주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.01.02 – 2024.01.05

세부 사항

1. 업무 내역 요약 정리

Done & Plan	To-do
1. 부트로더 개발 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 2. 하드디스트 읽기 모듈 개발 - 하드디스트 내 특정 섹터 읽기 3. 모드 전환 모듈 개발 - 리얼모드 - 리얼모드에서 보호모드로의 전환 4. 함수 만들기 - 어셈블리어로 함수 만들기 - C언어로 함수 만들기 - 개발의 편의를 위해 makefile 만들기 - C언어로 함수 만들기 2 5. 인터럽트 핸들러 개발 - PIC 셋팅 - IDT 선언 - IDT 구현 - ISR 구현 6. 키보드 드라이버 개발 - 키보드 드라이버 1 7. 입출력 관리자 개발 - 키보드 드라이버 2 8. 쉘 개발 - 기초적인 Shell	1. 부트로더 개발 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 2. 하드디스트 읽기 모듈 개발 - 하드디스트 내 특정 섹터 읽기 3. 모드 전환 모듈 개발 - 리얼모드 - 리얼모드에서 보호모드로의 전환 4. 함수 만들기 - 어셈블리어로 함수 만들기 - C언어로 함수 만들기

9. 하드디스크 드라이버 개발
- 하드디스크 드라이버
- Qemu
- 읽기
- 쓰기

10. 파일 시스템(ext2) 개발
- printf() 가변인자 구현
- Superblock
- Groupblock
- Bitmap
- Inode & ls
- cd
- 현재 Directory Path
- cat
- Block alloc & free
- Inode alloc & free
- mkdir
- rm

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 부트로더 개발

- 간단한 부트로더 개발

부트로더의 핵심은 512Byte이며 끝 바이트가 0x55AA로 끝나야만 된다는 것에 있습니다.
왜냐하면, 부트로더는 하드 디스크의 첫 섹터인 MBR(Master Boot Record)에 위치하는데, 이 섹터의 크기가 512 바이트입니다.
마지막 두 바이트는 부트 시그너처로 0x55AA라는 값을 가지며, 이는 해당 섹터가 유효한 부트로더 코드를 포함하고 있음을 BIOS나 UEFI에 알려주는 역할을 합니다.

Boot_start.asm

```
[org 0]
[bits 16]

jmp 0x07C0:start

start:


jmp $

times 510-($-$$) db 0
dw 0xAA55
```

```
[org 0]
```

[org 0]는 'origin'의 약자로, 어셈블리어에서 이후의 코드가 메모리의 어느 주소에서 시작될 것인지를 지정합니다. 여기서는 0 을 지정했는데, 실제로 이 코드가 메모리의 0 번지에서 시작한다는 것을 의미하지는 않습니다. 이는 컴파일러에게 상대적인 주소 값을 계산할 때 사용하는 기준점을 제공하는 것입니다.

```
[bits 16]
```

[bits 16]는 이 코드가 16 비트 모드로 실행될 것임을 나타냅니다. 초기 x86 기반 PC에서는 CPU가 16 비트 리얼 모드에서 시작하기 때문에, 부트로더도 16 비트 코드로 작성됩니다.

```
jmp 0x07C0:start
```

jmp 0x07C0:start는 점프 명령어로, 현재 코드 세그먼트를 0x07C0 으로 설정하고 라벨 start로 점프하라는 의미입니다. 여기서 0x07C0 은 BIOS가 부트로더를 로드하는 전형적인 메모리 주소입니다. 이 점프는 코드가 올바른 세그먼트 상대 주소에서 실행되도록 보장합니다.

```
start:
```

start:는 라벨을 선언합니다. 이 라벨은 위의 점프 명령어에서 참조하는 대상이며, 실제 실행 흐름이 시작되는 지점입니다.

```
jmp $
```

jmp $는 무한 루프를 만듭니다. $는 현재 주소를 나타내므로, 자기 자신으로 점프하게 됩니다. 이 부분은 실제

부트로더의 기능이 시작되기 전에 임시로 시스템을 멈추게 하는 역할을 합니다.

```
times 510-($-$$) db 0
```

times 510-($-$$) db 0 이 부분은 코드를 510 바이트로 채우기 위해 사용됩니다. $$는 이 섹션의 시작 주소를 나타내고, $는 현재 주소를 나타냅니다. 따라서 ($-$$)는 현재까지 코드의 크기를 나타냅니다. 510 에서 이 값을 빼서 나머지 부분을 0 으로 채우라는 의미입니다. 이렇게 하면 부트 시그너처를 제외한 나머지 부분을 적절하게 채울 수 있습니다.

```
dw 0xAA55
```

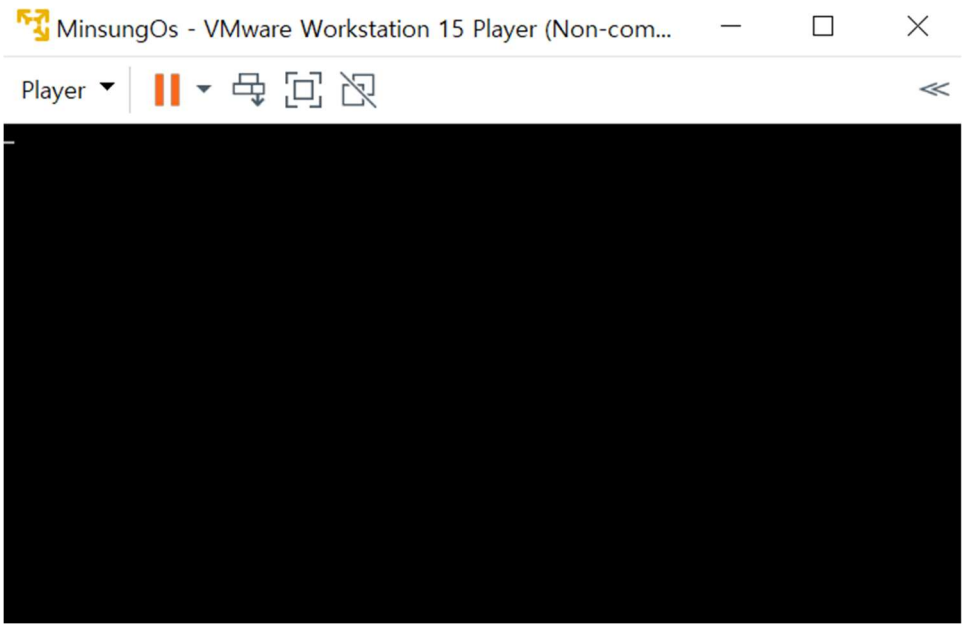
마지막으로 dw 0xAA55 는 실제 부트 시그너처를 설정합니다. dw는 'define word'의 약자로, 16 비트 값을 정의합니다. 이 값은 엔디언에 따라 메모리에 55 AA로 저장되며, 이는 부트로더가 유효함을 BIOS에 알려주는 역할을 합니다.

요약하자면, 이 코드 조각은 CPU가 16 비트 모드에서 실행하도록 설정하고, BIOS가 부트로더를 메모리의 기대 위치에 로드했음을 보정하며, 무한 루프에 들어가서 아무것도 하지 않는 간단한 부트로더의 예시입니다.

Boot_start.asm을 nasm을 이용하여 Boot_start.img로 컴파일합니다.

```
nasm -f bin -o Boot_start.img Boot_start.asm
```

이제 이 Boot_start.img를 VMware Workstation 15 를 이용하여 가상머신에서 돌려봅니다.



실행하게 된다면 작성한 코드대로 무한 루프에 들어가서 아무것도 하지않는 검정색 화면이 뜰 것입니다.
이로서 하드디스크에서 512Byte를 읽어와 아무것도 출력하지 않는 Os를 만들어냈습니다.

- 문자를 출력하는 부트로더 개발

메인보드가 기본적으로 제공하는 BIOS (Basic Input Output Syetem)을 이용하여, 문자를 출력하는 부트로더를 만들어 보겠습니다.

Boot_print.asm

```
[org 0]
[bits 16]

jmp 0x07C0:start

start:

mov ax, 0xB800
mov es, ax

mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09

jmp $

times 510-($-$$) db 0
dw 0xAA55
```

```
mov ax, 0xB800
mov es, ax
```

이 두 줄은 비디오 메모리를 가리키는 세그먼트를 설정합니다. 0xB800 은 텍스트 모드 비디오 버퍼의 시작 주소입니다. ax 레지스터에 이 주소를 로드하고, 그 다음에 es 세그먼트 레지스터에 ax의 값을 복사합니다.

(mov es, 0xB800 이라고 안하고) ax 레지스터에 주소를 로드하고 es 세그먼트 레지스터에 ax의 값을 복사하는 이유: mov 명령어는 세그먼트 레지스터에 직접 상수 값을 옮길 수 없음. 세그먼트 레지스터에 값을 로드하기 전에, 그 값을 일반 레지스터(예: ax)에 먼저 로드해야 합니다. 그런 다음 mov 명령어를 사용하여 그 레지스터의 값을 세그먼트 레지스터로 옮길 수 있음 → x86 어셈블리 언어의 설계 규칙

ES는 x86 아키텍처의 세그먼트 레지스터 중 하나입니다. 세그먼트 레지스터는 메모리 세그먼트를 참조하는데 사용되며, 특히 리얼 모드에서 메모리 관리에 중요한 역할을 합니다.
x86 CPU에서는 메모리 주소를 세그먼트와 오프셋의 조합으로 지정합니다. 세그먼트는 메모리의 큰 구획을 가리키고, 오프셋은 그 세그먼트 내의 특정 주소를 나타냅니다. ES는 'Extra Segment'의 약자로, 기본적으로는 데이터를 저장하기 위한 추가적인 세그먼트를 가리키는 데 사용됩니다.
예를 들어, ES 레지스터가 0xB800 을 가리킬 때 ES:0 주소는 0xB8000 을 가리키게 됩니다. 텍스트 모드 비디오 메모리에 접근할 때 ES 레지스터를 이용하여, 화면에 표시할 문자 데이터를 해당 메모리 주소에 쓸 수 있습니다.
코드에서 mov es, ax는 AX 레지스터에 저장된 값을 ES 세그먼트 레지스터로 옮기는 명령입니다. 이 경우 AX에는 0xB800 이 저장되어 있으므로, 이 명령을 실행한 후 ES는 비디오 메모리 세그먼트를 가리키게 되어, 비디오 메모리에 직접 접근이 가능해집니다.

```
mov byte[es:0], 'h'
mov byte[es:1], 0x09
```

이 코드는 비디오 메모리의 첫 번째 문자 셀에 'h' 문자를 쓰고, 그 문자의 속성(색상 및 깜빡임)을 설정합니다. 여기서 0x09 는 하늘색 전경색과 검정색 배경색을 나타냅니다.

```
mov byte[es:2], 'i'
mov byte[es:3], 0x09
```

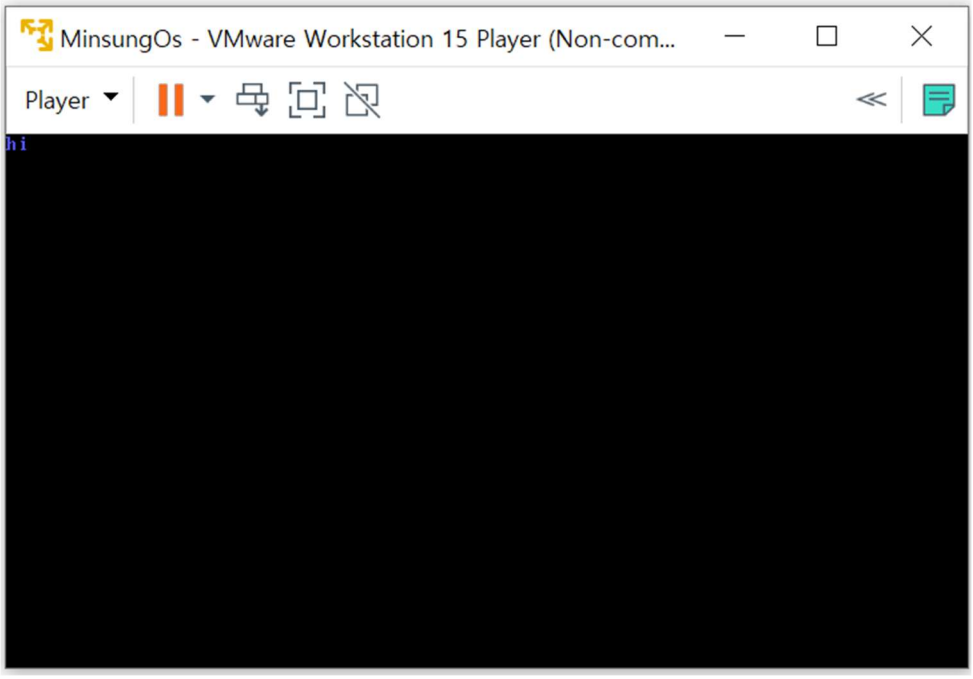
이 다음 두 줄은 두 번째 문자 셀에 'i' 문자를 쓰고, 동일한 속성을 적용합니다

전체적으로 이 코드는 컴퓨터를 부팅할 때 화면에 'hi'라는 글자를 하늘색 전경과 검정색 배경으로 표시하고, 그 상태에서 멈춰서 아무것도 하지 않는 매우 간단한 부트로더의 기능을 수행합니다.

Boot_print.asm을 nasm을 이용하여 Boot_ print.img로 컴파일합니다.

```
nasm -f bin -o Boot_print.img Boot_print.asm
```

이제 이 Boot_print.img를 VMware Workstation 15 를 이용하여 가상머신에서 돌려봅니다.



실행하게 된다면 작성한 코드대로 무한 루프에 들어가서 'hi'라는 글자를 하늘색 전경과 검정색 배경으로 표시하는 것이 뜰 것입니다.
이로서 하드디스크에서 512Byte를 읽어와 모니터에 hi를 출력하는 Os를 만들어냈습니다.

2. 하드디스크 읽기 모듈 개발

- 하드디스크 내 특정 섹터 읽기

처음 BIOS가 자동으로 읽었던 섹터를 "섹터 1"이라 합시다. 이 512Byte 코드는 물리주소 0x7C00 에 적재되어 있습니다.

x86 의 예약된 메모리 구조를 고려해서 물리 주소 0x10000 에 "섹터 2"를 올려봅시다.

Sector2_read.asm

```
[org 0x10000]
[bits 16]

mov ax, 0xB800
mov es, ax

mov byte[es:4], 'i'
mov byte[es:5], 0x09
mov byte[es:6], 'h'
mov byte[es:7], 0x09

jmp $

times 512-($-$$) db 0
```

이 코드는 16 비트 리얼 모드에서 실행되는 작은 어셈블리 프로그램으로, 화면의 텍스트 모드 비디오 메모리에 문자를 출력하고 무한 루프에 들어가는 기능을 합니다.

```
[org 0x10000]
[bits 16]
```

[org 0x10000] 디렉티브는 어셈블러에게 이 프로그램이 메모리 주소 0x10000 에서 시작한다고 알려줍니다. 그러나 이 디렉티브는 실제 메모리 주소를 바꾸지 않고, 오히려 프로그램 내에서의 주소 계산을 위한 기준점(base)으로 사용됩니다.

[bits 16] 디렉티브는 이 코드가 16 비트 모드에서 실행될 것임을 나타냅니다. 즉, 이후의 코드는 16 비트 리얼 모드에 맞게 작성되었음을 의미합니다.

```
mov ax, 0xB800
mov es, ax
```

이 두 줄은 비디오 메모리의 세그먼트 시작 주소인 0xB800 을 es 레지스터에 로드합니다. es는 추가 세그먼트 레지스터로, 여기서는 비디오 메모리에 접근하기 위해 사용됩니다.

```
mov byte[es:4], 'i'
mov byte[es:5], 0x09
mov byte[es:6], 'h'
mov byte[es:7], 0x09
```

이 네 줄의 코드는 텍스트 모드 비디오 메모리에 문자를 쓰는 명령입니다. 비디오 메모리는 각 문자와 그 문자의 속성(색상과 밝기)을 위한 두 바이트를 사용합니다.

'i'와 'h' 문자는 각각 4 번지와 6 번지에 쓰여지며, 그 뒤를 따르는 0x09 는 문자의 속성(여기서는 밝은 청색)을 설정합니다. 문자들은 화면상의 두 번째 문자 위치에 나타나게 됩니다.

```
jmp $
```

jmp $는 현재 주소로 점프하는 명령어로, 이 경우 프로그램을 무한 루프에 빠뜨립니다. $는 어셈블러에서 현재 주소를 나타내는 기호입니다. 따라서, 이 명령은 계속해서 자기 자신으로 점프하여 다른 어떠한 코드도 실행되지 않게 합니다.

```
times 512-($-$$) db 0
```

이 라인은 부트 섹터나 다른 종류의 섹터를 512 바이트로 채우기 위한 것입니다. times 디렉티브는 주어진 명령을 특정 횟수만큼 반복하라는 지시어입니다.
512-($-$$) 계산은 코드의 시작($$)부터 현재 위치($)까지의 바이트 수를 512 에서 빼서 얼마나 많은 0 을 추가해야 하는지 계산합니다. 그 결과는 남은 공간을 0 으로 채워 프로그램의 크기를 512 바이트로 만듭니다.

이 코드 자체는 실제 하드웨어에서 실행할 수 있는 완전한 부트 섹터나 프로그램은 아니지만, 비디오 메모리에 문자를 출력하고 무한 루프에 들어가는 기본적인 메커니즘을 보여줍니다. 이 코드를 실행하려면 부트 시그니처나 다른 초기화 코드가 필요하며, 메모리 주소 0x10000 으로 로드되기 위한 추가적인 부트스트랩 로직이 필요합니다.

Boot_read.asm

```
[org 0]
[bits 16]

jmp 0x07C0:start

start:

mov ax, 0xB800
mov es, ax

mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09

read:
    mov ax, 0x1000
    mov es, ax
    mov bx, 0 ; 0x1000:0000 주소로 읽어 => 물리주소 0x10000

    mov ah, 2 ; 디스크에 있는 데이터를 es:bx의 주소로
    mov al, 1 ; 1섹터를 읽을 것이다
    mov ch, 0 ; 0번째 실린더
    mov cl, 2 ; 2번째 섹터부터 읽기 시작한다
    mov dh, 0 ; 헤드는 0
    mov dl, 0 ; 플로피 디스크 읽기
    int 13h
```

```
    jc read ; 에러나면 다시

jmp 0x1000:0

times 510-($-$$) db 0
dw 0xAA55
```

이 코드는 16 비트 리얼 모드 아래에서 실행되는 간단한 부트 섹터 프로그램으로, 화면에 "hi"를 출력하고, 하드 디스크의 두 번째 섹터를 메모리에 로드하는 기능을 가지고 있습니다.

```
[org 0]
[bits 16]
```

[org 0]는 이 프로그램이 메모리의 0 번지에서 시작될 것으로 어셈블러에게 알립니다. 실제로는 BIOS가 부트 섹터를 0x7C00 에 로드하지만, 코드 내부적으로는 0 번지로 간주합니다.

[bits 16]는 이 코드가 16 비트 모드에서 실행됨을 나타냅니다.

```
jmp 0x07C0:start
```

이는 프로그램의 나머지 부분으로 점프하는 명령입니다. 0x07C0 은 세그먼트 주소이고, start는 이 코드 내의 레이블입니다. 이 명령은 CPU가 실제로 이 코드를 0x7C00 세그먼트 주소에서 실행하도록 합니다.

```
start:
```

start:는 이후 코드의 시작점을 나타내는 레이블입니다.

```
mov ax, 0xB800
mov es, ax
```

0xB800 은 텍스트 모드 비디오 메모리를 위한 세그먼트 주소입니다. 이 주소를 es 레지스터에 로드하여, 이후의 메모리 연산에서 비디오 메모리를 쉽게 참조할 수 있게 합니다.

```
mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09
```

이 네 줄은 세그먼트 es에 설정된 비디오 메모리의 시작 부분에 'h'와 'i' 문자를 밝은 청색으로 출력합니다.

```
read:
      mov ax, 0x1000
      mov es, ax
      mov bx, 0 ; 0x1000:0000 주소로 읽어 => 물리주소 0x10000
```

read: 레이블은 디스크 읽기 연산의 시작점을 나타냅니다.

ax 레지스터에 0x1000 을 로드하고, 이를 es 세그먼트 레지스터에 복사합니다. 이는 메모리 세그먼트를 설정하는 작업입니다.

bx 레지스터를 0 으로 설정합니다. es:bx 조합은 이후에 디스크에서 읽은 데이터를 저장할 메모리 주소를 의미합니다. 여기서는 0x1000:0x0000 이므로, 물리주소로 환산하면 0x10000 이 됩니다.

```
mov ah, 2 ; 디스크에 있는 데이터를 es:bx 의 주소로
mov al, 1 ; 1 섹터를 읽을 것이다
mov ch, 0 ; 0 번째 실린더
mov cl, 2 ; 2 번째 섹터부터 읽기 시작한다
mov dh, 0 ; 헤드는 0
mov dl, 0 ; 플로피 디스크 읽기
int 13h
```

ah 레지스터에 0x02 를 설정합니다. 이는 BIOS 인터럽트 int 13h의 디스크 읽기 기능을 나타냅니다.
al 레지스터에 1 을 설정합니다. 이는 한 개의 섹터를 읽겠다는 의미입니다.
ch 레지스터에 0 을 설정하여 0 번째 실린더를 선택합니다.
cl 레지스터에 2 를 설정하여 2 번째 섹터부터 읽기를 시작하겠다는 것을 의미합니다.
dh 레지스터에 0 을 설정하여 첫 번째 헤드를 선택합니다.
dl 레지스터에 0 을 설정하여 첫 번째 드라이브(플로피 디스크)를 선택합니다.
int 13h 명령은 위에서 설정된 값에 따라 디스크 읽기 작업을 수행합니다.

```
jc read ; 에러나면 다시
```

jc read는 'Carry' 플래그가 설정되어 있으면, 즉 에러가 발생하면 read 레이블로 다시 점프하여 읽기를 재시도하게 합니다.

```
jmp 0x1000:0
```

이 명령어는 메모리 주소 0x10000 에 로드된 코드를 실행하기 위해 해당 위치로 점프합니다.

```
times 510-($-$$) db 0
dw 0xAA55
```

times 510-($-$$) db 0 는 부트 섹터가 총 512 바이트 되도록 나머지 부분을 0 으로 채웁니다. 마지막 dw 0xAA55 는 유효한 부트 섹터의 서명(signature)입니다.

이 코드는 실제 컴퓨터에서 부팅할 수 있는 부트 섹터 코드의 기본 형태를 가지고 있습니다. 컴퓨터가 부팅될 때, BIOS는 이 코드를 메모리의 0x7C00 위치에 로드하고 실행합니다. 코드는 화면에 "hi"를 출력한 다음, 두 번째 섹터를 메모리의 0x10000 위치에 로드하려고 시도합니다. int 13h 서비스는 디스크의 섹터를 읽어 메모리에 로드하는 기본적인 방법입니다.

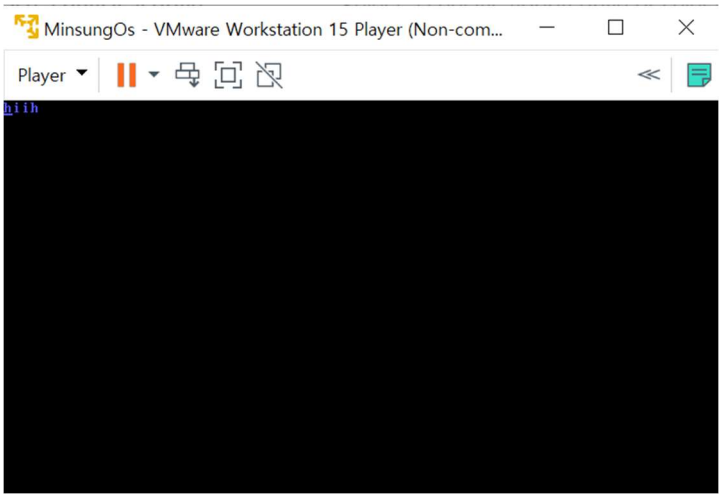
Boot_read.asm은 times 510-($-$$) db 0 이고 Sector2_read.asm 은 times 512-($-$$) db 0 인 이유:
Boot_read.asm 파일은 부트 시그니처 포함을 위해 510 바이트를 0 으로 채우고, 마지막 2 바이트에는 부트 시그니처(dw 0xAA55 명령으로 부트 시그니처를 추가)를 둠.
반면, Sector2_read.asm 파일은 추가적인 부트 시그니처가 필요 없으므로 전체 512 바이트를 데이터 또는 0 으로 채움.

이제 두 개의 파일, Boot_read.asm 과 Sector2_read.asm 이 준비되었습니다. 이 둘을 각각 NASM으로 컴파일하면 각각의 img 파일을 얻을 수 있습니다.
이 둘을 이제 하나의 img 파일로 합쳐봅시다. 명령 프롬프트를 열어 다음과 같이 입력합니다.

```
nasm -f bin -o Boot_read.img Boot_read.asm
nasm -f bin -o Sector2.img Sector2.asm
```

```
PS C:\실크로드소프트\qa팀 인턴\OS (2024년 1분기)\dev> cmd /c copy /b Boot_read.img+Sector2.img final.img
Boot_read.img
Sector2.img
        1개 파일이 복사되었습니다.
```

이제 이 final.img를 가상머신에 돌려서 제대로 동작하는지 확인합니다. 오류가 없다면 hiih 가 출력될 것입니다.



이제 BIOS의 초기 로딩 메커니즘에 의존하지 않고, 스스로 원하는 섹터의 데이터를 원하는 메모리 위치에 적재할 수 있는 능력을 갖추게 되었습니다. 이는 컴퓨터가 부팅하는 과정을 우리의 의도에 따라 더 세밀하게 제어할 수 있음을 의미하며, 이를 통해 운영 체제를 더 유연하고 효율적으로 로드할 수 있는 기반을 마련하게 되었습니다.

3. 모드 전환 모듈 개발

- 리얼모드(Real Mode)

" 0x10000 에다 섹터 2 를 로드한 후, jmp를 통해 0x10000 로 이동하는 건 이해가 되는데 왜 jmp 0x1000:0 이 되는 건가? 이렇게 되면 물리주소 0x10000 가 아닌 0x1000 에 이동하는 것이 아닌가? "

"물리주소 0x7C00 에 첫 번째 섹터(부트섹터)가 BIOS에 의해 자동으로 적재된다고 했는데 왜 Boot_read.asm의 첫 줄에 jmp 0x07C0:start 라고 하는건가? jmp 0x7C00:start 뭐 이래야 하는 거 아닌가? "

이에 대해 이해하기 위해서는 16 비트 환경에서의 세그먼트:오프셋 구조를 이해해야 합니다.

리얼모드에서는 20 비트 주소 공간을 표현하는 방법으로 '세그먼트:오프셋' 방식을 사용합니다. 이는 16 비트 주소 공간을 넘어서 더 큰 주소 공간을 표현하기 위한 방법입니다.

예를 들어, '0x1000:0' 이라는 주소가 있다고 가정해봅시다. 여기서 '0x1000'이 세그먼트 부분이고, '0'이 오프셋 부분입니다.
이 둘을 합쳐서 물리 주소를 얻기 위해서는 세그먼트 부분을 16 배하고, 그 결과에 오프셋을 더합니다. 이렇게 하면 '0x1000'이 0x10000 이 되고,

오프셋 '0'을 더해서 최종적으로 '0x10000'이라는 물리 주소를 얻게 됩니다.

비슷하게, '0x07C0:start' 주소의 경우에도 '0x07C0'이 세그먼트 부분이고, 'start'가 오프셋 부분입니다.
세그먼트 부분을 16 배하면 '0x7C00'이 되고, 여기에 'start' 오프셋을 더하면 최종적으로 '0x7C00'이라는 물리 주소를 얻게 됩니다.
이 때 'start'는 주소의 시작점을 의미하므로 '0'으로 간주할 수 있습니다.
이처럼, 세그먼트:오프셋 방식은 세그먼트 부분을 16 배하고, 그 결과에 오프셋을 더해서 물리 주소를 계산하는 방식입니다.
이 방식을 이해하면 'jmp 0x1000:0'이나 'jmp 0x07C0:start' 같은 주소 표현이 어떻게 물리 주소를 가리키는지 이해할 수 있습니다.

- 리얼모드에서 보호모드로의 전환

32 비트 환경에서는 16 비트 세그먼트:오프셋과는 다른 32 비트 세그먼트:오프셋 변환 과정을 거치게 됩니다.
32 비트 환경에서는 세그먼트와 오프셋을 그대로 더해 물리 주소를 계산하는 방식을 사용합니다.
즉 0x1000:0050 이면 그대로 더해 0x1050 이 되는 것이죠. 하지만 여기에 더해 여러 가지 기능적인 부분이 추가되게 됩니다.
32 비트 보호 모드에서는 세그먼트:오프셋 구조가 조금 복잡해집니다. '세그먼트' 부분이 이전처럼 메모리의 시작 위치를 가리키는 것이 아니라, '선택자(selector)'라는 것을 가리키게 됩니다.
선택자는 세그먼트 디스크립터 테이블 내의 인덱스로, 이 테이블은 각 세그먼트의 시작 주소, 길이, 접근 권한 등의 정보를 저장하고 있습니다. CPU는 선택자를 통해 이 테이블을 참조하고, 해당 세그먼트의 실제 시작 주소를 얻어냅니다.
예를 들어, '0x0008:0050' 주소가 있다면 '0x0008'은 선택자를 의미합니다. CPU는 이 선택자를 통해 세그먼트 디스크립터 테이블을 참조하여 실제 세그먼트 시작 주소를 찾아냅니다. 만약 선택자 '0x0008'이 테이블에서 '0x1000'을 가리킨다면, 이는 세그먼트의 시작 주소가 '0x1000'임을 의미합니다.
그런 다음, 이 시작 주소에 오프셋 '0x0050'을 더하면, 최종적으로 '0x1050'이라는 물리 주소를 얻게 됩니다. 즉, '0x0008:0050' 주소는 결국 물리 주소 '0x1050'을 가리키게 됩니다.
이렇게, 32 비트 보호 모드에서는 선택자를 통해 세그먼트 정보를 참조하고, 오프셋을 더하여 물리 주소를 계산하는 방식을 사용합니다.
각각의 세그먼트는 하나의 Table을 가지게 됩니다. 따라서 우리가 할 일은 각각의 세그먼트에 대해 하나의 Table을 만들어 놓은 다음 이 Table들이 어디에 위치해 있는지를 CPU에게 알려주면 됩니다.

Boot_protect.asm

```
[org 0]
[bits 16]

jmp 0x07C0:start

start:
mov ax, cs
mov ds, ax
mov es, ax

mov ax, 0xB800
mov es, ax

mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09

read:
    mov ax, 0x1000
    mov es, ax
    mov bx, 0 ; 0x1000:0000 주소로 읽어 => 물리주소 0x10000

    mov ah, 2 ; 디스크에 있는 데이터를 es:bx의 주소로
    mov al, 1 ; 1섹터를 읽을 것이다
    mov ch, 0 ; 0번째 실린더
    mov cl, 2 ; 2번째 섹터부터 읽기 시작한다
    mov dh, 0 ; 헤드는 0
    mov dl, 0 ; 플로피 디스크 읽기
    int 13h

    jc read ; 에러나면 다시
```

```
    mov dx, 0x3F2 ;플로피디스크 드라이브의
    xor al, al  ; 모터를 끈다
    out dx, al

    cli

lgdt[gdtr]

mov eax, cr0
or eax, 1
mov cr0, eax

jmp $+2
nop
nop

mov bx, DataSegment
mov ds, bx
mov es, bx
mov fs, bx
mov gs, bx
mov ss, bx

jmp dword CodeSegment:0x10000


gdtr:
dw gdt_end - gdt - 1
dd gdt+0x7C00
```

```
gdt:

    dd 0,0 ; NULL 세그
    CodeSegment equ 0x08
    dd 0x0000FFFF, 0x00CF9A00 ; 코드 세그
    DataSegment equ 0x10
    dd 0x0000FFFF, 0x00CF9200 ; 데이터 세그
    VideoSegment equ 0x18
    dd 0x8000FFFF, 0x0040920B ; 비디오 세그

gdt_end:

times 510-($-$$) db 0
dw 0xAA55
```

이 코드는 리얼 모드에서 보호 모드로 전환하는 과정을 포함한 부트 섹터의 어셈블리 코드입니다. 화면에 “hi”를 출력하고, 디스크 읽기를 수행하며, 글로벌 디스크립터 테이블(GDT)을 설정한 후 CPU를 보호 모드로 전환하는 복잡한 작업을 수행합니다.

```
[org 0]
[bits 16]
```

[org 0]는 이 프로그램이 메모리 시작점인 0 번지에서 시작되어야 함을 어셈블러에 알려주는 지시어입니다. 실제로는 BIOS가 부트 섹터를 0x7C00 주소에 로드하지만, 코드 내부에서는 0 번지로 간주됩니다.
[bits 16]는 이 코드가 16 비트 CPU 모드에서 실행됨을 의미합니다. 이는 오래된 리얼 모드에서 실행되는 코드임을 나타냅니다.

```
jmp 0x07C0:start
```

이 줄은 CPU에게 0x07C0 세그먼트와 start 레이블 위치로 점프하라고 지시합니다. 이렇게 함으로써 코드가 0x7C00 에서 실행될 것으로 기대하는 부분을 맞추게 됩니다.

```
mov ax, cs
mov ds, ax
mov es, ax
```

mov ax, cs는 현재 코드 세그먼트 레지스터(cs)의 값을 ax 레지스터로 복사합니다. cs는 현재 코드가 저장된 세그먼트 주소를 가지고 있습니다.
mov ds, ax는 ax에 저장된 값을 데이터 세그먼트 레지스터(ds)로 복사합니다. 이는 데이터 세그먼트를 코드 세그먼트와 동일하게 설정합니다.
mov es, ax는 같은 값을 추가 세그먼트 레지스터(es)로도 복사합니다. 이는 es를 cs와 동일하게 설정합니다.

```
mov ax, 0xB800
mov es, ax
```

여기서 mov ax, 0xB800 명령은 ax 레지스터에 0xB800 값을 넣습니다. 0xB800 은 텍스트 모드 비디오 메모리의 세그먼트 주소입니다.
그 다음 mov es, ax 명령은 ax의 값을 es 세그먼트 레지스터로 다시 복사합니다. 이러한 작업을 통해 es는 이제 비디오 메모리를 가리키게 됩니다.

mov es, ax를 2 번 하는 이유:
코드의 첫 부분에서 ds와 es를 cs와 같게 설정하는 것은 초기화 과정의 일부입니다. 이렇게 함으로써, 코드와 데이터 세그먼트가 같은 메모리 영역을 가리키도록 하여 데이터에 접근할 때 예상치 못한 문제를 방지합니다. 그러나 실제로 비디오 메모리에 접근하기 위해서는 es를 비디오 메모리의 세그먼트 주소로 바꿔야 합니다.
만약 es 레지스터의 값을 변경하지 않고 비디오 메모리에 접근하려 한다면, es가 가리키는 기본 세그먼트 주소는 여전히 코드 세그먼트 주소가 되고, 이는 0x7C00 으로 설정되어 있을 것입니다. 그 결과, 코드는 의도하지 않은 메모리 위치에 데이터를 쓰게 되어 예상치 못한 동작이나 시스템의 충돌을 일으킬 수 있습니다.
따라서 es를 명시적으로 0xB800 으로 설정하는 것은 비디오 메모리에 올바르게 접근하기 위한 필수적인 단계입니다.

```
mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09
```

화면의 첫 번째 문자 위치에 'h' 문자를 출력하고, 배경과 전경 색상을 설정하는 속성 값 0x09 를 적용합니다.
두 번째 문자 위치에 'i' 문자를 출력하고, 같은 속성 값을 적용합니다.

```
read:
        mov ax, 0x1000
        mov es, ax
        mov bx, 0 ; 0x1000:0000 주소로 읽어 => 물리주소 0x10000
```

read: 레이블은 디스크 읽기 루틴의 시작점입니다.
0x1000 값을 ax에 로드하고 es에 복사함으로써, 메모리의 0x10000 위치에 데이터를 저장할 준비를 합니다.

```
        mov ah, 2
```

ah 레지스터에 2 를 설정함으로써, '디스크 읽기' 함수를 선택합니다. int 13h 인터럽트는 ah에 설정된 값에 따라 다른 기능을 수행합니다.

```
        mov al, 1
```

al 레지스터에 1 을 설정하여 한 섹터를 읽겠다는 의도를 나타냅니다.

```
        mov ch, 0
```

ch는 실린더 번호를 설정하는데 사용됩니다. 여기서 0 은 첫 번째 실린더를 나타냅니다. (실린더 번호는 0 부터 시작합니다.)

```
        mov cl, 2
```

cl은 섹터 번호를 설정하는데 사용됩니다. 여기서 2 는 두 번째 섹터를 의미합니다. (섹터 번호 역시 1 부터 시작합니다.)

```
        mov dh, 0
```

dh는 헤드 번호를 설정합니다. 대부분의 경우, 플로피 드라이브에서는 단일 헤드만 사용하므로 0 을 설정합니다.

```
        mov dl, 0
```

dl은 드라이브 번호를 설정합니다. 0 은 일반적으로 첫 번째 플로피 드라이브를 나타냅니다. 하드 드라이브의 경우, 80h부터 시작합니다.

```
        int 13h
```

마지막으로, int 13h 명령을 실행하여 위에서 설정한 모든 값들과 함께 디스크 읽기 작업을 수행합니다.

이 조합된 명령어는 디스크의 두 번째 섹터에서 데이터를 읽어와서, es:bx에서 설정한 메모리 주소로 데이터를 로드하는 작업을 수행합니다. 이 경우, es가 0x1000 으로 설정되어 있고, bx가 0 으로 설정되어 있으므로, 로드된 데이터는 물리적 주소 0x10000 에 저장됩니다.

```
        jc read ; 에러나면 다시
```

jc 명령어는 만약 'Carry Flag'가 설정되어 있다면 (에러가 발생했다면) read 레이블로 점프하여 읽기 작업을 다시 수행합니다.

```
        mov dx, 0x3F2 ;플로피디스크 드라이브의
```

dx 레지스터에 0x3F2 값을 로드합니다. 0x3F2 는 플로피 디스크 드라이브의 디지털 출력 레지스터(Digital Output Register, DOR)의 포트 주소입니다. 이 레지스터는 플로피 디스크 드라이브의 모터 제어와 리셋 기능을 다루는 데 사용됩니다.

```
        xor al, al    ; 모터를 끈다
```

xor al, al 명령은 al 레지스터의 값을 자기 자신과 XOR 연산하게 합니다. 어떤 값이든 자기 자신과 XOR하면 0 이 되므로, 이는 al 레지스터를 0 으로 클리어합니다. DOR에 0 을 쓰면 플로피 디스크 드라이브의 모든 모터가 꺼집니다.

```
        out dx, al
```

out dx, al 명령은 al 레지스터의 값을 dx에 저장된 포트 주소로 보냅니다. 여기서는 al이 0 이므로, 0x3F2 포트에 0 을 보내어 플로피 드라이브의 모터를 꺼버립니다.

```
        cli
```

cli (Clear Interrupt Flag) 명령은 인터럽트 플래그를 클리어하여 인터럽트를 비활성화합니다. 이는 보호 모드로 전환하는 동안 인터럽트에 의해 발생할 수 있는 문제를 방지하기 위해 수행됩니다. 인터럽트가 비활성화되면, 현재 실행 중인 코드가 완료될 때까지 시스템은 다른 인터럽트 요청을 무시하게 됩니다.

이 명령어들은 플로피 디스크 작업이 끝난 후 불필요한 전력 소비를 막기 위해 드라이브 모터를 끄고, 시스템이 중요한 작업을 수행하는 동안 다른 인터럽트에 의해 방해받지 않도록 하기 위해 사용됩니다.

```
lgdt [gdtr]
```

GDT(Global Descriptor Table)와 GDTR(Global Descriptor Table Register)은 컴퓨터의 메모리 관리 시스템과 관련된 중요한 요소들입니다. 이들은 x86 아키텍처에서 보호 모드를 구현할 때 사용됩니다. 각각에 대해 설명해보겠습니다.

GDT (Global Descriptor Table)

GDT는 세그먼트 기반의 메모리 관리 시스템에서 각 메모리 세그먼트의 특성을 정의하는 테이블입니다. 여기에는 각 세그먼트의 베이스 주소(base address), 한계(limit), 그리고 액세스 권한(access rights)과 같은 속성이 담겨 있습니다. 세그먼트의 속성은 메모리 보호, 코드와 데이터의 분리, 효율적인 멀티태스킹 등을 가능하게 합니다.

GDTR (Global Descriptor Table Register)

GDTR은 CPU 내에 있는 레지스터 중 하나로, 현재 시스템에서 사용 중인 GDT의 위치와 크기를 나타냅니다. lgdt 명령어를 통해 GDTR은 GDT의 시작 주소(base address)와 한계(limit)를 로드합니다. 즉, GDTR은 GDT의 실제 데이터가 아니라, GDT가 어디에 위치하는지와 그 크기가 얼마나 되는지를 CPU에 알려주는 역할을 합니다.

간단히 말해, GDT는 데이터를 저장하는 메모리 영역이고, GDTR은 그 메모리 영역을 가리키는 CPU 내의

포인터입니다.

lgdt [gdtr] 명령은 "Load Global Descriptor Table Register"의 약자로, 글로벌 디스크립터 테이블(GDT)의 시작 주소와 한계(limit, 즉 크기)를 GDTR(Global Descriptor Table Register)에 로드하는 x86 명령어입니다.

lgdt 명령어는 특히 CPU가 리얼 모드에서 보호 모드로 전환할 때 필요한데, 이 모드 전환 과정 중에 새로운 메모리 세그먼트 관리 방식을 설정하는 데 사용됩니다.

[gdtr]는 GDTR을 가리키는 메모리 주소입니다. 이 주소에는 GDT의 시작 주소와 GDT의 크기 정보가 있습니다. lgdt 명령은 다음과 같은 형식의 데이터를 기대합니다:

```
gdtr:
    dw  GDT의 크기 - 1 ; GDT의 한계값(limit)
    dd  GDT의 시작 주소 ; GDT의 베이스(base) 주소
```

dw (Define Word)는 16 비트 값을 정의합니다. 여기서는 GDT의 크기에서 1을 뺀 값을 나타냅니다.
dd (Define Doubleword)는 32 비트 값을 정의합니다. 여기서는 GDT의 시작 주소를 나타냅니다.

이 명령을 실행함으로써, CPU는 GDT의 위치를 알게 되고, 보호 모드에서 메모리 세그먼트를 올바르게 관리할 수 있게 됩니다.

```
mov eax, cr0
```

CR0 레지스터의 현재 값을 EAX 레지스터로 이동시킵니다. CR0 에는 시스템의 동작 방식을 제어하는 여러 플래그들이 있는데, 그 중에서도 가장 낮은 비트(0 번 비트)는 PE(Protection Enable) 플래그입니다.

```
or eax, 1
```

EAX 레지스터의 값을 1과 OR 연산합니다. 이 연산의 결과로 EAX의 0번 비트가 1로 설정됩니다. 이 비트는 PE 플래그로, 보호 모드를 활성화하는 데 사용됩니다. 즉, 이 명령은 보호 모드를 활성화하기 위한 준비를 합니다.

```
mov cr0, eax
```

변경된 EAX 레지스터의 값을 다시 CR0 레지스터로 이동시켜 실제로 보호 모드를 활성화합니다. 이 때부터 시스템은 리얼 모드가 아닌 보호 모드로 동작하게 됩니다.

보호 모드에서는 더 큰 메모리 주소 공간에 접근할 수 있고, 세분화된 메모리 보호 기능을 사용할 수 있으며, 멀티태스킹과 같은 고급 기능을 구현할 수 있습니다. 요약하자면, 이 코드는 컴퓨터를 단순하고 제한된 리얼 모드에서 훨씬 더 강력한 보호 모드로 전환하는 과정을 수행합니다.

```
jmp $+2
```

jmp $+2 는 현재 명령어의 주소($)에서 2바이트 뒤로 점프하라는 명령입니다. $는 현재 명령어의 주소를 나타내며, +2 는 바로 다음 명령어로 점프하라는 것을 의미합니다. 이 경우, 바로 다음 명령어는 nop입니다. 이는 프로세서가 명령어 파이프라인을 비우고, 특히 CR0 레지스터를 수정한 후 보호 모드로 전환하는 데 필요한 내부 동기화를 수행할 시간을 제공합니다.

```
nop
nop
```

nop는 "No Operation"의 약자로, CPU가 아무런 작업도 하지 않고 다음 명령어로 넘어가도록 하는 명령어입니다. 이 두 nop 명령은 CPU에게 보호 모드로 전환하는 동안 아무런 다른 작업도 하지 않도록 하여, 전환 과정이 안정적으로 완료되도록 합니다.

```
mov bx, DataSegment
mov ds, bx
mov es, bx
mov fs, bx
mov gs, bx
mov ss, bx
```

DataSegment는 이전에 정의된 GDT(글로벌 디스크립터 테이블) 내의 데이터 세그먼트 선택자입니다. mov bx, DataSegment는 bx 레지스터에 데이터 세그먼트 선택자를 로드합니다.

mov ds, bx는 데이터 세그먼트 레지스터 ds를 bx에 저장된 값으로 설정합니다. 데이터 세그먼트는 일반적으로 데이터를 저장하는 데 사용됩니다.

mov es, bx, mov fs, bx, mov gs, bx는 추가 세그먼트 레지스터들을 bx에 저장된 값, 즉 데이터 세그먼트 선택자로 설정합니다. 이 세그먼트 레지스터들은 특정 작업에 따라 다양한 용도로 사용될 수 있습니다.

mov ss, bx는 스택 세그먼트 레지스터 ss를 설정합니다. 스택 세그먼트는 함수 호출, 지역 변수 저장 등 스택에 관련된 데이터를 저장하는 데 사용됩니다.

```
jmp dword CodeSegment:0x10000
```

jmp dword CodeSegment:0x10000 는 코드 세그먼트에 대한 far jump(원거리 점프)를 수행합니다. CodeSegment는 GDT 내의 코드 세그먼트 선택자를 나타내고, 0x10000 은 새로운 코드 세그먼트 내에서의 오프셋을 나타냅니다.

dword는 이 점프가 32 비트 오퍼랜드를 사용한다는 것을 나타냅니다. 즉, 이 명령은 CPU가 GDT에 정의된 코드 세그먼트의 0x10000 오프셋으로 실행을 이동시키라고 지시합니다.

이 명령어들의 실행은 보호 모드에서 CPU의 세그먼트 레지스터들을 새로운 세그먼트 선택자로 업데이트하고, 새로운 코드 세그먼트에서 프로그램의 실행을 계속하도록 합니다. 이는 컴퓨터가 보호 모드로 완전히 전환된 후에 수행되는 작업으로, 이 모드에서는 세그먼트의 크기와 권한을 정의하는 GDT에 따라 메모리에 접근하게 됩니다.

```
gdtr:
dw gdt_end - gdt - 1
dd gdt+0x7C00
```

gdtr:은 GDTR에 로드할 데이터를 정의하는 레이블입니다.

dw gdt_end - gdt - 1 은 GDT의 총 크기에서 1 을 뺀 값을 정의합니다. dw는 Define Word로, 16 비트의 크기를 가진 데이터를 정의합니다. GDT의 크기를 나타낼 때는 실제 바이트 수에서 1 을 뺀 값을 사용합니다.

dd gdt+0x7C00 은 GDT의 베이스 주소를 정의합니다. dd는 Define Doubleword로, 32 비트의 크기를 가진 데이터를 정의합니다. 여기서 gdt는 GDT 테이블의 시작 주소를 나타내고, 0x7C00 은 부트 섹터가 로드된 메모리 위치를 나타냅니다. 이 주소는 GDT가 실제 메모리에 로드되는 위치를 반영합니다.

```
gdt:
        dd 0,0 ; NULL 세그
        CodeSegment equ 0x08
        dd 0x0000FFFF, 0x00CF9A00 ; 코드 세그
        DataSegment equ 0x10
        dd 0x0000FFFF, 0x00CF9200 ; 데이터 세그
        VideoSegment equ 0x18
        dd 0x8000FFFF, 0x0040920B ; 비디오 세그
```

gdt:은 GDT를 정의하는 레이블입니다.

dd 0,0 은 GDT의 첫 번째 항목으로, NULL 세그먼트 디스크립터를 정의합니다. 이는 GDT의 첫 번째 항목이 항상 비어있어야 한다는 규칙을 따릅니다.

CodeSegment equ 0x08 는 코드 세그먼트의 선택자를 0x08 로 설정합니다. equ는 이퀄라이즈(equalize)의 약자로, 지정된 값을 상수로 정의합니다.

dd 0x0000FFFF, 0x00CF9A00 는 코드 세그먼트 디스크립터를 정의합니다. 여기서 설정된 값들은 세그먼트의 한계, 접근 권한, 그리고 기타 속성을 설정합니다.

DataSegment equ 0x10 는 데이터 세그먼트의 선택자를 0x10 으로 설정합니다.

dd 0x0000FFFF, 0x00CF9200 는 데이터 세그먼트 디스크립터를 정의합니다. 이 또한 세그먼트의 한계와 속성을 설정합니다.

VideoSegment equ 0x18 는 비디오 세그먼트의 선택자를 0x18 로 설정합니다.

dd 0x8000FFFF, 0x0040920B는 비디오 메모리 세그먼트 디스크립터를 정의합니다. 이는 비디오 메모리에 접근할 때 사용됩니다.

```
gdt_end:
```

gdt_end:는 GDT의 끝을 나타내는 레이블입니다. 이는 gdtr에서 GDT의 크기를 계산할 때 사용됩니다.

```
times 510-($-$$) db 0
dw 0xAA55
```

times 510-($-$$) db 0 은 부트 섹터를 512 바이트로 채우기 위해 필요한 만큼 0 으로 채웁니다. 여기서 $는 현재 주소를 나타내고, $$는 섹션의 시작 주소를 나타냅니다. 따라서 ($-$$)는 현재까지 코드의 길이를 나타내며, 510-($-$$)는 510 에서 현재 코드 길이를 뺀 나머지 바이트 수를 계산합니다.

dw 0xAA55 는 부트 섹터의 유효성을 체크하는 부트 시그니처입니다. BIOS는 이 시그니처를 확인하여 유효한 부트 섹터를 찾습니다.

이 코드들은 리얼 모드에서 보호 모드로의 전환을 준비하고, 부트 섹터가 올바르게 로드되었음을 보장하는 데 필수적인 부분입니다.

Sector2_protect.asm

```
CodeSegment equ 0x08
DataSegment equ 0x10
VideoSegment equ 0x18

[org 0x10000]
[bits 32]

mov ax, VideoSegment
mov es, ax

mov byte[es:0x08], 'P'
mov byte[es:0x09], 0x09

jmp $

times 512-($-$$) db 0
```

이 코드는 32 비트 프로텍티드 모드(보호 모드)에서 실행되며, 비디오 메모리에 문자를 출력하고 실행을 멈추게 하는 부트 섹터 코드의 일부입니다. CodeSegment, DataSegment, VideoSegment는 세그먼트 선택자를 설정하고, [org 0x10000]과 [bits 32]는 코드의 위치와 모드를 지정합니다.

```
CodeSegment equ 0x08
DataSegment equ 0x10
VideoSegment equ 0x18
```

이 세 줄은 세그먼트 선택자를 정의합니다. equ는 'equal'의 약어로, 주어진 값에 이름을 할당합니다.
CodeSegment는 코드 세그먼트의 선택자로 0x08 을 할당합니다.
DataSegment는 데이터 세그먼트의 선택자로 0x10 을 할당합니다.
VideoSegment는 비디오 메모리 세그먼트의 선택자로 0x18 을 할당합니다.
이 선택자들은 글로벌 디스크립터 테이블(GDT)에 정의된 세그먼트에 대응합니다.

```
[org 0x10000]
[bits 32]
```

[org 0x10000]는 코드가 메모리의 0x10000 위치에서 실행될 것임을 어셈블러에 지시합니다.
[bits 32]는 코드가 32 비트 프로텍티드 모드에서 실행될 것임을 나타냅니다.

```
mov ax, VideoSegment
mov es, ax
```

mov ax, VideoSegment는 VideoSegment의 값을 ax 레지스터로 이동합니다.
mov es, ax는 ax 레지스터의 값을 es 세그먼트 레지스터로 이동합니다. 이는 비디오 메모리에 접근하기 위해 es를 비디오 메모리 세그먼트로 설정합니다.

```
mov byte[es:0x08], 'P'
mov byte[es:0x09], 0x09
```

첫 번째 줄은 비디오 메모리의 0x08 오프셋에 문자 'P'를 쓰는 명령입니다.
두 번째 줄은 비디오 메모리의 0x09 오프셋에 속성 바이트 0x09 를 쓰는 명령입니다. 이 속성 바이트는 문자의 전경색과 배경색을 설정합니다.

```
jmp $
```

jmp $는 현재 위치로 무한 점프하는 명령으로, CPU를 무한 루프로 들어가게 하여 더 이상의 코드 실행을 막습니다.

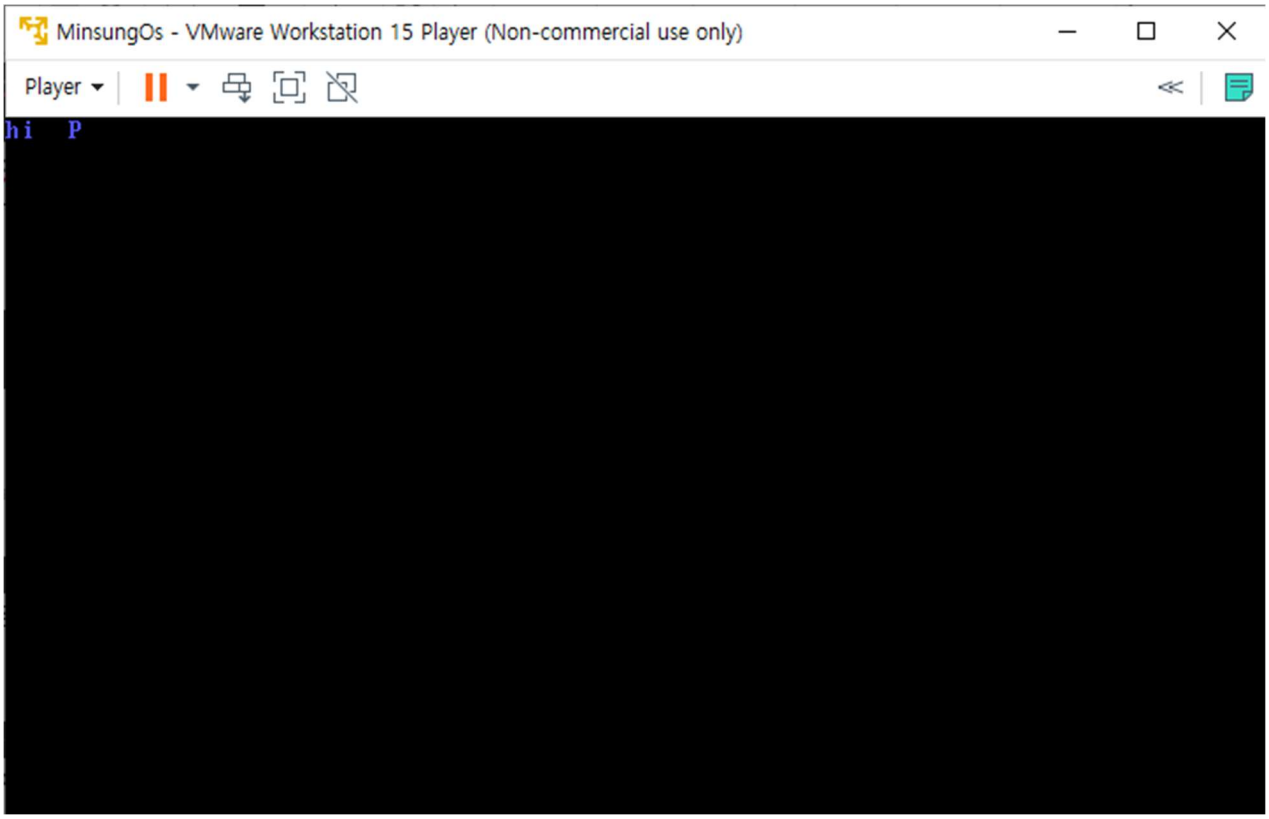
```
times 512-($-$$) db 0
```

이 줄은 부트 섹터를 512 바이트로 채우기 위해 남은 공간을 0 으로 채우는 명령입니다. $는 현재 주소를 의미하고, $$는 섹션의 시작 주소를 의미합니다. 따라서 ($-$$)는 현재까지의 코드 길이를 나타내며, 512-($-$$)는 필요한 0 의 수를 계산합니다.

이 코드는 프로텍티드 모드에서 실행되어 화면에 'P' 문자를 출력하고, 더 이상의 작업 없이 시스템을 멈추게 됩니다.

2 개의 파일을 만들었으니 컴파일 한 후 가상머신에 돌려봅니다.

```
PS C:\실크로드소프트\qa팀 인턴\OS (2024년 1분기)\dev> nasm -f bin -o Sector2_protect.img Sector2_protect.asm
PS C:\실크로드소프트\qa팀 인턴\OS (2024년 1분기)\dev> cmd /c copy /b Boot_protect.img+Sector2_protect.img final.img
Boot_protect.img
Sector2_protect.img
        1개 파일이 복사되었습니다.
```



Boot_protect.asm 파일에서 시작해서 화면에 'hi'라는 인사말을 띄운 뒤, 하드디스크를 읽어서 두 번째 섹터의 내용을 메모리 주소 0x10000 의 위치에 저장했습니다. 그리고 나서 16 비트의 제한된 환경에서 벗어나 넓은 공간을 활용할 수 있는 32 비트 환경으로 환경 설정을 바꾸었습니다. 마지막으로, Sector2_protect.asm에 정의된 코드로 점프하여 'P'라는 글자를 화면에 성공적으로 출력함으로써, 모든 과정이 매끄럽게 진행됨을 확인했습니다.

4. 함수 만들기

- 어셈블리어로 함수 만들기

어셈블리어로 C언어의 함수에 해당하는 것을 만들어봅니다.

지금까지 문자를 출력하려면 일일이 비디오 세그먼트에다 한 문자 한 문자 정성스럽게 써야만 했습니다.

이를 함수로 바꿔서 문자열의 첫 번째 주소만 전달해도 모든 문자열을 출력할 수 있도록 해봅니다.

Sector2_function.asm

```
CodeSegment equ 0x08
DataSegment equ 0x10
VideoSegment equ 0x18

[org 0x10000]
[bits 32]

START:
    mov bx, DataSegment
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    mov ss, bx
    lea esp, [START]

    mov edi,0
    mov esi, msg
    call printf

    jmp $

printf:
    push eax
    push es
    mov ax, VideoSegment ; 비디오
    mov es, ax

printf_loop:

    mov al, byte [esi]
    mov byte [es:edi], al
    inc edi
    mov byte [es:edi], 0x09
    inc esi
    inc edi
    or al, al
    jz printf_end
    jmp printf_loop

printf_end:
    pop es
    pop eax
    ret

msg db 'Call printf',0
```

```
times 512-($-$$) db 0
```

```
CodeSegment equ 0x08
DataSegment equ 0x10
VideoSegment equ 0x18
```

여기서 equ는 'equal'의 약자로, 오른쪽에 있는 값을 왼쪽에 있는 이름에 할당합니다. CodeSegment, DataSegment, VideoSegment는 세그먼트 선택자로, 이후 코드에서 사용할 각 세그먼트의 오프셋을 나타냅니다.

```
[org 0x10000]
```

[org 0x10000]는 이 프로그램이 메모리 주소 0x10000 에서 시작됨을 나타냅니다. org는 'origin'의 약자로, 메모리 상에서 코드가 시작될 위치를 지정합니다.

```
[bits 32]
```

[bits 32]는 이 코드가 32 비트 모드로 실행될 것임을 나타냅니다. 이는 프로세서에게 명령어들이 32 비트로 처리되어야 한다는 것을 알려줍니다.

```
START:
```

START:는 이 코드 블록의 시작 지점에 레이블을 붙입니다. 이는 나중에 코드 내에서 START로 참조할 수 있게 해줍니다.

```
mov bx, DataSegment
mov ds, bx
mov es, bx
mov fs, bx
mov gs, bx
mov ss, bx
```

이 명령어들은 ds, es, fs, gs, ss 세그먼트 레지스터에 DataSegment 값을 로드합니다. 이는 세그먼트 레지스터들을 초기화하여, 데이터 세그먼트를 가리키게 합니다.

```
lea esp, [START]
```

lea 명령어는 'load effective address'의 약자로, START 레이블의 주소를 esp (스택 포인터) 레지스터에 로드합니다. 이는 스택 포인터를 현재 코드의 시작 부분으로 설정합니다.

```
mov edi, 0
mov esi, msg
call printf
```

edi 레지스터를 0 으로 초기화한 후, esi 레지스터에 msg 레이블의 시작 주소를 로드합니다. printf 함수를 호출하여 msg 문자열을 화면에 출력합니다.

```
jmp $
```

jmp $는 현재 위치로 점프하는 명령어로, 이는 무한 루프를 생성하여 프로그램이 종료되지 않고 계속 실행되게 합니다.

```
printf:
	push eax
	push es
```

printf 레이블은 printf 함수의 시작점입니다. eax와 es 레지스터의 값을 스택에 저장합니다. 이는 함수가 끝난 후 원래의 값으로 복원할 수 있게 해줍니다.

```
mov ax, VideoSegment
mov es, ax
```

ax 레지스터에 VideoSegment 값을 로드하고, 이 값을 es 세그먼트 레지스터로 옮깁니다. es는 비디오 메모리 영역을 가리키게 됩니다.

```
printf_loop:
	mov al, byte [esi]
	mov byte [es:edi], al
	inc edi
	mov byte [es:edi], 0x09
	inc esi
	inc edi
	or al, al
	jz printf_end
	jmp printf_loop
```

printf_loop는 문자열을 화면에 출력하는 루프입니다. 문자열의 각 문자를 비디오 메모리로 복사하고, 문자의 속성 바이트를 설정합니다. 속성 바이트 0x09 는 문자의 색상을 지정합니다. 문자열 끝을 확인하기 위해 al 레지스터를 0 과 비교하고, 0 이면 루프를 종료합니다.

```
printf_end:
	pop es
	pop eax
	ret
```

printf_end는 printf 함수의 끝입니다. 스택에서 es와 eax 값을 복원하고, ret 명령으로 호출한 위치로 돌아갑니다.

```
msg db 'Call printf',0
```

msg에 'Call printf'라는 문자열을 저장하고, 문자열의 끝을 나타내기 위해 0 을 추가합니다.

```
times 512-($-$$) db 0
```

부트 섹터를 512 바이트로 채우기 위해 필요한 만큼의 0 으로 나머지 공간을 채웁니다.

```
dw 0xAA55
```

부트 섹터의 끝에는 항상 0xAA55 라는 시그니처가 있어야 하며, 이는 부트 섹터가 유효하다는 것을 BIOS에 알려줍니다.

이 코드는 부트 섹터에서 실행되며, 프로텍티드 모드에서 화면에 문자열을 출력하는 작업을 수행한 후, 무한 루프에 들어가서 프로그램이 더 이상 진행되지 않도록 합니다.

컴파일을 해보고 가상머신에 돌려보겠습니다.

```
nasm -f bin -o Sector2_function.img Sector2_function.asm
```

```
PS C:\실크로드소프트\qa팀 인턴\OS (2024년 1분기)\dev> cmd /c copy /b Boot_function.img+Sector2_function.img
Boot_function.img
Sector2_function.img
        1개 파일이 복사되었습니다.
```

- C언어로 함수 만들기

```c
// HelloWorld.c

void main()
{
    int line = 5;
    char str[11] = "HelloWorld";

    char *video = (char*)(0xB8000 + 160 * line);

    for (int i = 0; str[i] != 0; i++) {
        *video++ = str[i];
        *video++ = 0x03;
    }
    return;
}
```

이 코드는 C 언어로 작성된 것으로, 메모리의 특정 위치에 문자열을 쓰는 작업을 수행합니다.

```c
void main()
{
        int line = 5;
```

```
	char str[11] = "HelloWorld";
```

'line'이라는 변수에 5 를 할당하고, 'str'이라는 문자열 배열에 "HelloWorld"라는 문자열을 저장합니다.

```
	char *video = (char*)(0xB8000 + 160 * line);
```

'video'라는 포인터를 선언하고, 메모리 주소 '0xB8000 + 160 * line'에 대한 포인터로 초기화합니다. 여기서 '0xB8000'은 텍스트 모드 비디오 메모리의 시작 주소를 나타내며, '160 * line'은 5 번째 라인으로 이동하는 것을 의미합니다. 이 코드는 특정 환경(예: 리얼 모드의 x86 아키텍처)에서 실행되는 것을 가정하고 있습니다.

```
	for (int i = 0; str[i] != 0; i++) {
		*video++ = str[i];
		*video++ = 0x03;
	}
```

'for' 루프를 사용하여 'str'에 있는 각 문자를 'video'가 가리키는 메모리 위치에 복사합니다. 이후 'video' 포인터를 1 증가시켜 다음 바이트를 가리키게 한 후, 그 위치에 '0x03'을 쓰고 'video' 포인터를 다시 1 증가시킵니다. '0x03'은 문자의 색상과 배경을 결정하는 값입니다.

```
	return;
}
```

'main' 함수를 종료합니다.

이 코드는 특정 환경에서 문자열 "HelloWorld"를 5 번째 라인에 녹색 글자로 출력하는 효과를 가집니다. 이 코드를 실행하기 위해서는 해당 시스템이 직접적인 메모리 접근을 허용하고, 0xB8000 주소에 비디오 메모리가 매핑되어 있어야 합니다.

```
gcc -c -m32 -ffreestanding HelloWorld.c -o HelloWorld.o
```

이 명령어는 GNU Compiler Collection(GCC)의 컴파일러를 사용하여 C 소스 코드를 컴파일하는 데 사용됩니다.

gcc: 이는 GNU Compiler Collection의 일부인 C 컴파일러를 호출하는 명령어입니다.
-c: 이 옵션은 소스 파일을 컴파일하고 어셈블만 수행하며, 링크는 수행하지 않습니다. 결과적으로 오브젝트 파일(.o 파일)이 생성됩니다.
-m32: 이 옵션은 컴파일러에게 32 비트 코드를 생성하도록 지시합니다. 이는 주로 64 비트 시스템에서 32 비트 코드를 컴파일할 때 사용됩니다.
-ffreestanding: 이 옵션은 컴파일러에게 'freestanding' 환경에서 실행될 것임을 알립니다. 즉, 표준 라이브러리 함수를 사용할 수 없으며, 진입점이 반드시 'main'이 아니어도 된다는 것을 의미합니다. 이는 주로 운영 체제 커널 또는 임베디드 시스템 코드를 컴파일하는 데 사용됩니다.
HelloWorld.c: 이는 컴파일할 소스 코드 파일의 이름입니다.
-o HelloWorld.o: -o 옵션 다음에 오는 이름(여기서는 'HelloWorld.o')은 컴파일러가 생성할 출력 파일의 이름을 지정합니다.
따라서, 이 명령어는 'HelloWorld.c'라는 C 소스 코드 파일을 32 비트 'freestanding' 오브젝트 코드로 컴파일하고, 결과를 'HelloWorld.o'라는 파일에 저장하라는 의미입니다.

```
minsung@ubuntu:~/Dev/Os$ ld -melf_i386 -Ttext 0x10200 -nostdlib HelloWorld.o -o HelloWorld.img
ld: warning: cannot find entry symbol  start; defaulting to 0000000000010200
```

이 명령어는 GNU의 링커(ld)를 사용하여 컴파일된 오브젝트 파일을 링크하고 실행 가능한 이미지 파일을 생성합니다.

ld: 이는 GNU의 링커를 호출하는 명령어입니다. 링커는 여러 오브젝트 파일을 하나의 실행 가능한 이미지로 결합합니다.
-melf_i386: 이 옵션은 링커에게 생성할 출력 파일의 형식이 ELF(Executable and Linkable Format)이며, 아키텍처가 i386(32 비트 인텔 아키텍처)임을 알립니다.
-Ttext 0x10200: -Ttext 옵션은 프로그램의 텍스트 섹션(즉, 실행 가능한 코드가 들어가는 섹션)이 메모리에서 시작될 주소를 지정합니다. 여기서는 '0x10200'이라는 주소를 지정하였습니다.
-nostdlib: 이 옵션은 링커에게 표준 라이브러리를 링크하지 말라는 지시를 내립니다. 이는 보통 'freestanding' 환경에서 실행될 코드를 링크할 때 사용됩니다.
HelloWorld.o: 이는 링크할 오브젝트 파일의 이름입니다.
-o HelloWorld.img: -o 옵션 다음에 오는 이름(여기서는 'HelloWorld.img')은 링커가 생성할 출력 파일의 이름을 지정합니다.

따라서, 이 명령어는 'HelloWorld.o'라는 오브젝트 파일을 표준 라이브러리 없이 링크하고, 텍스트 섹션의 시작 주소를 '0x10200'으로 설정하여, ELF 형식의 32 비트 실행 가능 이미지를 생성하며, 그 결과를 'HelloWorld.img'라는 파일에 저장하라는 의미입니다.

```
objcopy -O binary HelloWorld.img disk.img
```

이 명령어는 GNU의 'objcopy' 도구를 사용하여 'HelloWorld.img' 파일의 형식을 변환하고, 그 결과를 'disk.img' 파일에 저장합니다.

objcopy: 이는 GNU의 바이너리 유틸리티 중 하나로, 오브젝트 파일의 형식을 변환하거나, 오브젝트 파일에서 특정 섹션을 추출하거나, 오브젝트 파일에 패치를 적용하는 등의 작업을 수행합니다.
-O binary: -O 옵션은 출력 파일의 형식을 지정합니다. 'binary'를 지정하면, 출력 파일은 헤더나 메타데이터 없이 순수한 바이너리 데이터만을 포함하게 됩니다.
HelloWorld.img: 이는 입력 파일의 이름으로, 형식을 변환할 오브젝트 파일을 지정합니다.
disk.img: 이는 출력 파일의 이름으로, 변환된 바이너리 데이터를 저장할 파일을 지정합니다.

따라서, 이 명령어는 'HelloWorld.img'라는 오브젝트 파일을 순수한 바이너리 형식으로 변환하고, 그 결과를 'disk.img'라는 파일에 저장하라는 의미입니다. 이 과정을 통해 'HelloWorld.img' 파일이 바로 실행 가능한 디스크 이미지 파일로 변환됩니다.

Boot_c.asm

```
[org 0]
[bits 16]

jmp 0x07C0:start

start:
mov ax, cs
mov ds, ax
mov es, ax

mov ax, 0xB800
mov es, ax

mov byte[es:0], 'h'
mov byte[es:1], 0x09
mov byte[es:2], 'i'
mov byte[es:3], 0x09

read:
    mov ax, 0x1000
    mov es, ax
    mov bx, 0

    mov ah, 2
    mov al, 2
    mov ch, 0
    mov cl, 2
    mov dh, 0
    mov dl, 0
    int 13h

    jc read

    mov dx, 0x3F2
    xor al, al
    out dx, al

    cli

lgdt[gdtr]

mov eax, cr0
or eax, 1
mov cr0, eax

jmp $+2
nop
nop
```

```
mov bx, DataSegment
mov ds, bx
mov es, bx
mov fs, bx
mov gs, bx
mov ss, bx

jmp dword CodeSegment:0x10000


gdtr:
dw gdt_end - gdt - 1
dd gdt+0x7C00

gdt:

    dd 0,0
    CodeSegment equ 0x08
    dd 0x0000FFFF, 0x00CF9A00
    DataSegment equ 0x10
    dd 0x0000FFFF, 0x00CF9200
    VideoSegment equ 0x18
    dd 0x8000FFFF, 0x0040920B

gdt_end:

times 510-($-$$) db 0
dw 0xAA55
```

Sector2_c.asm

```
CodeSegment equ 0x08
DataSegment equ 0x10
VideoSegment equ 0x18

[org 0x10000]
[bits 32]

START:
    mov bx, DataSegment
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    mov ss, bx
    lea esp, [START]

    mov edi, 0
    mov esi, msg
    call printf

    jmp dword CodeSegment: 0x10200

printf:
    push eax
    push es
    mov ax, VideoSegment ; 비 디 오
    mov es, ax

printf_loop:

    mov al, byte [esi]
    mov byte [es:edi], al
    inc edi
    mov byte [es:edi], 0x09
    inc esi
    inc edi
    or al, al
    jz printf_end
    jmp printf_loop

printf_end:
    pop es
    pop eax
    ret

msg db 'Call printf',0

times 512-($-$$) db 0
```

```
minsung@ubuntu:~/Dev/Os$ nasm -f bin -o Sector2_c.img Sector2_c.
asm
minsung@ubuntu:~/Dev/Os$ nasm -f bin -o Boot_c.img Boot_c.asm
minsung@ubuntu:~/Dev/Os$ cat Boot_c.img Sector2_c.img disk.img > final.img
```

이 명령어는 Unix/Linux 시스템에서 사용하는 cat 명령어로, 여러 파일의 내용을 합쳐서 하나의 파일에 출력합니다.

cat: 이는 concatenate(연결)의 줄임말로, 하나 이상의 파일의 내용을 출력하거나 파일을 만들거나, 파일을 연결하는 데 사용하는 명령어입니다.

Boot.img Sector2.img disk.img: 이들은 cat 명령어가 처리할 입력 파일들의 이름입니다. 입력 파일들의 내용은

순서대로 출력되거나 합쳐집니다.

>: 이 기호는 리다이렉션(redirect) 연산자로, 명령어의 출력을 표준 출력(보통은 콘솔이나 터미널)에서 다른 곳(여기서는 'final.img' 파일)으로 변경하는 데 사용합니다.

final.img: 이는 cat 명령어의 출력이 저장될 파일의 이름입니다. 이 파일은 명령어가 실행된 후에 'Boot.img', 'Sector2.img', 'disk.img' 세 파일의 내용을 순서대로 포함합니다.

따라서, 이 명령어는 'Boot.img', 'Sector2.img', 'disk.img' 세 파일의 내용을 순서대로 합쳐서 'final.img'라는 파일에 저장하라는 의미입니다. 이렇게 생성된 'final.img' 파일은 세 입력 파일의 내용을 순차적으로 포함하게 됩니다.

final.img를 가상머신에 올려보면,

C언어로 운영체제를 개발할 수 있는 환경을 구축한 것을 볼 수 있습니다.