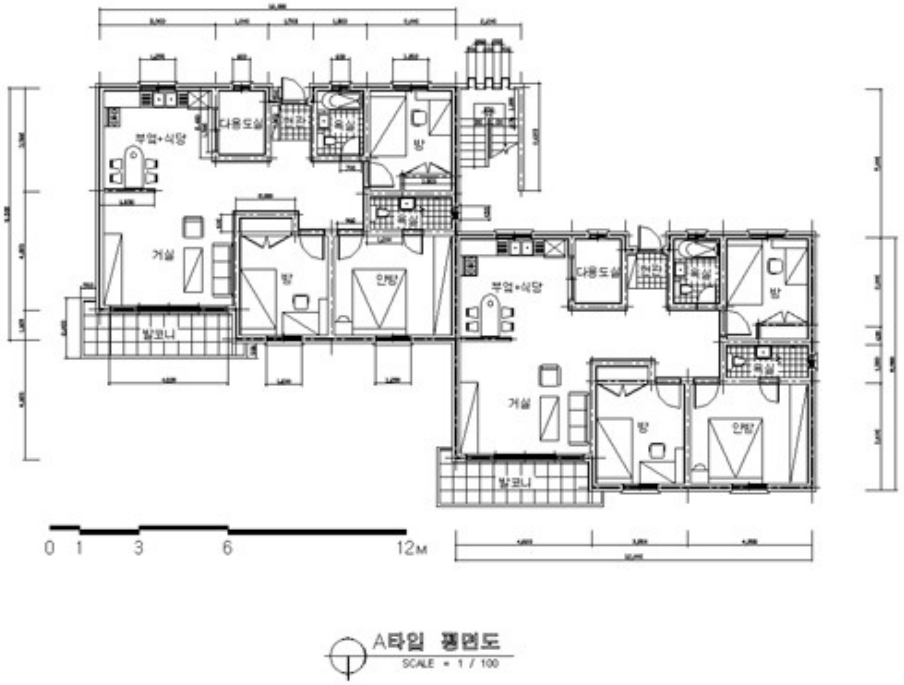
User Process

- GDT
  - ✓ Global Descriptor Table
  - ✓ GDTR Register를 통하여 접근
  - ✓ OS 전체에 관한 Code Segment Descriptor, Data Segment Descriptor를 연속된 Assembly Code로 나타내면 그 전체 영역이 GDT가 된다.
  - ✓ GDT는 맨 처음에 Null Descriptor가 삽입되고, 그 이후에 Memory 영역에 대한 Descriptor를 임의로 작성이 가능하다.
  - ✓ 참고로 OS 초기 작동 시 열거되는 GDT의 내용에 따라서 이후에 OS에서 작동되는 전체 Program 및 Process에 관한 Segment의 Memory 할당 영역이 제한 받을 수 있다.
  - ✓ LGDT instruction을 통하여 GDT 내용 읽기 가능

```
align 8, db 0       ; set the unit of memory size

; unit 1 (2+2+4 = 8byte)
dw 0x0000                           ; 2byte

; 'GDTR' register setting --> input the size of 'GDT'
GDTR:
    dw  GDTEND - GDT - 1            ; 2byte
    dd  ( GDT - $$ + 0x10000 )      ; 4byte

; unit 2~6 (8*5 = 48byte)
; 'GDT' setting
GDT:
    NULLDescriptor:     ; 1. NULL (basic) -> 8byte
        dw  0x0000
        dw  0x0000
        db  0x00
        db  0x00
        db  0x00
        db  0x00

    IA_32eCODEDESCRIPTOR:
        dw 0xFFFF
        dw 0x0000
        db 0x00
        db 0x9A
        db 0xAF
        db 0x00

    IA_32eDATADESCRIPTOR:
        dw 0xFFFF
        dw 0x0000
        db 0x00
        db 0x92
        db 0xAF
        db 0x00

    CODEDESCRIPTOR:     ; 4. Code Segment -> 8byte
        dw  0xFFFF
        dw  0x0000
        db  0x00
        db  0x9A
        db  0xCF
        db  0x00

    DATADESCRIPTOR:     ; 5. Data Segment -> 8byte
        dw  0xFFFF
        dw  0x0000
        db  0x00
        db  0x92
        db  0xCF
        db  0x00

GDTEND:
```

User Process

- LDT
  - ✓ GDT를 토대로 OS 전체의 Memory 영역이 정해지면, 이를 기반으로 Process의 Memory 사용 영역을 결정해야 하는데, LDT가 그러한 역할을 수행한다.



OS는 GDT를 기반으로
LDT Descriptor가 삽입되도록 준비해주고

Process는 LDT를 통하여
각 Segment가 들어갈 Memory 공간을 설정한다.

User Process

- LDT

입주자 여러분들을
환영합니다~

GDT

LDT

Process A

LDT

Process B

LDT

Process C

LDT

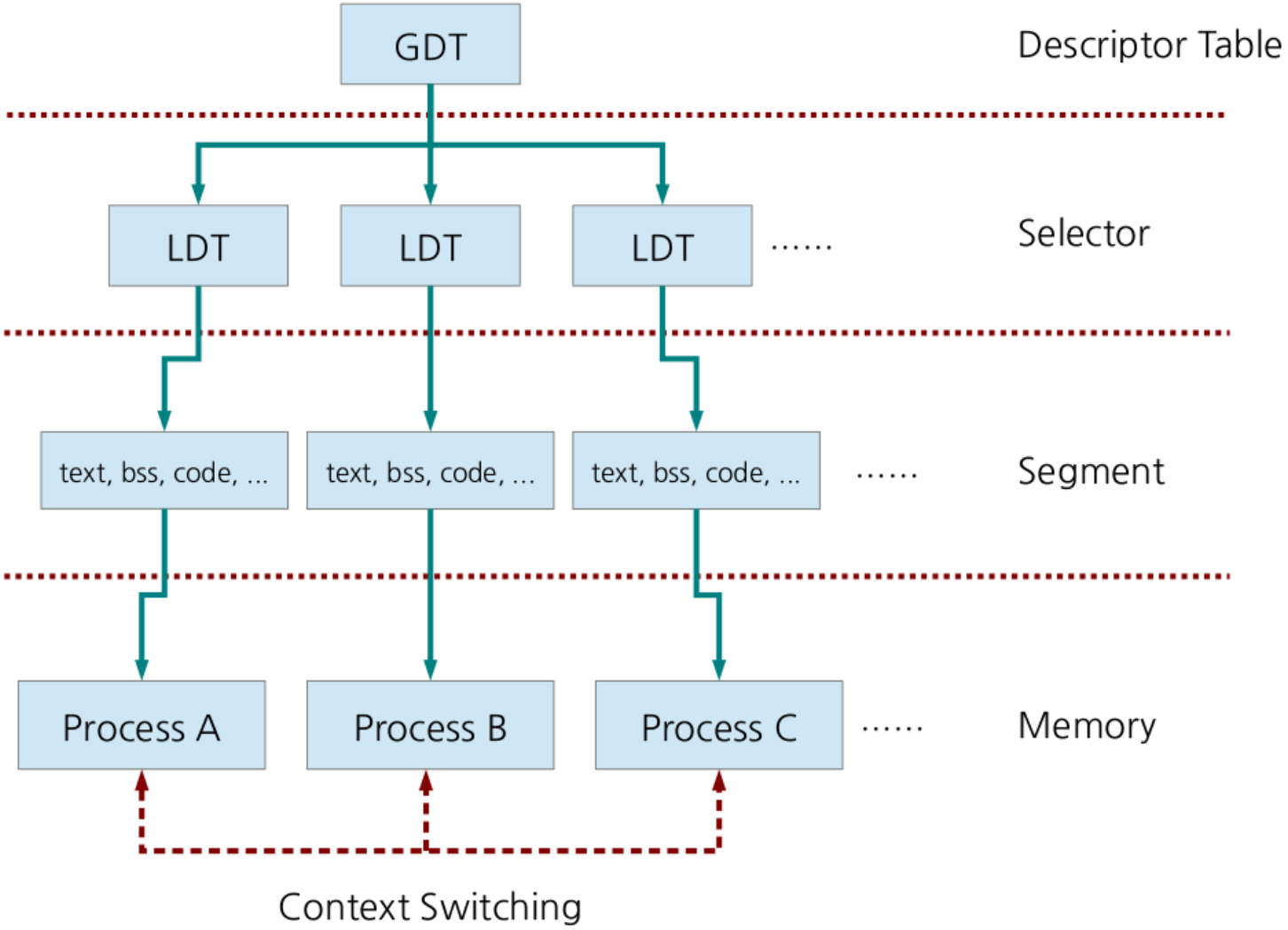
Process D

User Process

- LDT
  - ✓ Local Descriptor Table
  - ✓ LDTR Register를 통하여 접근
  - ✓ LDT는 Virtual Memory에서 Application의 Segment 사용 영역을 정의할 때 사용
  - ✓ Context Switching과도 관련되어 있으며 각각의 Segment에 대한 사용 정보를 저장
  - ✓ LDT의 위치를 나타내는 Segment Descriptor를 GDT 내부에 저장한 뒤 해당 Index를 LDTR Register에 넣어주면, CPU는 LDT를 읽어들여 Process의 Segment에 대한 정보에 접근 가능

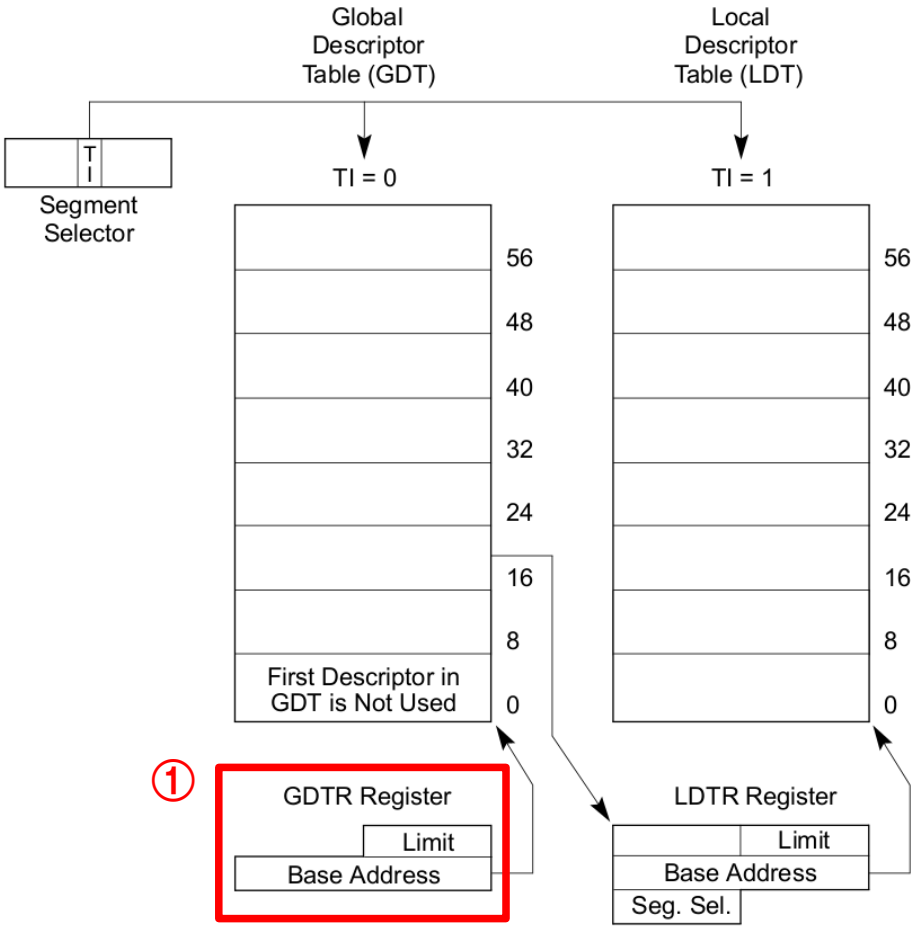
User Process

- 중간정리 - Descriptor Table



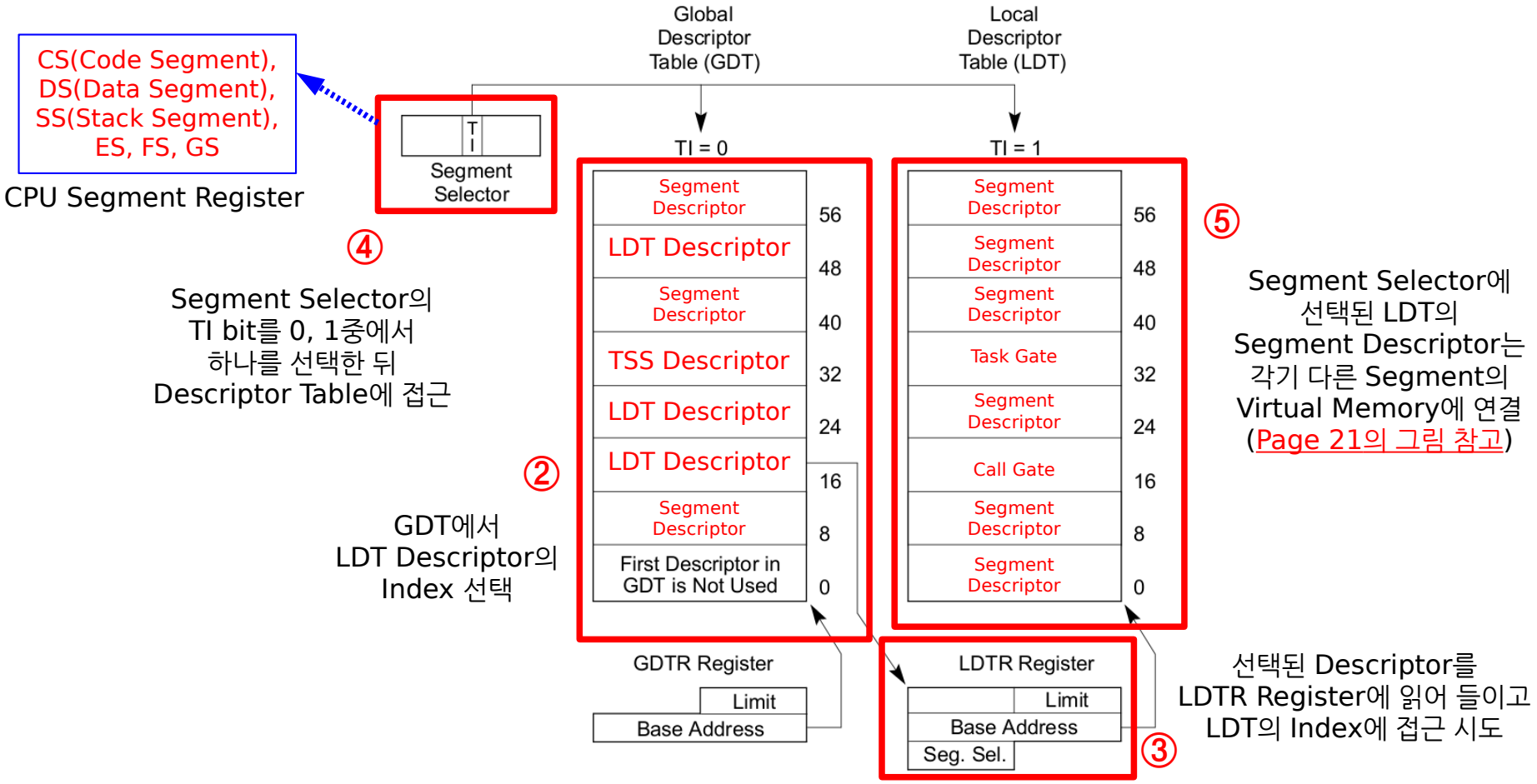
User Process

- GDT & LDT
  - ✓ GDTR Register에 Base Address와 Limit을 입력하여 GDT에 관한 내용을 입력 시킨다. (모든 OS는 1개의 GDT만 소유)



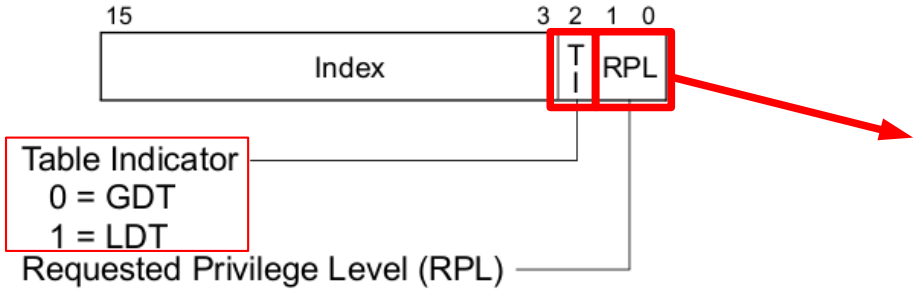
User Process

- GDT & LDT
  - ✓ GDT에는 OS 전체의 Code Segment, BSS Segment 등의 정보 및 LDT Index에 관한 정보가 포함되어 있다. (Process는 각각 1개의 LDT 소유)

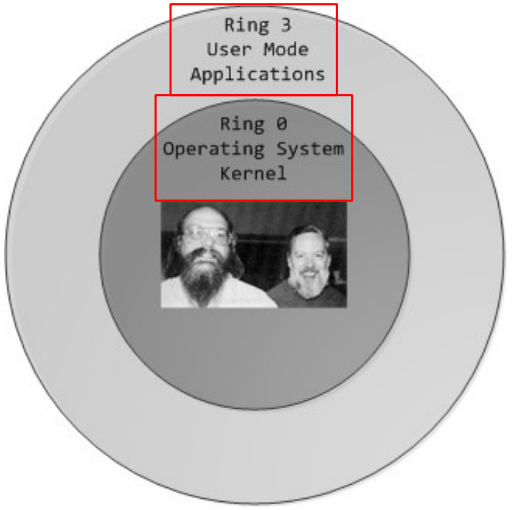


User Process

- GDT & LDT
  - ✓ 모든 Segment Selector는 TI(Table Indicator) bit를 통하여 GDT, LDT 접근 유무를 선택 할 수 있다.



Index Field – Descriptor Table에 대한 Index
TI Field – Table Indicator를 표시
RPL Field – Requestor Privilege Level을 표시 (0, 3)



  - ✓ Index bit에는 현재 선택되어 있는 GDT 또는 LDT의 Index를 입력한다.
  - ✓ TI bit를 0으로 설정해 놓으면 GDT에 대한 접근이 가능하지만, TI bit를 1로 설정해 놓으면 LDTR Register에 Loading되어 있는 LDT에 대한 접근만 가능하다.
    - ▪ 다른 LDT에 접근하고 싶은 경우 LLDT instruction을 사용하여 GDT상의 LDT가 속한 Index를 직접 입력해 주어야 한다.

User Process

- 중간정리 - Process 실행을 위한 GDT, LDT 할당
  - ✓ Process 실행을 위한 초기화 과정은 Software인 Linux Kernel에서 main()을 통하여 Protected Mode로 넘어가기 전에 GDT를 할당하는 것으로 시작한다.

```
void main(void) {
    ....
    go_to_protected_mode();
}
```

(kernel dir)/arch/x86/boot/main.c

```
void go_to_protected_mode(void)
{
    ....
    setup_idt();
    setup_gdt();
    protected_mode_jump(...);
}
```

(kernel dir)/arch/x86/boot/pm.c

```
static void setup_gdt(void) {
    ....
    asm volatile("lgdtl %0" : : "m" (gdt));
}
```

(kernel dir)/arch/x86/boot/pm.c

User Process

- 중간정리 - Process 실행을 위한 GDT, LDT 할당
  - ✓ 또한, 모든 Process의 시작 점인 init Process를 생성함에 있어서 기본적인 LDT Descriptor를 GDT안에 생성하여 fork(), exec()를 통한 Process Copy를 대비한다.

```
GLOBAL(in_pm32)
    # Set up data segments for flat 32-bit mode
    movl    %ecx, %ds
    movl    %ecx, %es
    movl    %ecx, %fs
    movl    %ecx, %gs
    movl    %ecx, %ss
    # The 32-bit code sets up its own stack, but this way we do have
    # a valid stack if some debugging hack wants to use it.
    addl    %ebx, %esp

    # Set up TR to make Intel VT happy
    ltr %di

    # Clear registers to allow for future extensions to the
    # 32-bit boot protocol
    xorl    %ecx, %ecx
    xorl    %edx, %edx
    xorl    %ebx, %ebx
    xorl    %ebp, %ebp
    xorl    %edi, %edi

    # Set up LDTR to make Intel VT happy
    lldt    %cx

    jmpl    *%eax           # Jump to the 32-bit entrypoint
ENDPROC(in_pm32)
```

(kernel dir)/arch/x86/boot/pmjump.S

```
static noinline void __init_refok rest_init(void)
{
    int pid;

    rcu_scheduler_starting();
    /*
     * We need to spawn init first so that it obtains pid 1, however
     * the init task will end up wanting to create kthreads, which, if
     * we schedule it before we create kthreadd, will OOPS.
     */
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
    complete(&kthreadd_done);

    /*
     * The boot idle thread must execute schedule()
     * at least once to get things moving:
     */
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    /* Call into cpu_idle with preempt disabled */
    cpu_startup_entry(CPUHP_ONLINE);
}
```

(kernel dir)/init/main.c

User Process

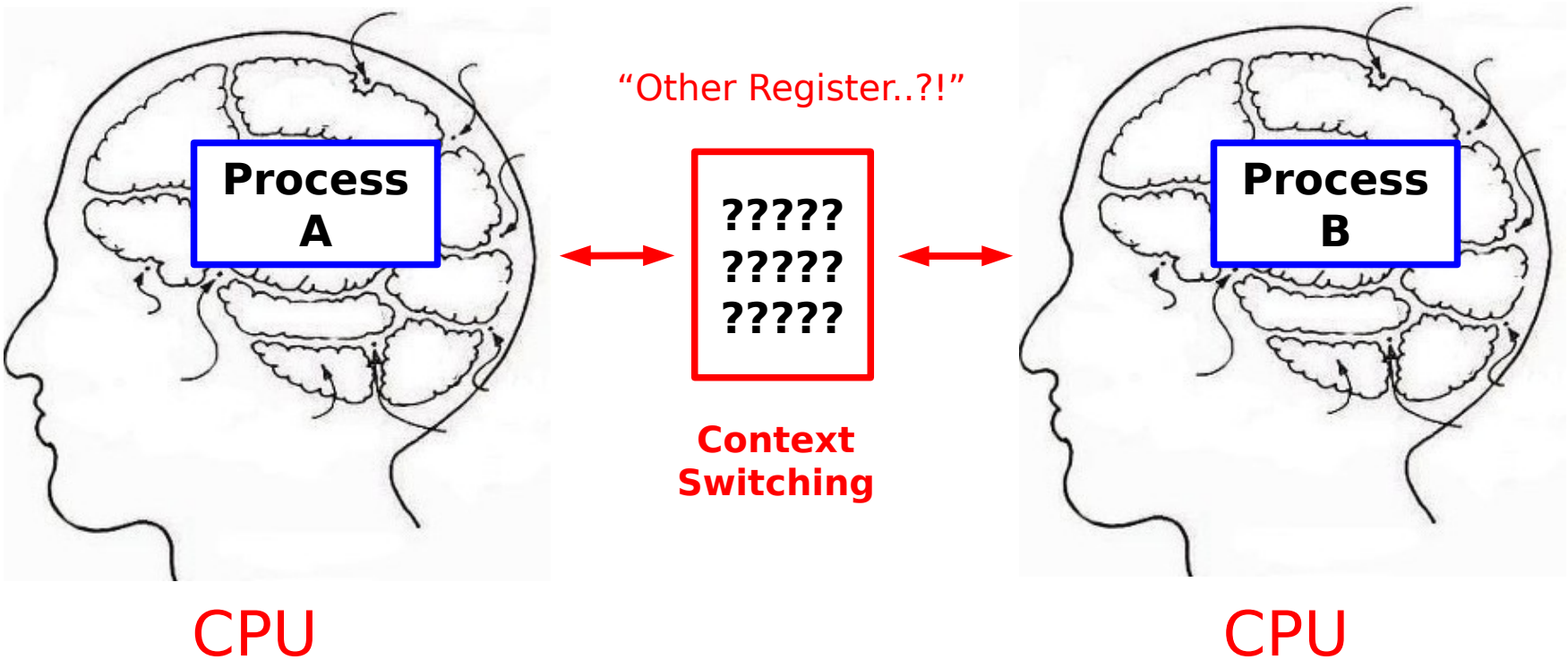
- 중간정리 - Process 실행을 위한 GDT, LDT 할당
  - ✓ 결국, 이렇게 생성된 init Process는 다른 모든 Process의 생성 기반 역할을 수행한다.
    - Linux 에서는 pstree 명령어로 현재 작동되는 Process의 Tree 구조 파악 가능

```
root@sdy-dankook:~# pstree
init─┬─NetworkManager─┬─dnsmasq
     │                └─2*[{NetworkManager}]
     ├─accounts-daemon───{accounts-daemon}
     ├─acpid
     ├─at-spi-bus-laun───2*[{at-spi-bus-laun}]
     ├─atd
     ├─avahi-daemon───avahi-daemon
     ├─bamfdaemon───2*[{bamfdaemon}]
     ├─bluetoothd
     ├─colord───2*[{colord}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd
     ├─2*[dbus-daemon]
     ├─dbus-launch
     ├─dconf-service───2*[{dconf-service}]
     ├─firefox───36*[{firefox}]
```

  - ✓ fork()를 통하여 init Process가 가지고 있던 GDT 내부의 LDT, TSS Descriptor에 대한 Copy를 수행하기 시작하고 각각의 Process는 자신만의 LDT, TSS Descriptor를 가지게 된다.
    - Descriptor를 Copy 한다는 것은 Process의 Segment에 대한 독립적인 관리를 수행한다는 의미와 같다.

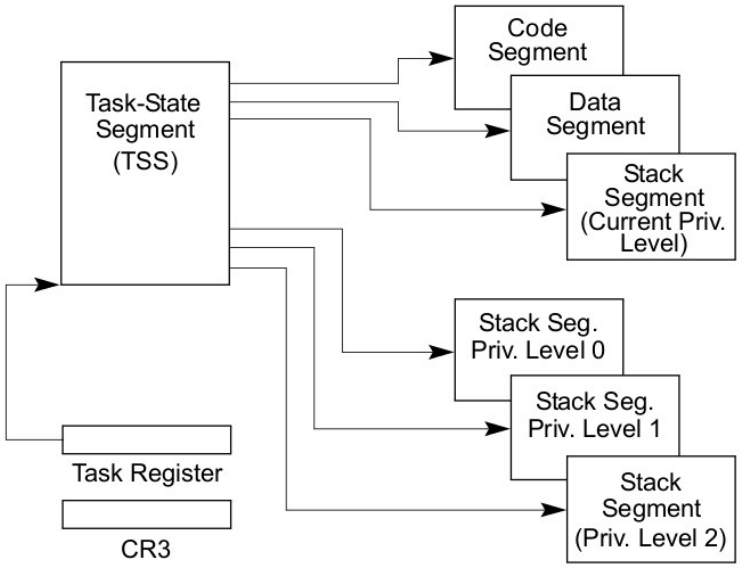
User Process

- Context Switching
  - ✓ Process의 Segment 영역을 선택 가능하게 하는 Segment Register가 있는 반면, CPU가 사용하는 Process를 바꾸는 Context Switching과 관련된 Register도 존재한다.



User Process

- Context Switching
  - ✓ Task Register (TR)
    - TSS (Task State Segment)가 있는 위치를 표시하는 GDT 내의 TSS Descriptor를 가리키는 Selector (= Register)
    - Task State Segment : Process 동작 상태에 관한 정보를 담고 있는 Segment
  - ✓ LTR (Load TR), STR (Save TR) Instruction을 통하여 참조 및 설정
  - ✓ Task State
    - General-Purpose registers
    - Segment registers
    - EFLAGS register
    - CR3 register
    - Task register
    - I/O map base address, I/O map
    - Stack pointers
    - Link to previously executed task



User Process

- Task Register & TSS
  - ✓ TSS의 자료구조

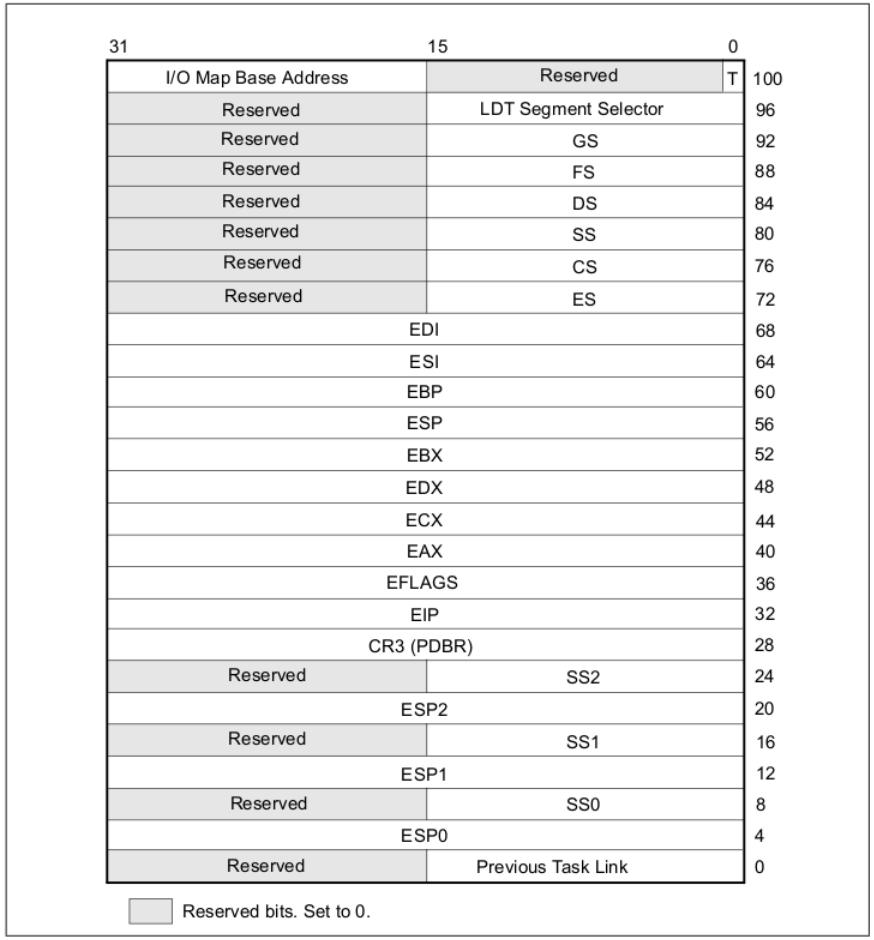


Figure 7-2. 32-Bit Task-State Segment (TSS)

User Process

- Task Register & TSS
  - ✓ Linux Kernel에서 나타내는 자료구조

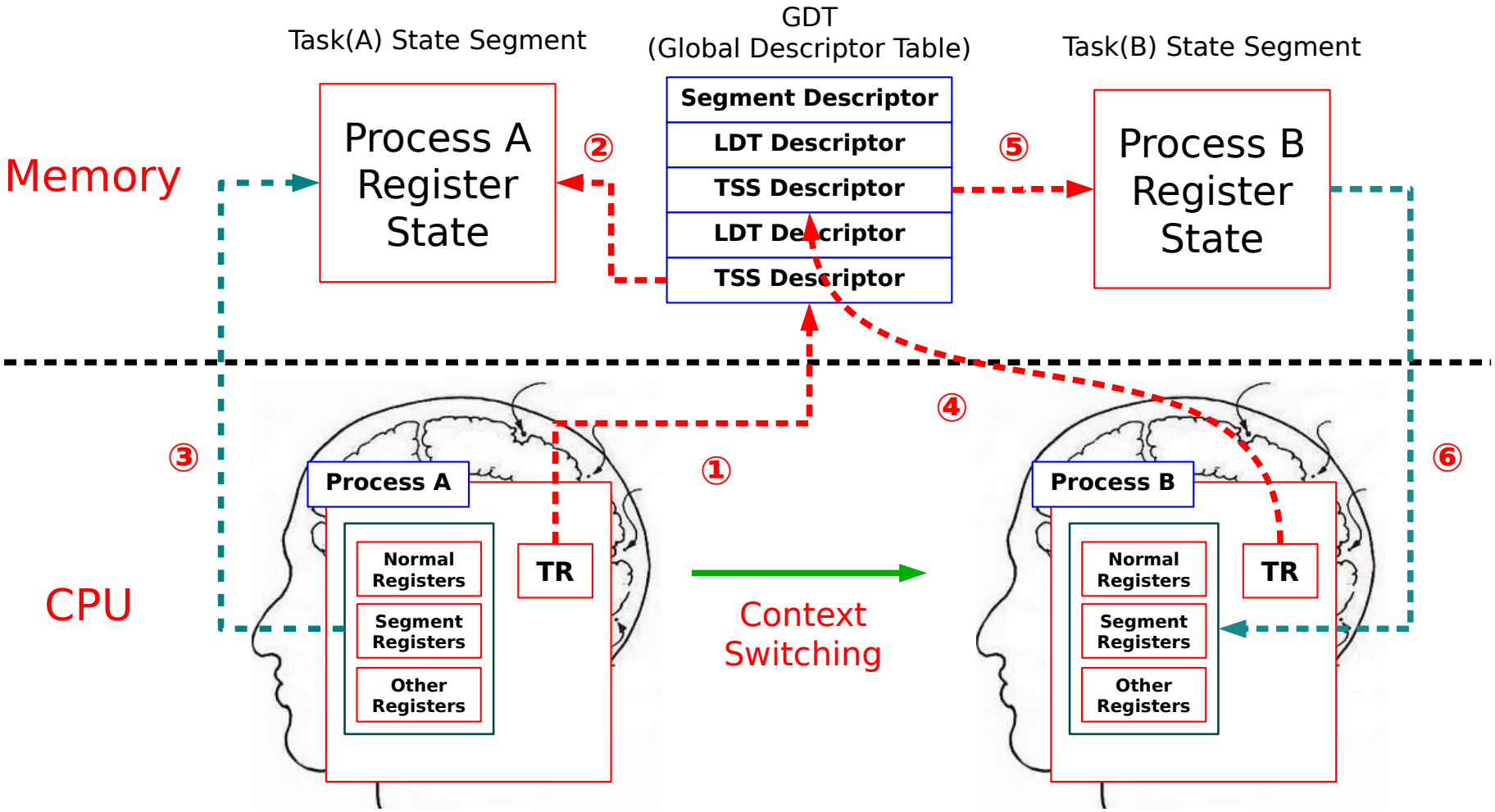
```
struct tss_struct {
    /*
     * The hardware state:
     */
    struct x86_hw_tss   x86_tss;

    /*
     * The extra 1 is there because the CPU will access an
     * additional byte beyond the end of the IO permission
     * bitmap. The extra byte must be all 1 bits, and must
     * be within the limit.
     */
    unsigned long       io_bitmap[IO_BITMAP_LONGS + 1];

    /*
     * .. and then another 0x100 bytes for the emergency kernel stack:
     */
    unsigned long       stack[64];

} ____cacheline_aligned;
```

(kernel dir)/arch/x86/include/asm/processor.h

```
#ifdef CONFIG_X86_32
/* This is the TSS defined by the hardware. */
struct x86_hw_tss {
    unsigned short      back_link, __blh;
    unsigned long       sp0;
    unsigned short      ss0, __ss0h;
    unsigned long       sp1;
    /* ss1 caches MSR_IA32_SYSENTER_CS: */
    unsigned short      ss1, __ss1h;
    unsigned long       sp2;
    unsigned short      ss2, __ss2h;
    unsigned long       __cr3;
    unsigned long       ip;
    unsigned long       flags;
    unsigned long       ax;
    unsigned long       cx;
    unsigned long       dx;
    unsigned long       bx;
    unsigned long       sp;
    unsigned long       bp;
    unsigned long       si;
    unsigned long       di;
    unsigned short      es, __esh;
    unsigned short      cs, __csh;
    unsigned short      ss, __ssh;
    unsigned short      ds, __dsh;
    unsigned short      fs, __fsh;
    unsigned short      gs, __gsh;
    unsigned short      ldt, __ldth;
    unsigned short      trace;
    unsigned short      io_bitmap_base;

} __attribute__((packed));
#else
struct x86_hw_tss {
    u32         reserved1;
    u64         sp0;
    u64         sp1;
    u64         sp2;
    u64         reserved2;
    u64         ist[7];
    u32         reserved3;
    u32         reserved4;
    u16         reserved5;
    u16         io_bitmap_base;

} __attribute__((packed)) ____cacheline_aligned;
#endif
```

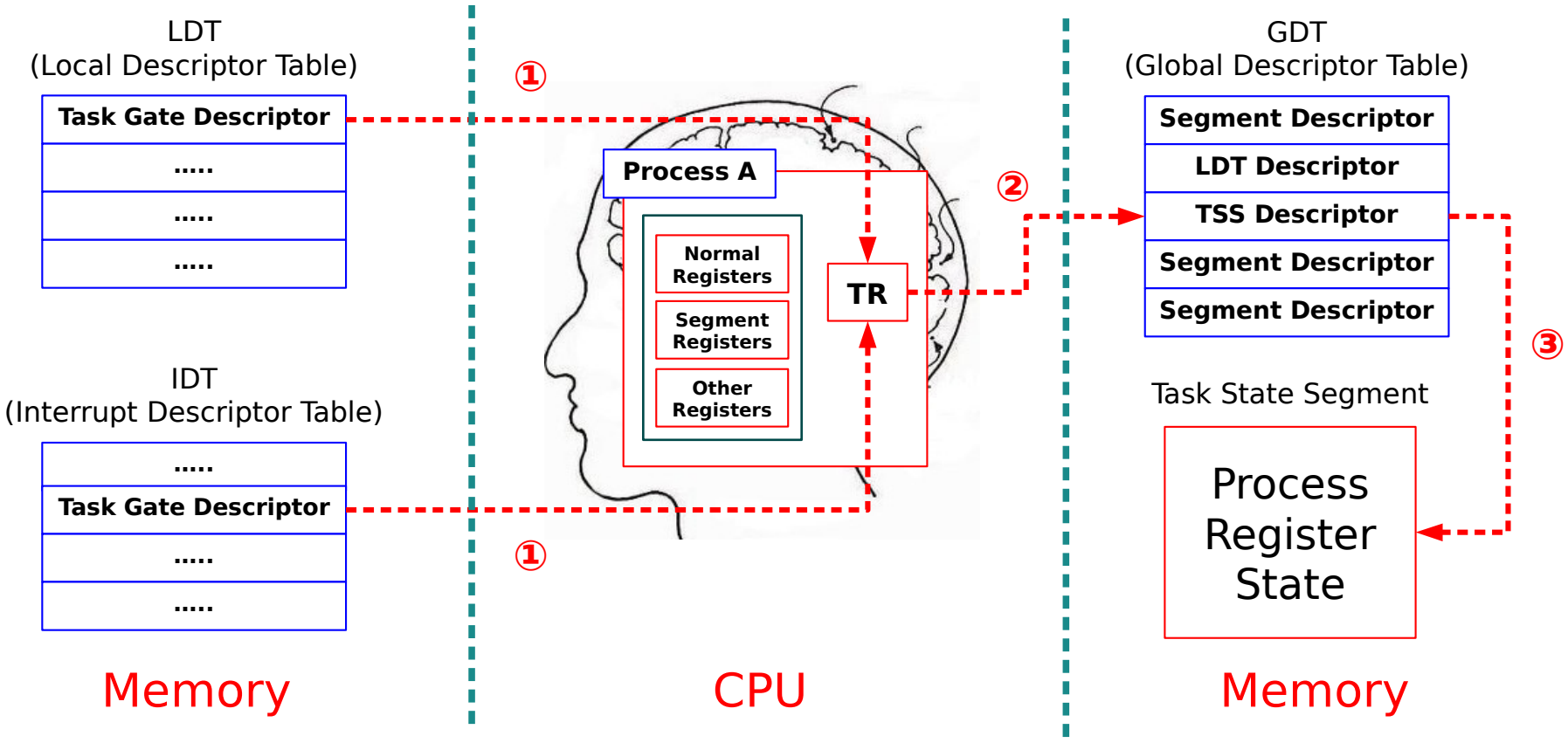
User Process

- Task Register & TSS
  - ✓ Context Switching이 발생하게 되면 다음과 같은 과정이 일어난다.



User Process

- Task Register & TSS
  - ✓ 또한, Task gate에 의해서 Context Switching이 발생하기도 한다.
    - Interrupt, Trap이 발생하였을 때 간접적으로 Context Switching를 발생 시키고, 특정 Task들이 TSS에 접근할 수 있도록 사용 한다.

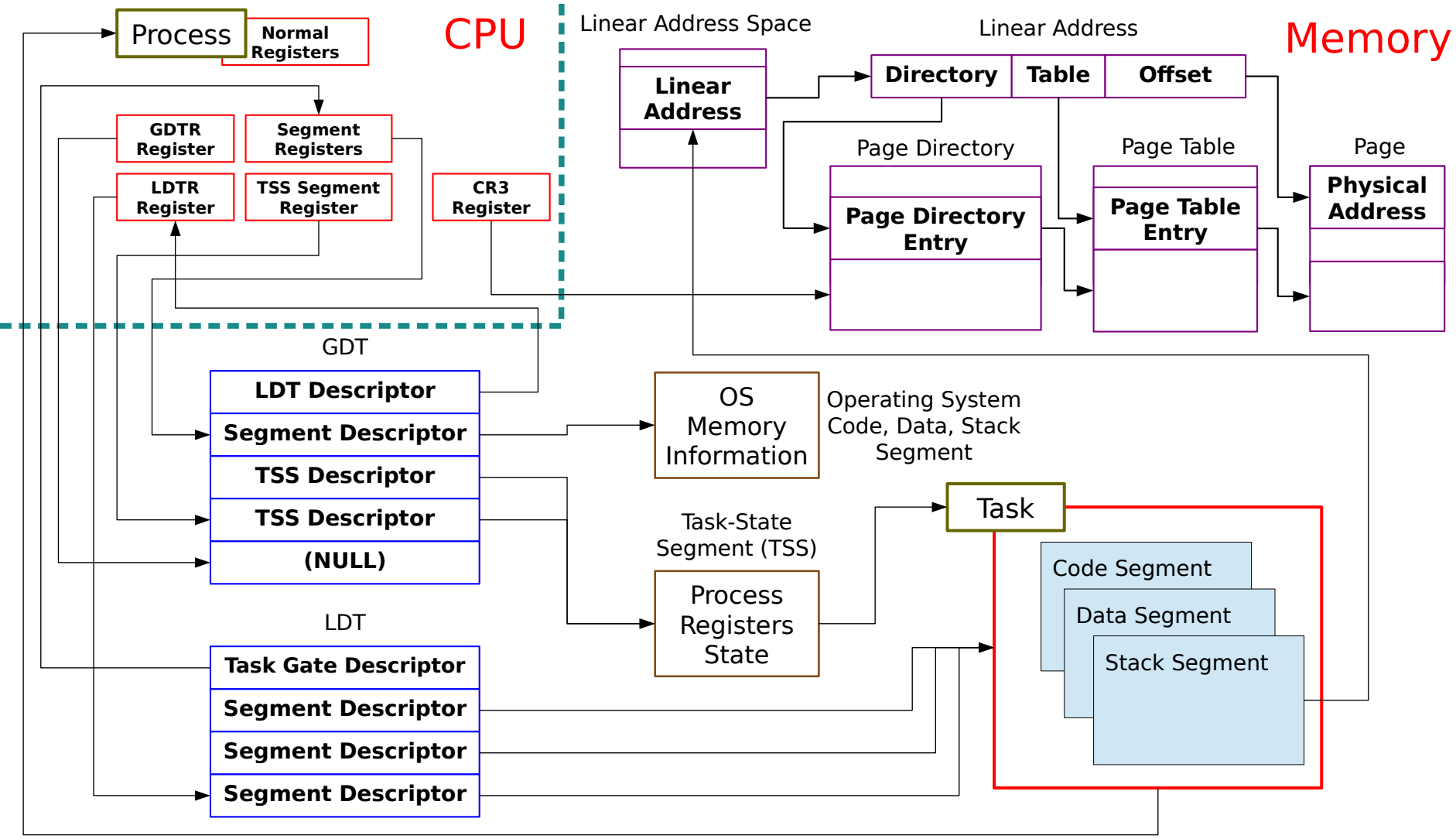


User Process

- Context Switching
  - ✓ GeekOS에서는 Project 2를 통하여 Context Switching을 실습 할 수 있도록 기본 소스를 제공한다.
    - Switch_To_User_Context()
      - (project 2 dir)/src/geekos/user.c
    - Switch_To_Address_Space()
      - (project 2 dir)/src/geekos/userseg.c
    - Spawn_Init_Process()
      - (project 2 dir)/src/geekos/main.c
    - Spawn()
      - (project 2 dir)/src/geekos/user.c
    - Start_User_Thread()
      - (project 2 dir)/src/geekos/kthread.c
    - Setup_User_Thread()
      - (project 2 dir)/src/geekos/kthread.c

User Process

- 중간정리 - GDT & LDT & Context Switching



2nd Homework

- Project 1을 이용하여 ELF File 적재하기
  - ✓ ELF Parsing 과제를 수행할 Project 1을 생성한다.
  - ✓ 생성된 Project는 ELF File을 Parsing 하는 부분이 제외된 상태로 나타난다.
  - ✓ 각 소스에는 TODO라는 이름의 함수가 적힌 부분이 있는데 해당 부분이 직접 채워 넣어야 하는 부분이다.
  - ✓ 완성된 과제는 교수님과 실습조교에게 E-mail로 보내도록 한다.
    - 첨부 내용 : 작성한 Source (Project 단위로 보낼 것), Screenshot