

boot1.asm소스 분석

org 0x9000 ;메모리 9000번지에서 시작한다.

[BITS 16] ;16비트단위로 데이터처리한다는 것을 알려준다.

cli ; Clear Interrupt Flag, 인터럽트가 발생안되게 설정하는 코드이다.

mov ax, 0xb800 ;비디오 메모리의 시작주소(0xb800)를 ax에 저장하여 ax레지스터를 비디오메모리로 사용
mov es, ax ;ES레지스터에 ax값을 넣는다.
mov ax, 0x00 ;ax레지스터를 0으로 초기화한다.
mov bx, 0 ;bx레지스터를 0으로 초기화한다.
mov cx, 80*25*2 ;cx레지스터에 80*25*2(가로*세로*2byte[글자])값을 저장한다.

CLS:

mov [es:bx], ax ;비디오메모리의 주소에 0으로 초기화된 ax의 값을 집어넣어 문자를 삭제한다.
add bx, 1 ;bx를 1씩 증가시킨다.
loop CLS ;CLS레이블로 돌아가 루프를 도는데 루프의 횟수는 cx에 저장된 값이다.

Initialize_PIC:

;ICW1 - 두 개의 PIC를 초기화

mov al, 0x11
out 0x20, al
out 0xa0, al

;ICW2 - 발생된 인터럽트 번호에 얼마를 더할지 결정

mov al, 0x20
out 0x21, al
mov al, 0x28
out 0xa1, al

;ICW3 - 마스터/슬레이브 연결 핀 정보 전달

mov al, 0x04
out 0x21, al
mov al, 0x02
out 0xa1, al

;ICW4 - 기타 옵션

mov al, 0x01
out 0x21, al
out 0xa1, al

mov al, 0xFF
out 0x21, al
out 0xa1, al

Initialize_Serial_port:

```
xor      ax, ax ;ax를 0으로 초기화한다.
xor      dx, dx ;dx를 0으로 초기화한다.
mov      al, 0xe3 ;al에 0xe3값을 넣는다.
int      0x14 ;시리얼포트 서비스를 위한 0x14 인터럽트를 발생시킨다.
```

READY_TO_PRINT:

```
xor      si, si ;si를 0으로 초기화한다.
xor      bh, bh ;bh를 0으로 초기화한다.
```

PRINT_TO_SERIAL:

```
mov      al, [msgRMode+si] ;al에 [msgRMode+si] 주소에 있는 값을 넣는다.
mov      ah, 0x01 ;ah에 0x01을 넣는다.
int      0x14 ;0x14인터럽트를 발생시킨다.
add      si, 1 ;si의 값을 1 증가시킨다.
cmp      al, 0 ;msgRMode의 끝에 도달하면 출력을 종료한다.
jne      PRINT_TO_SERIAL ;msgRMode의 끝에 도달하지 않으면 PRINT_TO_SERIAL로 돌아감
```

PRINT_NEW_LINE: ;개행을 해주는 레이블이다.

;0x0a(LF)를 시리얼포트에 0x14인터럽트를 통해 전송한다.

```
mov      al, 0x0a
mov      ah, 0x01
int      0x14
```

;0x0d(CR)를 시리얼포트에 0x14인터럽트를 통해 전송한다.

```
mov      al, 0x0d
mov      ah, 0x01
int      0x14
```

; OS assignment 2

; add your code here

; print current date to boch display

Activate_A20Gate:

```
mov      ax, 0x2401
int      0x15
```

;Detecting_Memory:

```
;      mov      ax, 0xe801
;      int      0x15
```

PROTECTED:

```
xor      ax, ax ;ax를 0으로 초기화한다.
mov      ds, ax ;ds에 ax값(0)을 넣는다.
```

call SETUP_GDT ;SETUP_GDT를 호출한다.

;32비트 보호모드 스위치 켜는 부분

;CR(CR0~CR3)이라는 레지스터는 CPU의 상태를 알려주거나 기능을 바꾸기 위한 용도.

;CR0 레지스터의 첫 번째 1비트가 PE비트, 이 비트가 1로 셋팅되면 CPU가 32비트로 동작함 = 보호모드가 시작함.

;그러나 CR0레지스터에 데이터를 바로 기록할 수 없으므로 eax레지스터로 값을 읽어온 후

;or명령어로 PE비트를 켜준 후 다시 CR0레지스터에 넣어준다.

mov eax, cr0

or eax, 1

mov cr0, eax

;윗라인으로 인해 이미 32비트모드로 바뀌었다. 따라서 16비트코드를 실행하면 CPU가 에러낸다.

;그러나 CPU의 읽고 해석하고 실행하는 3파이프라인이 문제가 됨.

;파이프라인에는 아직 16비트 명령어가 남아있을 수 있으므로 nop(아무작업안함)명령어로 이를 방지해줌.

jmp \$+2

nop

nop

;32비트 커널부분으로 점프한다.

jmp CODEDESCRIPTOR:ENTRY32

SETUP_GDT: ;GDT를 등록하는 레이블이다.

lgdt [GDT_DESC] ;lgdt opcode를 이용하여 GDT디스크립터를 로드한다.

ret ;호출한 곳으로 되돌아간다.

[BITS 32] ;이하의 코드는 32bit 코드로 설정한다.

ENTRY32:

mov ax, 0x10 ;보호모드 커널용 데이터 세그먼트 디스크립터를 ax레지스터에 저장.

;세그먼트 레지스터를 초기화하는 부분이다.

mov ds, ax ; DS 세그먼트 셀렉터에 설정한다.

mov es, ax ; ES 세그먼트 셀렉터에 설정한다.

mov fs, ax ; FS 세그먼트 셀렉터에 설정한다.

mov gs, ax ; GS 세그먼트 셀렉터에 설정한다.

; 스택을 0x00000000 ~ 0x0000FFFF 영역에 64KB 크기로 생성한다.

mov ss, ax ; SS 세그먼트 셀렉터에 설정

mov esp, 0xFFFFE ;스택을 관리해주는 esp레지스터를 0xFFFFE로 설정한다.

mov ebp, 0xFFFFE ;스택을 관리해주는 ebp레지스터를 0xFFFFE로 설정한다.

mov edi, 80*2 ;라인 하나의 바이트수를 edi에 저장한다.

lea esi, [msgPMode] ;esi레지스터에 msgPMode의 시작주소를 저장한다.

call PRINT ;PRINT레이블을 호출한다.

:IDT TABLE

cld

```
mov     ax,     IDTDESCRIPTOR
mov     es, ax
xor     eax, eax
xor     ecx, ecx
mov     ax, 256 ;IDT 영역에 256개의 빈 디스크립터를 복사한다.
mov     edi, 0
```

IDT_LOOP:

```
lea     esi, [IDT_IGNORE]
mov     cx, 8 ;디스크립터 하나는 8바이트이다.
rep     movsb
dec     ax
jnz     IDT_LOOP
```

lidt [IDTR]

sti

jmp CODEDESCRIPTOR:0x10000

;함수를 사용할 때 여러 레지스터를 사용하기 때문에 함수를 호출하기 전과 호출한 후의 값이 다르므로
;레지스터들을 스택에 저장한다.

PRINT:

```
push    eax ;eax 레지스터를 스택에 저장한다.
push    ebx ;ebx 레지스터를 스택에 저장한다.
push    edx ;edx 레지스터를 스택에 저장한다.
push    es ;es 레지스터를 스택에 저장한다.
mov     ax, VIDEODESCRIPTOR ;ax에 보호모드 커널용 비디오 디스크립터를 저장한다.
mov     es, ax ;es레지스터에 ax값을 저장한다.
```

PRINT_LOOP:

```
or      al, al ;al의 값이 0이면
jz      PRINT_END ;PRINT_END 레이블로 점프한다.
```

;edi의 값을 하나씩 증가시켜주며 화면에 한 문자씩 출력해준다.

```
mov     al, byte[esi]
mov     byte [es:edi], al
inc     edi
mov     byte [es:edi], 0x07
```

OUT_TO_SERIAL:

```
mov     bl, al ;al의 값을 bl에 저장한다.
```

mov dx, 0x3fd ;dx에 데이터 송수신의 정보를 제공하는 Line status register를 저장한다.

CHECK_LINE_STATUS:

```
in      al, dx ;al에 데이터를 입력한다.
and     al, 0x20 ;al을 0x20과 and연산한다.
cmp     al, 0 ;al에 저장된 값이 0이 아니면
jz      CHECK_LINE_STATUS ;CHECK_LINE_STATUS 레이블로 돌아간다.
mov     dx, 0x3f8 ;dx의 값을 다른 시리얼 포트에 변경한다.
mov     al, bl ;bl에 저장한 값을 al에 저장한다.
out     dx, al ;dx 시리얼 포트에 al의 값을 출력한다.

inc     esi ;esi레지스터를 1씩 증가시킨다.
inc     edi ;edi레지스터를 1씩 증가시킨다.
jmp     PRINT_LOOP ;PRINT_LOOP 레이블로 점프한다.
```

PRINT_END:

LINE_FEED: ;현 라인의 시작점으로 가는 레이블이다.

```
mov     dx, 0x3fd ;dx에 데이터 송수신의 정보를 제공하는 Line status register를 저장한다.
in      al, dx ;al에 데이터를 입력한다.
and     al, 0x20 ;al을 0x20과 and연산한다.
cmp     al, 0 ;al에 저장된 값이 0이 아니면
jz      LINE_FEED ;LINE_FEED 레이블로 돌아간다.
mov     dx, 0x3f8 ;al에 저장된 값이 0이면 0x3f8로 시리얼 포트를 바꿔준다.
mov     al, 0x0a ;al에 0x0a(LF)를 저장한다.
out     dx, al ;dx 시리얼포트에 al의 데이터를 출력한다.
```

CARRIAGE_RETURN: ;다음줄로 넘어가는 레이블이다.

```
mov     dx, 0x3fd ;dx에 데이터 송수신의 정보를 제공하는 Line status register를 저장한다.
in      al, dx ;al에 데이터를 입력한다.
and     al, 0x20 ;al을 0x20과 and연산한다.
cmp     al, 0 ;al에 저장된 값이 0이 아니면
jz      CARRIAGE_RETURN ;CARRIAGE_RETURN 레이블로 돌아간다.
mov     dx, 0x3f8 ;al에 저장된 값이 0이면 0x3f8로 시리얼 포트를 바꿔준다.
mov     al, 0x0d ;al에 0x0d(CR)를 저장하고
out     dx, al ;dx의 시리얼 포트에 al의 데이터를 출력한다.

pop     es ;스택에 저장했던 es레지스터값을 꺼낸다.
pop     edx ;스택에 저장했던 edx레지스터값을 꺼낸다.
pop     ebx ;스택에 저장했던 ebx레지스터값을 꺼낸다.
pop     eax ;스택에 저장했던 eas레지스터값을 꺼낸다.
ret     ;호출해준 위치로 돌아간다.
```

GDT_DESC:

```
dw GDT_END - GDT - 1 ;GDT의 limit
dd GDT                ;GDT의 베이스어드레스
```

GDT:

;하나의 디스크립터는 8바이트이다.
;NULL 디스크립터로 초기화해주는 부분이다.

```
NULLDESCRIPTOR equ 0x00
    dw 0 ;limit 0~15비트
    dw 0 ;베이스 어드레스의 하위 두바이트
    db 0 ;베이스 어드레스 16~23비트
    db 0 ;타입
    db 0 ;limit 16~19비트,플래그
    db 0 ;베이스 어드레스 31~24비트
```

;코드 세그먼트 디스크립터

```
CODEDESCRIPTOR equ 0x08
    dw 0xffff ;limit:0xFFFF
    dw 0x0000 ;base 0~15 : 0
    db 0x00 ;base 16~23 : 0
    db 0x9a ;P:1, DPL:0, Code, non-confirming, readable
    db 0xcf ;G:1, D:1, limit 16~19 bit:0xF
    db 0x00 ;base 24~32: 0
```

;데이터 세그먼트 디스크립터

```
DATADESCRIPTOR equ 0x10
    dw 0xffff ;limit:0xFFFF
    dw 0x0000 ; base 0~15 : 0
    db 0x00 ;base 16~23 : 0
    db 0x92 ;P:1, DPL:0, data, expand-up, readable, writable
    db 0xcf ;G:1, D:1, limit16~19 bit:0xF
    db 0x00 ;base 24~32 : 0
```

;비디오 세그먼트 디스크립터

;시작주소는 0x000B8000

```
VIDEODESCRIPTOR equ 0x18
    dw 0xffff ;limit 0xFFFF
    dw 0x8000 ;base 0~15 bit : 0x8000
    db 0x0b ;base 16~23 bit : 0x0B
    db 0x92 ;P:1, DPL:0, data, expand-up, readable, writable
    db 0x40 ;G:0, D:1, limit 16~19 bit:0
    db 0xcf ;
    db 0x00 ;base 24~32 : 0
```

IDTDESCRIPTOR equ 0x20

```
dw 0xffff
dw 0x0000
db 0x02
db 0x92
db 0xcf
db 0x00
```

GDT_END:

IDTR:

```
dw 256*8-1 ;IDT의 limit
dd 0x00020000 ;IDT의 Base Address
```

IDT_IGNORE:

```
dw ISR_IGNORE ;핸들러는 ISR_IGNORE의 번지수를 기입한다.
dw CODEDESCRIPTOR ;코드세그먼트 셀렉터는 CODEDESCRIPTOR를 기입한다.
db 0
db 0x8E
dw 0x0000
```

;실제로 인터럽트가 발생하였을 때 실행되어야 할 핸들러 루틴이다.

ISR_IGNORE:

```
;CPU의 모든 레지스터 값과 FLAG를 스택에 보존한다.
push    gs
push    fs
push    es
push    ds
pushad
pushfd
cli ;Interrupt Flag를 해제하여 인터럽트를 사용하지 못하게 한다.
nop
sti ;Interrupt Flag를 설정하여 인터럽트를 사용가능하게 한다.
;루틴이 끝날 때 스택에 저장했던 모든 값들을 다시 각각의 레지스터로 옮긴다.
popfd
popad
pop      ds
pop      es
pop      fs
pop      gs
iret ;인터럽트가 발생한 당시의 프로그램의 다음 명령으로 돌아가서 프로그램을 재개한다.
```

msgRMode db "Real Mode", 0

msgPMode db "Protected Mode", 0

times 2048-(\$-\$\$) db 0x00

5. 소스코드 (bootloader.asm)

org 0x7c00 ;메모리의 몇 번지에서 실행해야 하는지를 알려주는 선언문. 부트로더는 0x7c00으로 올라간다.

[BITS 16] ;이 프로그램이 16비트 단위로 데이터를 처리하는 프로그램임을 알린다.

START: ;BOOT1_LOAD로 점프하는 레이블이다.

jmp BOOT1_LOAD ;BOOT1_LOAD로 점프

BOOT1_LOAD:

mov ax, 0x0900

mov es, ax

mov bx, 0x0

;0x13 인터럽트는 DISK I/O, 64비트책 p136 참조

mov ah, 2 ;0x13 인터럽트 호출시 ah에 저장된 값에 따라 수행되는 결과가 다름. 2는 섹터 읽기

mov al, 0x4 ;al 읽을 섹터 수를 지정 1~128 사이의 값을 지정 가능

mov ch, 0 ;실린더 번호 cl의 상위 2비트까지 사용가능 하여 표현

mov cl, 2 ;읽기 시작할 섹터의 번호 1~18 사이의 값, 1에는 부트로더가 있으니 2이상부터

mov dh, 0 ;읽기 시작할 헤드 번호 1~15 값

mov dl, 0x80 ;드라이브 번호. 0x00 - 플로피; 0x80 - 첫 번째 하드, 0x81 - 두 번째 하드

int 0x13 ;0x13 인터럽트 호출

jc BOOT1_LOAD ;Carry 플래그 발생시(=Error) 다시 시도

VIDEO_M:

mov ax, 0xb800 ;비디오 메모리의 시작주소(0xb800)를 ax에 저장하여 ax레지스터를 비디오메모리로 사용한다.

mov es, ax ;ES레지스터에 ax값을 넣는다.

mov ax, 0x00 ;ax레지스터를 0으로 초기화한다.

mov dx, 0 ;dx레지스터를 0으로 초기화한다.

mov bx, 0 ;bx레지스터를 0으로 초기화한다.

mov cx, 80*25*2 ;cx레지스터에 80*25*2(가로*세로*2byte[글자])값을 저장한다.

CLS:

mov [es:bx], ax ;비디오메모리의 주소에 0으로 초기화된 ax의 값을 집어넣어 문자를 삭제한다.

add bx,1 ;bx를 1씩 증가시킨다.

loop CLS ;CLS레이블로 돌아가 루프를 도는데 루프의 횟수는 cx에 저장된 값이다.

INITPRINT: ;ssuos_1,ssuos_2,ssuos_3을 출력해주는 레이블이다.

xor bx,bx ;bx를 0으로 초기화한다.

mov si, ssuos_1 ;si에 ssuos_1의 시작주소를 대입한다.

call PRINT_MSG ;PRINT_MSG레이블을 호출한다.

mov bx,' ' ;bx에 ' '에 해당하는 값을 대입한다. 이를 이용하여 탭을 표현하게 한다.

mov si,ssuos_2 ;si에 ssuos_2의 시작주소를 대입한다.

call PRINT_MSG ;PRINT_MSG레이블을 호출한다.


```

mov bx,160 ;bx에 160을 대입한다. 이를 이용하여 개행을 표현하게 한다.
mov si, ssuos_3 ;si에 ssuos_3의 시작주소를 대입한다.
call PRINT_MSG ;PRINT_MSG레이블을 호출한다.
mov bx, dx ;bx에 dx값을 대입한다.
mov si, select ;si에 select의 시작주소를 대입한다.
call PRINT_MSG ;PRINT_MSG레이블을 호출한다.

```

KEYBOARD: ;키보드 인터럽트를 받는 레이블이다.

```

mov ah, 10h ;ah에 10h를 대입한다.
INT 0x16 ;0x16 인터럽트를 발생시켜 키보드 인터럽트를 발생시킨다.
cmp al,13 ;al에 13이 들어오면
je ENTER ;ENTER레이블로 점프한다.
cmp ah,72 ;al에 72(↑)이 들어오면
je UP ;UP레이블로 점프한다.
cmp ah,80 ;al에 80(↓)이 들어오면
je DOWN ;DOWN레이블로 점프한다.
cmp ah,75 ;al에 75(←)이 들어오면
je LEFT ;LEFT레이블로 점프한다.
cmp ah,77 ;al에 77(→)이 들어오면
je RIGHT ;RIGHT레이블로 점프한다.

```

UP: ;↑방향키가 입력됐을 때 실행되는 레이블이다.

```

cmp dx, 0 ;dx에 있는 값이 0이라면(현재 내 커서의 위치가 ssuos_1이라면)
je CH_OS1 ;CH_OS1레이블로 점프한다.
cmp dx, ' ' ;dx에 있는 값이 ' '이라면(현재 내 커서의 위치가 ssuos_2이라면)
je CH_OS2 ;CH_OS2레이블로 점프한다.
cmp dx, 160 ;dx에 있는 값이 160이라면(현재 내 커서의 위치가 ssuos_3이라면)
je CH_OS1 ;CH_OS1레이블로 점프한다.

```

DOWN: ;↓방향키가 입력됐을 때 실행되는 레이블이다.

```

cmp dx, 0 ;dx에 있는 값이 0이라면(현재 내 커서의 위치가 ssuos_1이라면)
je CH_OS3 ;CH_OS3레이블로 점프한다.
cmp dx, ' ' ;dx에 있는 값이 ' '이라면(현재 내 커서의 위치가 ssuos_2이라면)
je CH_OS2 ;CH_OS2레이블로 점프한다.
cmp dx, 160 ;dx에 있는 값이 160이라면(현재 내 커서의 위치가 ssuos_3이라면)
je CH_OS3 ;CH_OS3레이블로 점프한다.

```

LEFT: ;←방향키가 입력됐을 때 실행되는 레이블이다.

```

cmp dx,0 ;dx에 있는 값이 0이라면(현재 내 커서의 위치가 ssuos_1이라면)
je CH_OS1 ;CH_OS1레이블로 점프한다.
cmp dx,' ' ;dx에 있는 값이 ' '이라면(현재 내 커서의 위치가 ssuos_2이라면)
je CH_OS1 ;CH_OS1레이블로 점프한다.
cmp dx,160 ;dx에 있는 값이 160이라면(현재 내 커서의 위치가 ssuos_3이라면)
je CH_OS3 ;CH_OS3레이블로 점프한다.

```

RIGHT: ;→방향키가 입력됐을 때 실행되는 레이블이다.

```
cmp dx,0 ;dx에 있는 값이 0이라면(현재 내 커서의 위치가 ssuos_1이라면)
je CH_OS2 ;CH_OS2레이블로 점프한다.
cmp dx,' ' ;dx에 있는 값이 ' '이라면(현재 내 커서의 위치가 ssuos_2이라면)
je CH_OS2 ;CH_OS2레이블로 점프한다.
cmp dx, 160 ;dx에 있는 값이 160이라면(현재 내 커서의 위치가 ssuos_3이라면)
je CH_OS3 ;CH_OS3레이블로 점프한다.
```

CH_OS1: ;커서가 ssuos_1에 있는 것을 프린트해주기 위한 레이블이다.

```
mov dx,0 ;dx에 0을 대입한다.
jmp INITPRINT ;INITPRINT레이블로 점프한다.
```

CH_OS2: ;커서가 ssuos_2에 있는 것을 프린트해주기 위한 레이블이다.

```
mov dx,' ' ;dx에 ' '을 대입한다.
jmp INITPRINT ;INITPRINT레이블로 점프한다.
```

CH_OS3: ;커서가 ssuos_3에 있는 것을 프린트해주기 위한 레이블이다.

```
mov dx,160 ;dx에 160을 대입한다.
jmp INITPRINT ;INITPRINT레이블로 점프한다.
```

ENTER: ;enter키가 입력됐을 때 실행되는 레이블이다.

```
cmp dx,0 ;dx에 있는 값이 0이라면(현재 내 커서의 위치가 ssuos_1이라면)
je KERNEL_LOAD ;KERNEL_LOAD레이블로 점프한다.
cmp dx,' ' ;dx에 있는 값이 ' '이라면(현재 내 커서의 위치가 ssuos_2이라면)
je KERNEL2_LOAD ;KERNEL2_LOAD레이블로 점프한다.
cmp dx,160 ;dx에 있는 값이 160이라면(현재 내 커서의 위치가 ssuos_3이라면)
je KERNEL3_LOAD ;KERNEL3_LOAD레이블로 점프한다.
```

KERNEL_LOAD: ;ssuos_1커널을 실행시키기 위한 레이블이다.

```
mov ax, 0x1000 ;커널주소를 ax에 대입한다.
mov es, ax
mov bx, 0x0
mov ah, 2 ;0x13 인터럽트 호출시 섹터를 읽으라는 명령
mov al, 0x3f ;읽을 섹터의수
mov ch, 0x0 ;ssuos_1의 실린더 번호
mov cl, 0x6 ;ssuos_1의 섹터번호
mov dh, 0x0 ;ssuos_1의 헤드 번호
mov dl, 0x80 ;드라이브 번호 0x00 = A: 0x80= C:
```

```
int 0x13 ;Read할 인터럽트를 실행한다.
```

```
jc KERNEL_LOAD ;캐리플래그가 1일 때(실행에 실패했을 때 KERNEL_LOAD 레이블로 점프한다.
jnc GO_BOOT1 ;캐리플래그가 0일 때(실행에 성공했을 때 GO_BOOT1 레이블로 점프한다.
```

KERNEL2_LOAD: ;ssuos_2커널을 실행시키기 위한 레이블이다.

```
mov     ax, 0x1000 ;커널주소를 ax에 대입한다.
mov     es, ax
mov     bx, 0x0
```

```
mov     ah, 2 ;0x13 인터럽트 호출시 섹터를 읽으라는 명령
mov     al, 0x3f ;읽을 섹터의수
mov     ch, 0x9 ;ssuos_2의 실린더 번호
mov     cl, 0x2f ;ssuos_2의 섹터번호
mov     dh, 0xe ;ssuos_2의 헤드번호
mov     dl, 0x80 ;드라이브 번호 0x00 = A: 0x80= C:
```

```
int     0x13 ;Read할 인터럽트를 실행한다.
```

```
jc      KERNEL2_LOAD ;캐리플래그가 1일 때(실행에 실패했을 때 KERNEL2_LOAD 레이블로 점프한다.
```

```
jnc     GO_BOOT1 ;캐리플래그가 0일 때(실행에 성공했을 때 GO_BOOT1 레이블로 점프한다.
```

KERNEL3_LOAD: ;ssuos_3커널을 실행시키기 위한 레이블이다.

```
mov     ax, 0x1000 ;커널주소를 ax에 대입한다.
mov     es, ax
mov     bx, 0x0
```

```
mov     ah, 2 ;0x13 인터럽트 호출시 섹터를 읽으라는 명령
mov     al, 0x3f 읽을 섹터의수
mov     ch, 0xe ;ssuos_3의 실린더번호
mov     cl, 0x7 ;ssuos_3의 섹터번호
mov     dh, 0xe ;ssuos_3의 헤드번호
mov     dl, 0x80 ;드라이브 번호 0x00 = A: 0x80= C:
```

```
int     0x13 ;Read할 인터럽트를 실행한다.
```

```
jc      KERNEL3_LOAD ;캐리플래그가 1일 때(실행에 실패했을 때 KERNEL3_LOAD 레이블로 점프한다.
```

```
jnc     GO_BOOT1 ;캐리플래그가 0일 때(실행에 성공했을 때 GO_BOOT1 레이블로 점프한다.
```

PRINT_MSG: ;문자열을 출력하는 레이블이다.

```
mov     al, byte[si] ;MSG의 주소에서 SI레지스터에 저장된 값만큼을 더한 위치의 문자를 ax레지스터의 하위1바이트에 복사
mov     byte[es:bx],al
```

```
add     si,1 ;SI레지스터에 1을 더하여 다음 문자열로 이동한다.
```

```
add     bx,1 ;bx레지스터에 1을 더한다.
```

```
mov     byte[es:bx],0x07 ;ax레지스터의 상위1바이트에 0x07을 대입하여 글자색을 흰색으로 지정해준다.
```

```
add     bx,1 ;bx의 값에 1을 더해준다.
```

```
cmp     al,0 ;al의 값이 0인지 비교
```

```
jne     PRINT_MSG ; al의 값이 0이 아니라면 PRINT_MSG 레이블로 이동한다.
```

```
ret ;이 레이블을 호출한 곳으로 다시 돌아간다.
```

GO_BOOT1: ;boot1으로 점프하기 위한 레이블이다.

jmp 0x0900:0x0000 ; bootloader에서 0x9000 메모리 번지에 boot1 내용을 로드했으므로 boot1로 점프한다.

;화면에 출력할 문자열들이다.

select db "[O]",0

ssuos_1 db "[] SSUOS_1",0

ssuos_2 db "[] SSUOS_2",0

ssuos_3 db "[] SSUOS_3",0

ssuos_4 db "[] SSUOS_4",0

partition_num : resw 1 ; 커널 선택 값을 저장할 2바이트 영역을 할당한다.

times 446-(\$-\$\$) db 0x00 현재위치(\$ (현재주소)에서 \$\$ (처음시작주소)를 뺀 주소)에서 446까지 0으로 채운다.

PTE: ;ssuos_1, ssuos_2, ssuos_3에 대한 정보가 포함되어 있는 레이블이다. 각 커널의 LBA정보를 알 수 있다.

;ssuos_1의 파티션이다.

partition1 db 0x80, 0x00, 0x00, 0x00, 0x83, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3f, 0x0, 0x00, 0x00

;ssuos_2의 파티션이다.

partition2 db 0x80, 0x00, 0x00, 0x00, 0x83, 0x00, 0x00, 0x00, 0x10, 0x27, 0x00, 0x00, 0x3f, 0x0, 0x00, 0x00

;ssuos_3의 파티션이다.

partition3 db 0x80, 0x00, 0x00, 0x00, 0x83, 0x00, 0x00, 0x00, 0x98, 0x3a, 0x00, 0x00, 0x3f, 0x0, 0x00, 0x00

;ssuos_4의 파티션이다.

partition4 db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

times 510-(\$-\$\$) db 0x00 ;현재위치(\$ (현재주소)에서 \$\$ (처음시작주소)를 뺀 주소)에서 510까지 0으로 채운다.

dw 0xaa55 ;dw는 word단위(2byte)이고 little endian방식이므로 511번지에는 0x55가, 512번지에는 0xaa를 채워넣는다.

;이 라인을 통해 부트로더가 끝임을 알려준다.