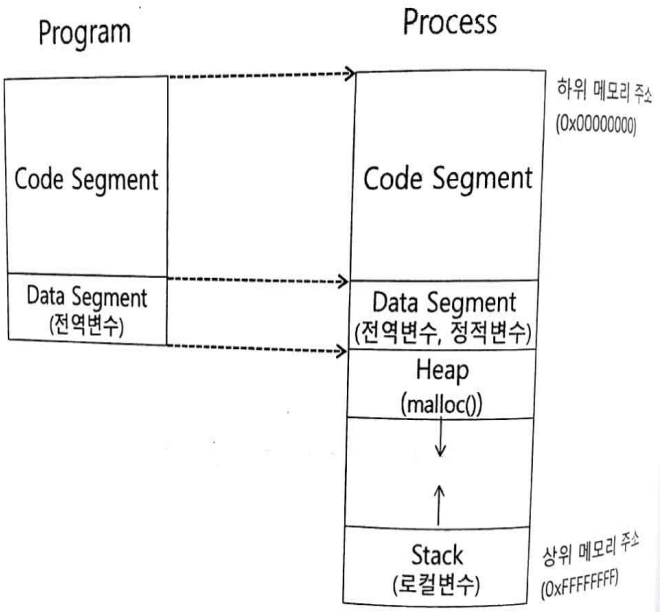
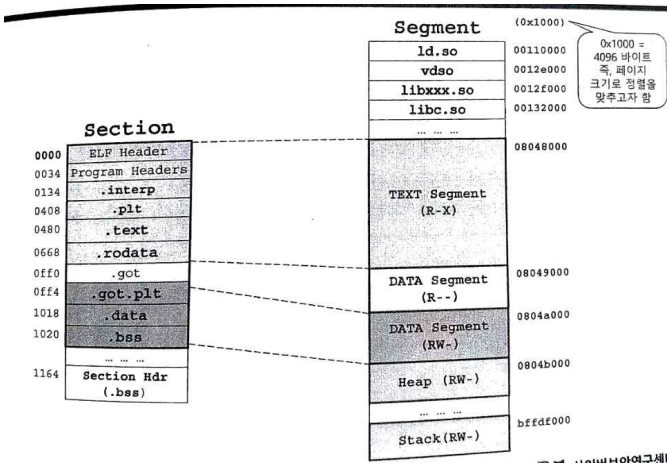
Chapter 4 : 프로세스 공간 관리

4.1 리눅스 프로세스 주소 공간

프로세스의 구조



프로그램 로딩과 메모리 매핑

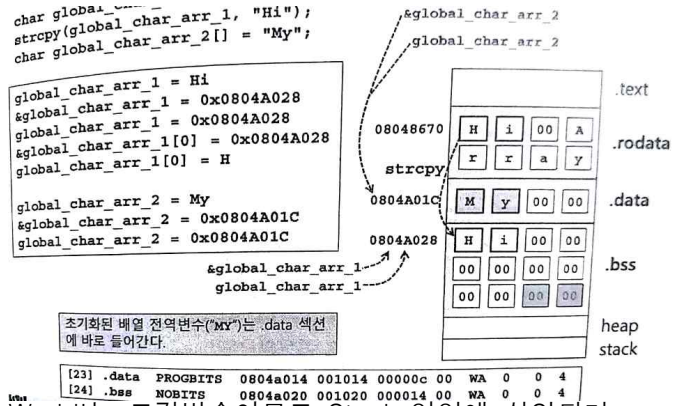


실습 : 배열 주소 공간

maptest2.c

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

char global_char_arr_1[10];
char global_char_arr_2[] = "My";

void main(){
  char local_char_arr_1[10];
  char local_char_arr_2[] = "World!";

  strcpy(global_char_arr_1, "Hi");
  strcpy(local_char_arr_1, "Array");
```

위 코드의 경우, 메모리 공간 할당은 아래와 같다.

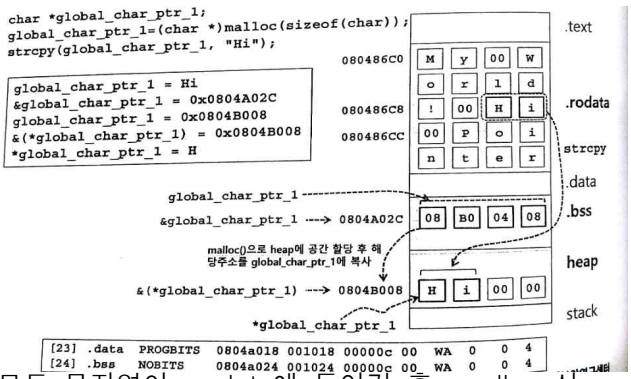


World!는 로컬변수이므로 Stack 영역에 삽입되며, My는 바로 .data안에 삽입된다.

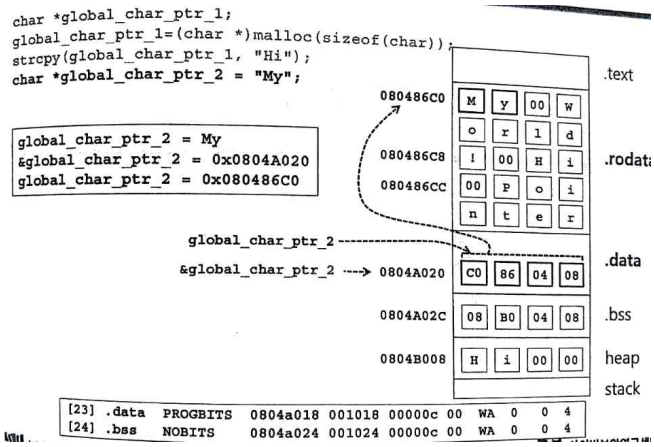
실습 : 포인터 주소공간

```
char *global_char_ptr_1;      char *global_char_ptr_2 = "My";

void main(){
  char *local_char_ptr_1;     char *local_char_ptr_2 = "World!";

  global_char_ptr_1=(char *)malloc(sizeof(char));
  strcpy(global_char_ptr_1, "Hi");

  local_char_ptr_1=(char *)malloc(sizeof(char));
  strcpy(local_char_ptr_1, "Pointer");
```

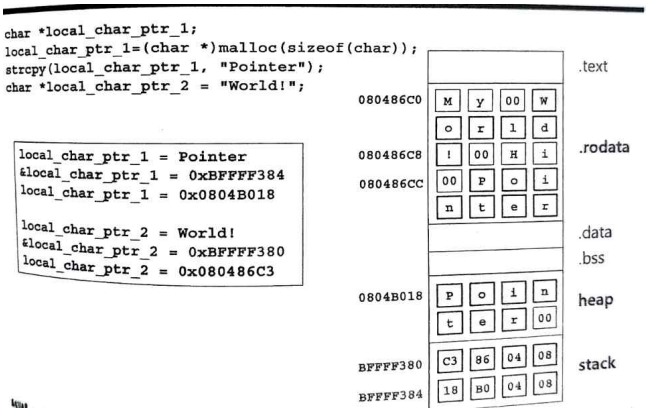
위 코드의 경우, 포인터 주소 할당은 다음과 같다.



모든 문자열이 .rodata에 들어간 후, malloc 시 .bss에 동적할당된 heap 영역의 주소가 들어간 후, 그 주소의 위치에 문자열을 복사한다.



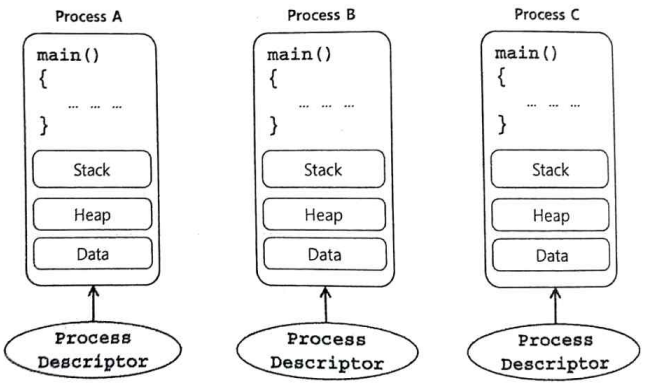
.rodata의 My를 가르키는 Pointer가 .data에 저장된다.



동적할당된 위치에 .rodata의 pointer 문자열이 저장되어, heap영역에 저장되고, stack영역에는 동적할당된 주소값 자체가 저장된다.

4.2 프로세스 생성 관리

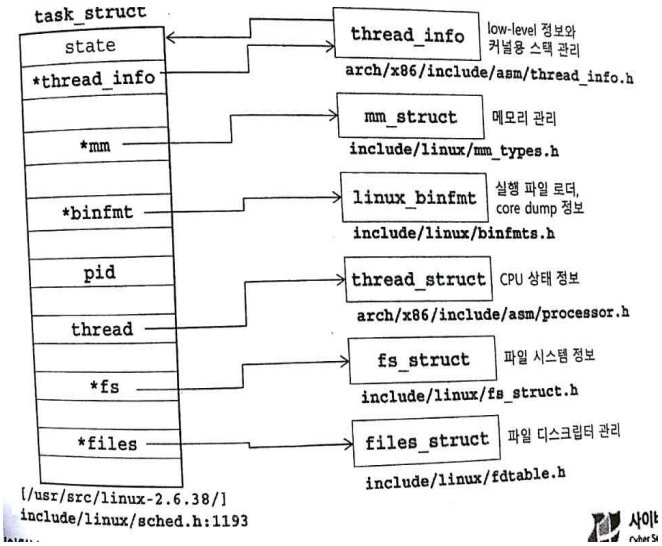
프로세스 생성



모든 프로세스는 코드,데이터, 스택의 주소 공간과 프로세스가 동작할 때의 레지스터 값등의 Context를 가지고 있음

커널 관점에서 프로세스 생성이란 Context를 관리하기 위한 프로세스 디스크립터를 생성하는 것을 의미

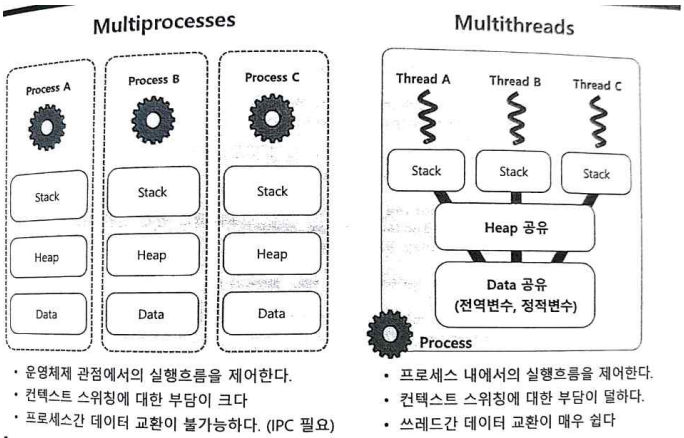
프로세스 디스크립터



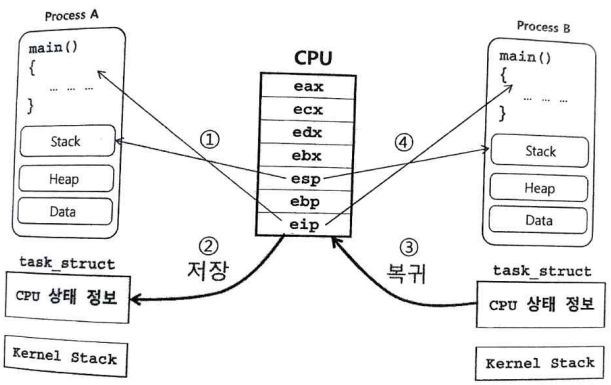
커널 쓰레드

- 커널에서만 동작
- Context Switching 불필요 (직접 매핑)
- kernel_thread() 함수 사용

멀티프로세스 vs 멀티쓰레드

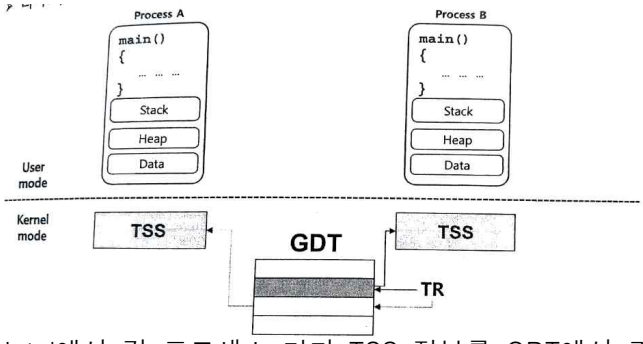


Context Switching (SW 방식)



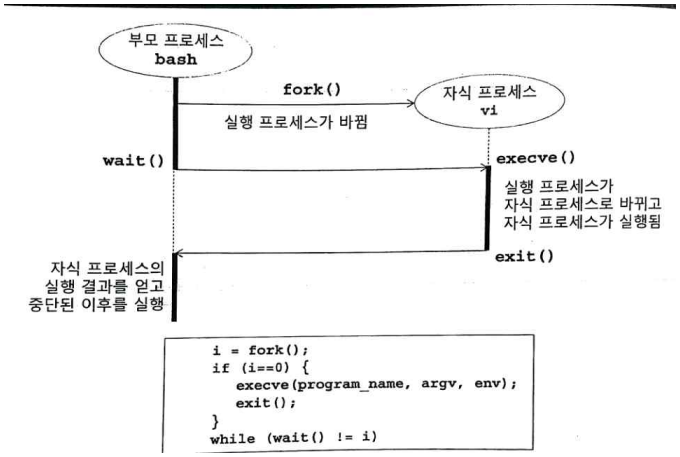
프로세스 A와 관련된 CPU 레지스터 값들을 task struct에 저장하고 프로세스 B의 task_struct에서 저장하고 있는 레지스터 정보를 CPU에 복사함

Context Switching (HW 방식)

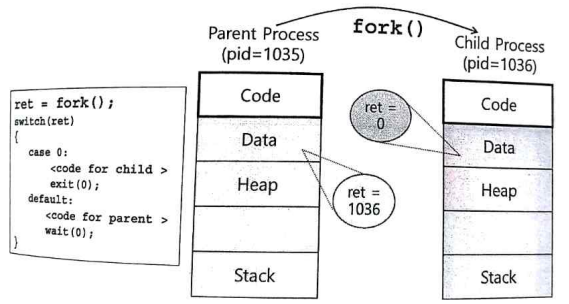


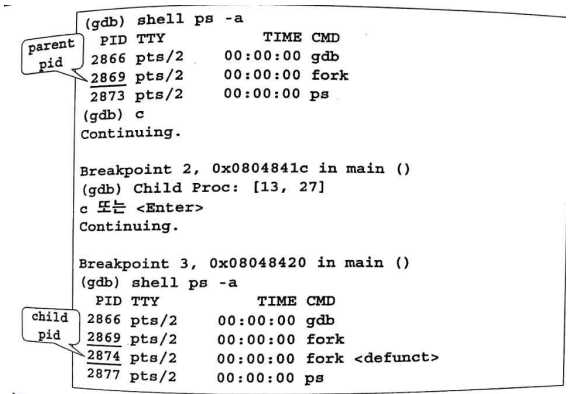
Intel에서 각 프로세스 마다 TSS 정보를 GDT에서 관리하고 Task Register에서 해당 TSS를 변경하여 전환 처리할 수 있도록 제시하였으나, 리눅스와 윈도우는 SW방식의 Context Swithching을 사용한다.

프로세스 라이프 사이클



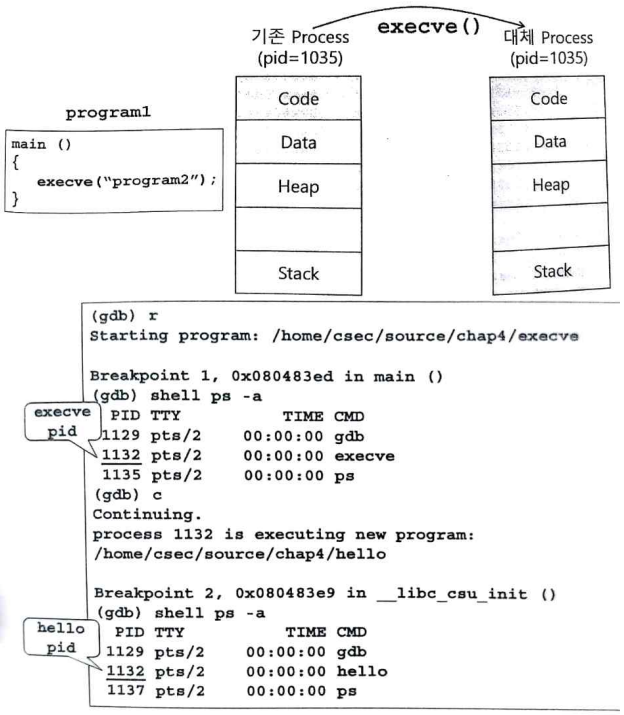
시스템 콜 : Fork()





Fork 실행 시, 새로운 프로세스가 실행되고, 다른 pid 값을 가진다.

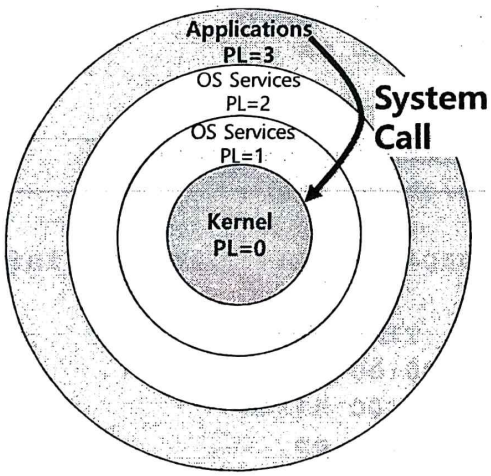
시스템 콜 : execve()



execve같은 경우에는 pid는 그대로인 상태로 프로세스 자체가 바뀐다.

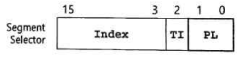
4.3 Segment Protection

Privilege Level (PL)

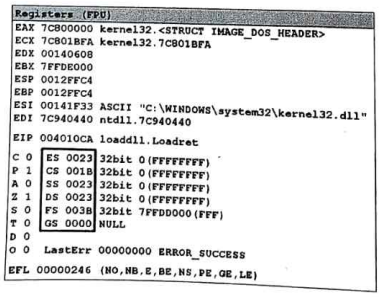


원래는 4개의 링 구조를 지원하지만, 대부분의 OS는 호환성을 위해 커널모드 PL = 0 과 사용자모드 PL = 3만 사용한다.

Privilege Level 확인

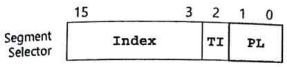


➢ kernel32.dll의 권한 = 3 (사용자 모드 코드)

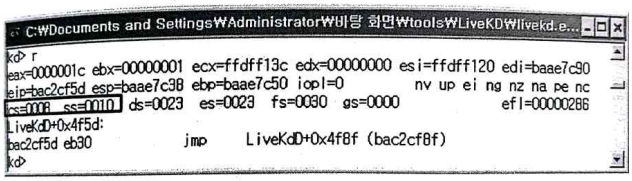


0x0023 = 00000000 00100011
0x001B = 00000000 00011011

CS값과 SS값의 비트 스트링의 제일 뒤 2비트가 11일 경우 사용자 프로그램, 00일 경우 커널이다.

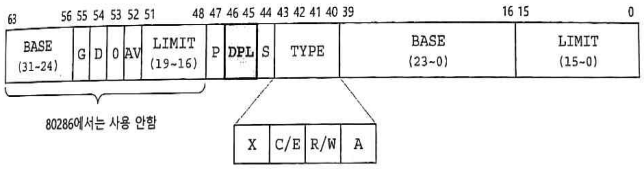


➢ livekd.exe의 권한 = 0 (커널 모드 코드)



0x0008 = 00000000 00001000
0x0010 = 00000000 00010000

Global Descriptor 구조

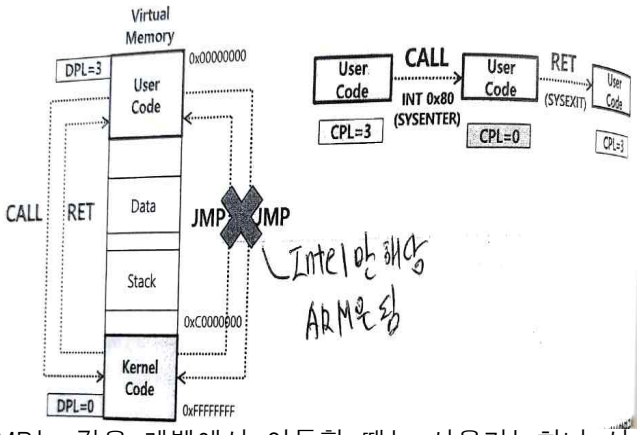


속성	설명
LIMIT	세그먼트의 크기를 나타냄 (G 비트=1일 경우 4KB 곱하여 계산)
BASE	세그먼트의 시작주소를 뜻함
TYPE	세그먼트의 타입으로 코드 또는 데이터 세그먼트로 설정 가능 (세부 설정 값은 다음 페이지 참조)
S	디스크립터 타입(1: 세그먼트 디스크립터, 0: 시스템 디스크립터)
DPL	세그먼트의 권한을 의미(11: 사용자 레벨 , 00: 커널 레벨)
P	Present의 의미로 현재 디스크립터가 유효한지 표시 (1: 유효함, 0: 유효하지 않음)
AV	Available의 약자로 커널이 임의의 용도로 사용할 수 있는 영역
D	Default 세그먼트 길이 (1: 32비트, 0: 16비트)
G	LIMIT에 곱해질 가중치(Granularity)를 정함(1: 4KB, 0: 기본 Byte단위로 곱하지 않음)

Privilege Level 종류

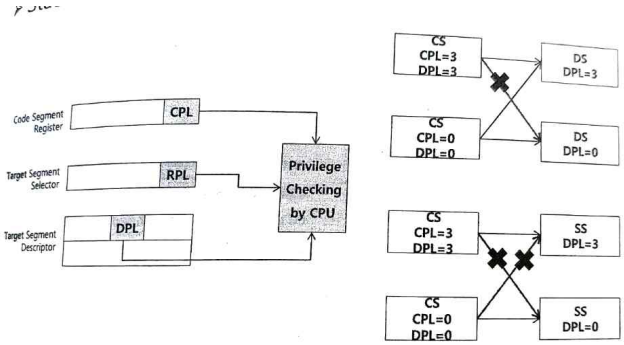
- CPL(Current Privilege Level)
  - 현재 실행중인 프로세스의 PL
  - CS Register의 PL
- RPL(Requested Privilege Level)
  - 타겟 세그먼트의 PL
- DPL(Descriptor Privilege Level)
  - 메모리를 차지하고 있는 타겟 세그먼트의 PL
  - Descriptor 생성 시 설정

시스템 콜에 의한 CPL 변경



JMP는 같은 레벨에서 이동할 때는 사용가능하나 서로 다른 권한 사이로는 이동할 수 없다.
하위 권한에서 상위 권한 영역 접근은 시스템 콜에 의해 가능하며,DPL은 변동없고 CPL 값만 변경된다.

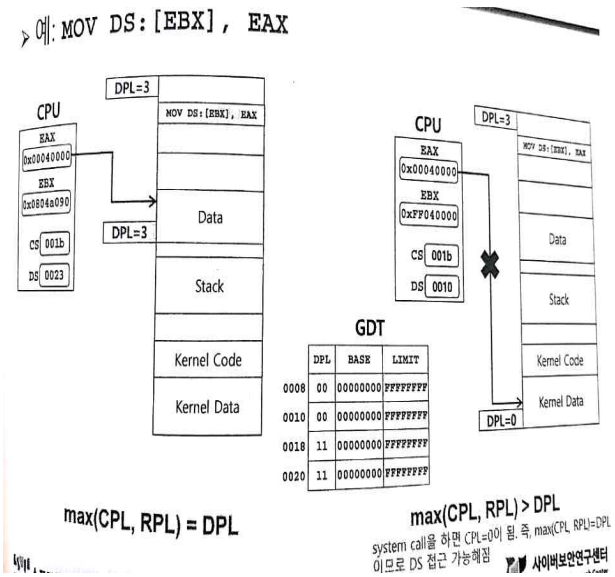
Privilege Level Checking

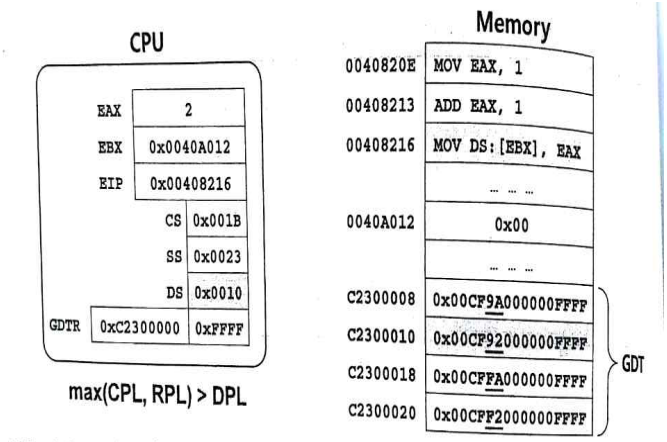


Data 세그먼트(DS, ES, FS, GS)
- max(CPL,RPL) <= DPL

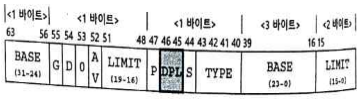
Stack 세그먼트(SS)
- CPL = RPL = DPL

Privilege Level Checking(DS)





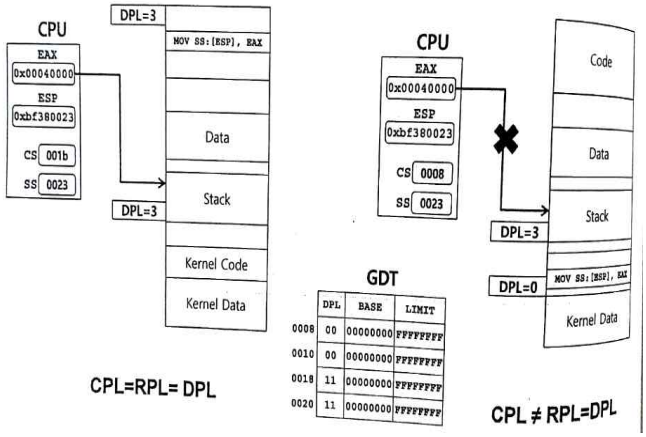
CPL: 0x1B=00011011 → 3
RPL: 0x10=00010000 → 0
DPL: 0x92=10010010 → 0
- 0xC2300000 + 2*8 (=16=0x10)= 0xC2300010
- 0xC2300010번지의 DPL = 0x92 = 10010010



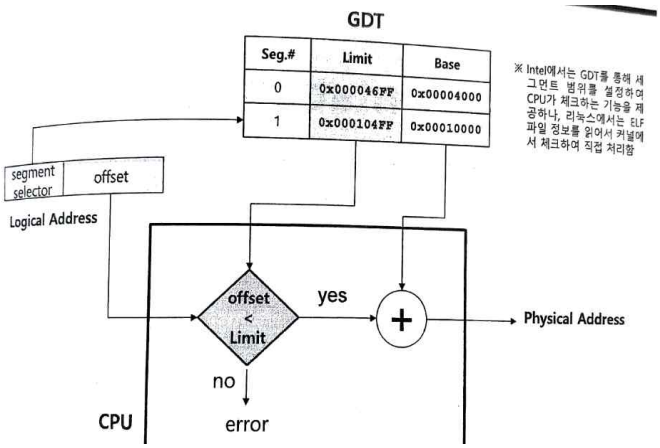
DPL은 앞에서 3번째 바이트의 2~3비트에 들어있다.

Privilege Level Checking(SS)

➢ 예: MOV SS:[ESP], EAX



Limit Checking



Type Checking

Loading

- CS에는 코드 세그먼트만 로딩 가능
- DS,ES,FS,GS에는 코드 세그먼트 로딩 불가
- SS는 쓰기가 허용된 세그먼트만 로딩 가능

Accessing

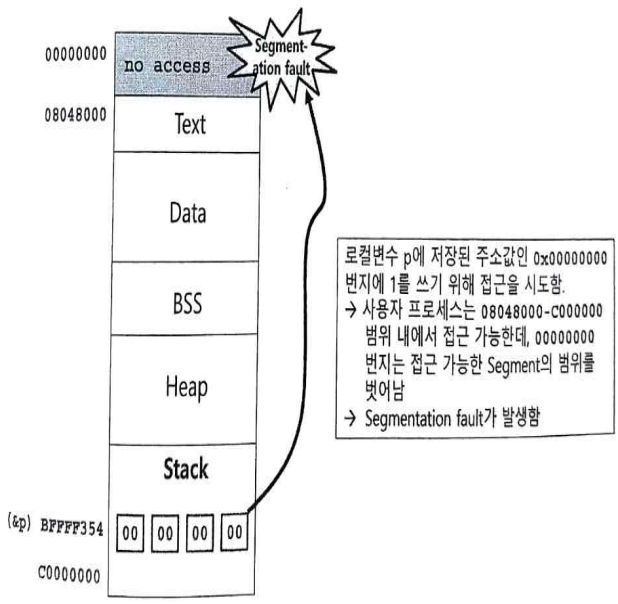
- 코드 또는 Read-only데이터 세그먼트에는 쓰기 금지
- Non-readable 코드 세그먼트에는 접근 금지

4.3 Segment Fault

실습 : SegFault1.c

segfault1.c

```
#include <stdio.h>

void main(){
 int *p;
 *p = 1;
 printf("*p = %d\n", *p);
 return;
}
```



실습 : SegFault3.c

segfault3.c

```
#include <stdio.h>

void main(){
    int *p;
    p = (int*)0x08049004;
    *p = 1;
    printf("*p = %d\n", *p);
    return;
}
```

```
csec@syssec:~/source/chap4$ gdb segfault3
(gdb) l
1       #include <stdio.h>
2       void main(){
3        int *p;
4        p = (int*)0x08049004;
5        *p = 1;
6        printf("*p = %d\n", *p);
7        return;
8       }
(gdb) b 5
(gdb) r
(gdb) shell ps -a
  PID TTY          TIME CMD
26516 pts/0    00:00:00 gdb
26523 pts/0    00:00:00 segfault3
26526 pts/0    00:00:00 ps
(gdb) shell cat /proc/26523/maps
08048000-08049000 r-xp 00000000 08:01 933741 /home/csec/source/chap4/segfault3
08049000-0804a000 r--p 00000000 08:01 933741 /home/csec/source/chap4/segfault3
0804a000-0804b000 rw-p 00001000 08:01 933741 /home/csec/source/chap4/segfault3
b7fef000-b7ff0000 rw-p 00000000 00:00 0
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0      [stack]
```

포인터 p가 가리키는 0x08049004에 해당하는 08049000-0804a000 범위는 읽기(r)만 가능한데 이 Segment에 쓰기(w) 시도

→ Segmentation Fault 발생!

실습 : SegFault4.c

segfault4.c

```
#include <stdio.h>

void main(){
    int *p;
    p = (int*)0x0804a004;
    *p = 1;
    printf("*p = %d\n", *p);
    getchar();
    return;
}
```

```
csec@syssec:~/source/chap4$ ps -a
  PID TTY          TIME CMD
26710 pts/0    00:00:00 segfault4
26711 pts/3    00:00:00 ps

csec@syssec:~/source/chap4$ cat /proc/26710/maps
08048000-08049000 r-xp 00000000 08:01 933742  /home/csec/source/chap4/segfault4
08049000-0804a000 r--p 00000000 08:01 933742  /home/csec/source/chap4/segfault4
0804a000-0804b000 rw-p 00001000 08:01 933742  /home/csec/source/chap4/segfault4
b7fef000-b7ff0000 rw-p 00000000 00:00 0
b7ffc000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0       [stack]
```

- 포인터 p가 가리키는 0x0804a004은 쓰기(w) 가능 → segmentation fault 발생 안함

실습 : SegFault5.c

segfault5.c

```
#include <stdio.h>

void main(){
    int *p;
    p = (int*)0x0028c004;
    *p = 1;
    printf("*p = %d\n", *p);
    getchar();
    return;
}
```

- 프로그램이 실행중인 상태에서 다른 터미널에서 프로세스 맵 확인

```
csec@syssec:~/source/chap4$ ps -a
  PID TTY          TIME CMD
  493 pts/5    00:00:00 segfault5
  495 pts/2    00:00:00 ps
csec@syssec:~/source/chap4$ cat /proc/493/maps
00110000-0012c000 r-xp 00000000 08:01 918326  /lib/i386-linux-gnu/ld-2.13.so
0012c000-0012d000 r--p 0001b000 08:01 918326  /lib/i386-linux-gnu/ld-2.13.so
0012d000-0012e000 rw-p 0001c000 08:01 918326  /lib/i386-linux-gnu/ld-2.13.so
0012e000-0012f000 r-xp 00000000 00:00 0       [vdso]
0012f000-00289000 r-xp 00000000 08:01 918339  /lib/i386-linux-gnu/libc-2.13.so
00289000-0028a000 ---p 0015a000 08:01 918339  /lib/i386-linux-gnu/libc-2.13.so
0028a000-0028c000 r--p 0015a000 08:01 918339  /lib/i386-linux-gnu/libc-2.13.so
0028c000-0028d000 rw-p 0015c000 08:01 918339  /lib/i386-linux-gnu/libc-2.13.so
0028d000-00290000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 934994  /home/csec/source/chap4/segfault5
08049000-0804a000 r--p 00000000 08:01 934994  /home/csec/source/chap4/segfault5
0804a000-0804b000 rw-p 00001000 08:01 934994  /home/csec/source/chap4/segfault5
b7fef000-b7ff0000 rw-p 00000000 00:00 0
b7ffc000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0       [stack]
```

- 포인터 p가 가리키는 0x0028c004은 공유라이브러리 libc.so 영역
  - → 쓰기(w) 가능하여 segmentation fault 발생 안함
  - → Copy on Write 메커니즘에 의해 libc.so의 물리 메모리에는 영향을 주지 않음

write시 커널에서 copy해서 나한테 독립 라이브러리로 빌리

Chapter 5 : 프로세스 공간 관리

5.1 Assembly 개요

어셈블리어

어셈블리어

- 기계어의 비트 형식을 Mnemonic code로 나타냄

Mnemonic code

- 기계어의 비트 형식이 나타내는 의미를 symbol로 표현한 것으로 프로그램을 이해하거나 작성하기 쉽다.
- 예 : MOV EBX, 03h

어셈블리 프로그래밍을 하는 이유

- 컴퓨터 하드웨어의 구성 요소에 직접 엑세스할 때
- 컴파일러를 설계하거나 시스템 프로그램을 작성하려고 할 때
- 빠른 수행이 필요한 프로그램을 작성하려고 할 때
- 기억 장소를 적게 차지하거나 입출력 장치를 보다 효율적으로 사용하려는 경우

고급언어 프로그램의 입출력 접근

Level 3 고급 언어 프로그램

∨

Level 2 OS Function

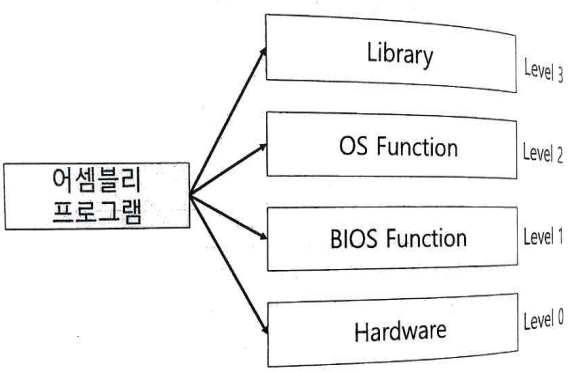
∨

Level 1 BIOS Function

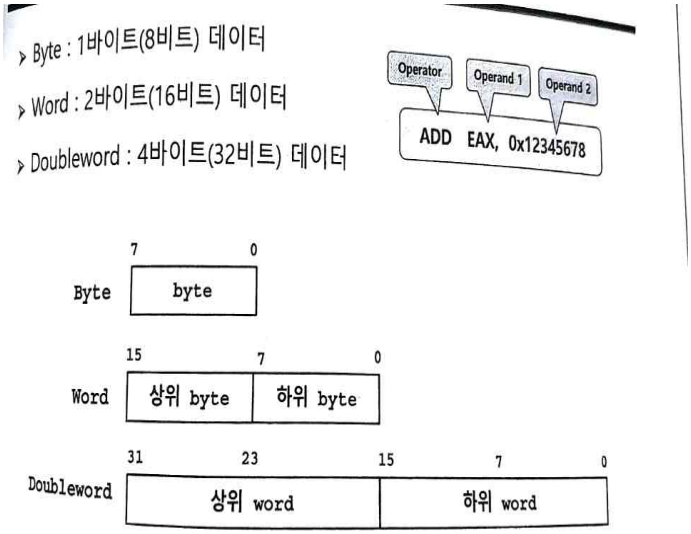
∨

Level 0 Hardware

어셈블리 프로그램의 입출력 접근



어셈블리 데이터 타입



어셈블리 문법

	Intel 문법	AT&T 문법
사용대상	윈도우	리눅스
방향	Operator dest, src	Operator src, dest

Prefix	mov eax, 1
	mov ebx, 0xff
	int 80h
		movl $1, %eax
		movl $0xff, %ebx
		int $0x80

Memory Operand	mov eax, [ebx+3]	movl 3(%ebx), %eax
	SegReg : [base+index*scale+disp]	%SegReg : disp(base, index, scale)

Suffix	mov al, bl
	mov ax, bx
	mov eax, ebx
	mov dword ptr ss:[esp], 0x12345678
		movb %bl, %al
		movw %bx, %ax
		movl %ebx, %eax
		movl $0x12345678, (%esp)