

주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.01.08 – 2024.01.12

세부 사항

1. 업무 내역 요약 정리

Plan	To-do
1. 부트로더 개발 - 정의 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 2. 하드디스크 읽기 모듈 개발 - 하드디스크의 실린더와 헤드 - 섹터 - 하드디스크 내 특정 섹터 읽기 3. 모드 전환 모듈 개발 - 리얼모드와 보호모드에 대한 기본 개념 - 리얼모드 환경에서의 세그먼트:오프셋 구조 - 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	금주에는 작주에 진행하였던 것들을 복기하며, 더 자세히 공부해보았습니다. 1. 비트와 바이트 - 비트와 바이트 정의 - 비트와 바이트의 차이점 - 비트와 바이트가 클 때의 장단점 2. 부트로더 개발 - 정의 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 3. 하드디스크 읽기 모듈 개발 - 하드디스크의 실린더와 헤드 - 섹터 - 하드디스크 내 특정 섹터 읽기 4. 모드 전환 모듈 개발 - 리얼모드와 보호모드에 대한 기본 개념 - 리얼모드 환경에서의 세그먼트:오프셋 구조 - 리얼모드에서 보호모드로의 전환

8. 셸 개발

- 셸과 cli의 차이점

CLI는 사용자와 컴퓨터 시스템 간의 상호작용 방식을 일컫는 반면, 셸은 그러한 상호작용을 가능하게 하는 구체적인 소프트웨어를 가리킵니다.

모든 셸은 CLI를 제공하지만, 모든 CLI가 셸은 아닙니다. 예를 들어, 애플리케이션 내부에 CLI 기능이 내장되어 있을 수 있지만, 그것이 운영 체제의 셸이라고 할 수는 없습니다.

셸은 사용자가 시스템과 상호작용하는 많은 방법 중 하나이며, CLI는 그러한 상호작용의 형태 중 하나입니다.

- 기초적인 Shell

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. 비트와 바이트




- 비트와 바이트 정의

비트(bit)와 바이트(byte)는 컴퓨터에서 데이터를 표현하고 처리하는 가장 기본적인 단위입니다.

비트: Binary Digit의 줄임말로, 0 또는 1의 두 가지 상태만을 가질 수 있는 정보의 최소 단위
컴퓨터는 이러한 비트를 이용하여 모든 정보를 표현하고 처리

비트 bit	비트 bit	비트 bit
1 bit 	2 bit 	3 bit 
0, 1	00, 01, 10, 11	000, 001 010, 011 100, 101 110, 111
2개 값 표현 가능	4개 값 표현 가능	8개 값 표현 가능

바이트: 8개의 비트가 모인 단위로, 일반적으로 컴퓨터에서 데이터를 표현하고 저장하는 기본 단위로 사용

바이트 Byte	바이트 Byte	표현 가능한 경우의 수																								
 비트 8개 모이면 바이트 정보의 기본 단위	1 Byte  영문자 A~Z 숫자 0~9 기호 256개 표현 가능	1 Byte  2 ⁸ 2 ⁷ 2 ⁶ 2 ⁵ 2 ⁴ 2 ³ 2 ² 2 ¹ <table border="1"> <tr><td>1 비트</td><td>2¹</td><td>2</td></tr> <tr><td>2 비트</td><td>2²</td><td>4</td></tr> <tr><td>3 비트</td><td>2³</td><td>8</td></tr> <tr><td>4 비트</td><td>2⁴</td><td>16</td></tr> <tr><td>5 비트</td><td>2⁵</td><td>32</td></tr> <tr><td>6 비트</td><td>2⁶</td><td>64</td></tr> <tr><td>7 비트</td><td>2⁷</td><td>128</td></tr> <tr><td>8 비트 (1 바이트)</td><td>2⁸</td><td>256</td></tr> </table>	1 비트	2 ¹	2	2 비트	2 ²	4	3 비트	2 ³	8	4 비트	2 ⁴	16	5 비트	2 ⁵	32	6 비트	2 ⁶	64	7 비트	2 ⁷	128	8 비트 (1 바이트)	2 ⁸	256
1 비트	2 ¹	2																								
2 비트	2 ²	4																								
3 비트	2 ³	8																								
4 비트	2 ⁴	16																								
5 비트	2 ⁵	32																								
6 비트	2 ⁶	64																								
7 비트	2 ⁷	128																								
8 비트 (1 바이트)	2 ⁸	256																								

컴퓨터를 처음 개발한 영미권 국가에서는 알파벳 소문자 a부터 z, 대문자 A부터 Z, 숫자 0부터 9, 그리고 다양한 기호들을 표현하기 위해 8개의 비트를 하나로 묶어 '바이트'라는 단위를 만들었습니다. 바이트는 8개의 비트로 구성되어 있으므로, 2의 8승인 총 256개의 다른 값을 표현할 수 있습니다. 이렇게 하면 영문자, 숫자, 기호 등을 충분히 담을 수 있습니다.

그래서 바이트는 정보를 표현하는 기본 단위가 되었고, 우리가 컴퓨터의 메모리나 저장장치의 용량을 계산할 때도 바이트를 기본 단위로 사용하게 되었습니다.

- 비트와 바이트의 차이점

단위의 차이: 비트는 Binary Digit의 줄임말로, 정보의 최소 단위입니다. 0 또는 1의 두 가지 상태만을 가질 수 있습니다. 반면, 바이트는 비트 8개가 모인 단위로, 컴퓨터에서 데이터를 표현하고 저장하는 기본 단위입니다.

사용 목적의 차이: 비트는 주로 데이터의 전송 속도를 나타내는 데 사용됩니다. 인터넷 속도는 'Mbps' (메가비트

퍼 세컨드)로 표현되며, 이는 1 초에 전송되는 비트의 수를 의미합니다.

반면, 바이트는 주로 저장용량이나 파일 크기를 표현하는 데 사용됩니다. 하드 드라이브의 용량이나 메모리의 크기는 'GB' (기가바이트)나 'TB' (테라바이트) 등의 단위로 표현됩니다.

표현력 차이: 1 비트는 2 개(0 과 1)의 정보만 표현할 수 있지만, 1 바이트(8 비트)는 2 의 8 승인 256 개의 다양한 정보를 표현할 수 있습니다.

- 비트와 바이트가 클 때의 장단점

비트가 크면:

장점: 정보를 더 정밀하게 표현할 수 있습니다. 예를 들어, 이미지나 음향 데이터의 해상도를 높이거나, 더 많은 색상을 표현하는 데 필요한 비트 수가 늘어납니다. 또한, 더 큰 숫자를 처리할 수 있으므로, 과학적 연산이나 정밀한 그래픽 처리에 유리합니다.

단점: 더 많은 저장 공간과 처리 시간이 필요합니다. 더 많은 비트를 저장하고 처리하려면 그만큼 더 많은 메모리 공간과 CPU 처리 시간이 필요하므로, 성능에 영향을 줄 수 있습니다.

바이트가 크면:

장점: 더 많은 정보를 저장하고 처리할 수 있습니다. 예를 들어, 텍스트나 이미지, 동영상 등의 복잡한 데이터를 저장하고 처리하는 데 필요한 바이트 수가 늘어납니다. 따라서, 더 큰 용량의 파일을 다룰 수 있습니다.

단점: 더 많은 저장 공간이 필요합니다. 더 큰 바이트를 사용하면 그만큼 더 많은 저장 공간을 필요로 하므로, 저장장치의 용량이 증가해야 합니다. 또한, 처리 속도가 느려질 수 있습니다. 더 많은 바이트를 처리하려면 그만큼 더 많은 시간이 필요하므로, 컴퓨터의 처리 속도가 느려질 수 있습니다.

따라서, 비트와 바이트의 크기는 필요한 정보의 양과 정밀도, 그리고 사용할 수 있는 자원 등을 고려하여 적절하게 선택해야 합니다.

2. 부트로더 개발

- 정의

부트로더: 컴퓨터가 부팅될 때 실행되는 프로그램으로, 운영체제를 메모리로 로드하고 실행하는 작업을 담당. 이는 컴퓨터를 켜면 처음으로 실행되는 소프트웨어임.

- 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 시그니처로 0x55AA라는 값을 가지며, 이는 해당 섹터가 유효한 부트로더 코드를 포함하고 있음을 BIOS에 알려주는 역할을 합니다.

* 왜 부트로더의 핵심은 512Byte인가?

부트로더가 512 바이트여야 하는 이유는 역사적인 이유와 기술적인 이유가 복합적으로 작용합니다.

원래 IBM PC가 설계될 때, 플로피 디스크의 섹터 크기를 512 바이트로 정했습니다.

부트로더는 컴퓨터가 켜질 때 가장 먼저 읽는 섹터인 부트 섹터에 저장되는데, 이 부트 섹터의 크기가 바로 512 바이트였습니다. 이런 설정은 이후 하드 드라이브와 같은 저장 매체의 표준으로 자리 잡았습니다.

따라서 BIOS(기본 입력/출력 시스템)는 첫 번째 섹터(부트 섹터)를 읽어서 그 안에 있는 부트로더를 메모리로 가져오는데, 이때 512 바이트만 읽도록 설계되어 있습니다.

만약 부트로더가 512 바이트보다 크거나 작다면, BIOS는 올바르게 부트로더를 읽지 못하고, 시스템은 부팅 과정을 올바르게 진행하지 못할 것입니다.

* 왜 부트로더의 끝 바이트는 0x55AA이어야 하는가?

부트로더의 끝 부분에 위치하는 0x55AA는 '시그니처 바이트(signature bytes)' 혹은 '매직 넘버(magic number)'라고 불립니다. 이 16 비트의 값은 부트 섹터의 맨 마지막 두 바이트에 위치하며, 시스템의 BIOS가 해당 섹터가 실제로 부트 가능한 부트 섹터인지를 확인하기 위한 용도로 사용됩니다.

BIOS가 컴퓨터를 부팅할 때, 저장 매체에서 부트 섹터를 메모리로 로드하고, 이 섹터의 마지막 두 바이트를 검사합니다. 만약 이 위치에 0x55AA가 있으면, BIOS는 이 섹터에 유효한 부트로더가 있다고 판단하고 실행을 시작합니다. 이는 일종의 검증 메커니즘으로, BIOS가 부팅할 수 있는 유효한 코드를 찾았는지를 확인하기 위한 것입니다.

만약 이 값이 없거나 다르면, BIOS는 그 섹터를 부팅 가능한 코드가 아니라고 간주하고 다음 부팅 가능한 장치를 찾아서 부팅 과정을 계속하게 됩니다. 그래서 이 시그니처는 부트 섹터가 실제로 컴퓨터를 시작할 수 있는 코드를 담고 있음을 보증하는 중요한 역할을 합니다.

* 섹터란?

하드디스크에서 데이터가 저장되는 곳

* BIOS란?

운영체제 중 가장 기본적인 소프트웨어이자 컴퓨터의 입출력을 처리.

사용자가 컴퓨터를 켜면 시작되는 프로그램으로 하드웨어와 컴퓨터 운영체제 사이의 저수준의 직접적인 통신을 담당

ASM 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.

세그먼트란?

세그먼트: 메모리의 연속적인 일부분

(세그먼트 방식은 초기 x86 아키텍처에서 메모리를 더 작은 단위로 나누어 관리하는데 사용)

세그먼트 레지스터에 값을 로드하는 것은 cpu에게 메모리의 어떤 부분을 참조할지 알려주는 것.

부트로더가 실행될 때 cs레지스터에 0x07C0 을 로드하면, 이후 실행되는 코드가 0x07C0 이 가리키는 메모리 영역에서 시작되는 것으로 cpu가 인식.

jmp 0x07C0: start → cpu의 실행흐름을 세그먼트 0x07C0 의 start 라벨 위치로 이동시키라는 지시.

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 (times 디렉티브는 주어진 명령을 특정 횟수만큼 반복하라는 지시어). \$\$는 이 섹션의 시작 주소를 나타내고, \$는 현재 주소를 나타냅니다. 따라서 (\$-\$\$)는 현재까지 코드의 크기를 나타냅니다. 510에서 이 값을 빼서 나머지 부분을 0으로 채우라는 의미입니다. 이렇게 하면 부트 시그니처를 제외한 나머지 부분을 적절하게 채울 수 있습니다.

db란?

Define byte의 줄임말로, 한 바이트(8 비트) 크기의 데이터를 정의하고 메모리에 할당 (ex: db 0x55는 메모리에 55라는 값을 가진 한 바이트를 할당)

```
dw 0xAA55
```

마지막으로 dw 0xAA55는 실제 부트 시그니처를 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.

dw란?

Define word의 줄임말로, 두 바이트(16 비트) 크기의 데이터를 정의하고 메모리에 할당 (ex: dw 0xAA55는 메모리에 AA55라는 값을 가진 두 바이트를 저장)

∴ times 510-(\$-\$\$) db 0으로 510 바이트를 0으로 채우고, dw 0xAA55로 마지막 2 바이트에 시그니처 바이트를 추가

시그니처 바이트: 부트 섹터의 유효성을 확인하기 위해 사용되는, 특정한 값을 가진 바이트들을 의미 (0x55와 0xAA → 0x55AA)

왜 한 번에 512 바이트를 0으로 안채우고 마지막 2 바이트에 시그니처 바이트를 추가하는가?

BIOS가 컴퓨터를 부팅할 때 부트 섹터를 읽고, 그 섹터가 유효한 부팅 가능한 코드를 포함하고 있는지 확인하는데, 이때 마지막 2 바이트에 위치한 시그니처를 체크함. 만약 이 시그니처가 없거나 다른 값이라면, BIOS는 그 섹터를 부팅 가능한 코드가 아니라고 간주하고 다른 부팅 장치를 찾게 됨.

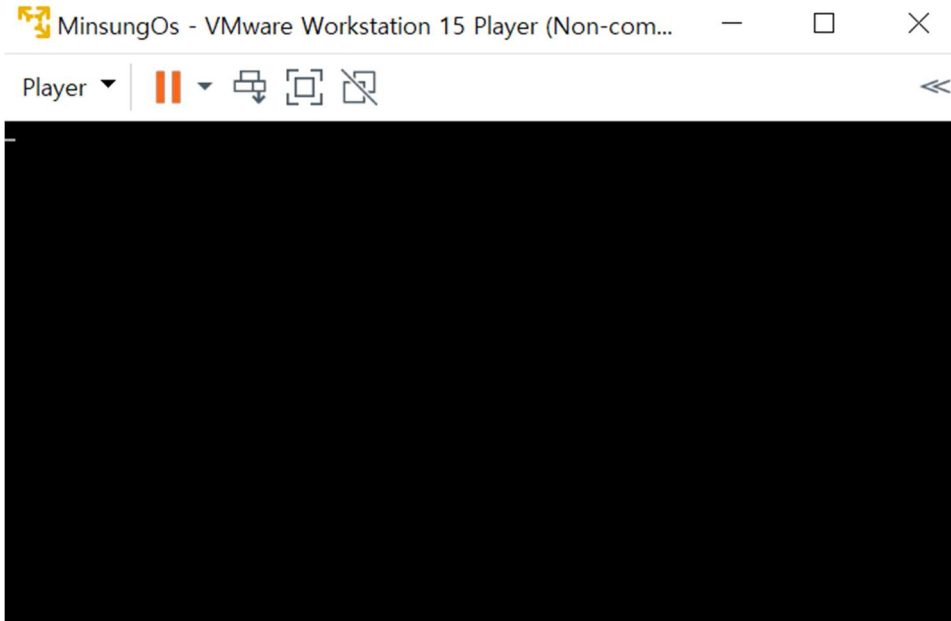
요약하자면, 이 코드 조각은 CPU가 16 비트 모드에서 실행하도록 설정하고, BIOS가 부트로더를 메모리의 기대 위치에 로드했음을 보정하며, 무한 루프에 들어가서 아무것도 하지 않는 간단한 부트로더의 예시입니다.

Boot_start.asm을 nasm을 이용하여 Boot_start.img로 컴파일합니다.

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

(VMware에 올리기 위하여 디스크 이미지로 컴파일)

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.

```

ASM 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.

ax 레지스터: 범용 레지스터 중 하나로, 주로 산술 연산, 데이터 이동, 입출력 명령 등에 사용
 세그먼트 레지스터: 메모리를 세그먼트로 분할하여 접근하는 데 사용되는 특별한 종류의 레지스터
 es 세그먼트 레지스터: 데이터를 저장하기 위한 추가적인 세그먼트를 가리키는 데 사용

(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
```

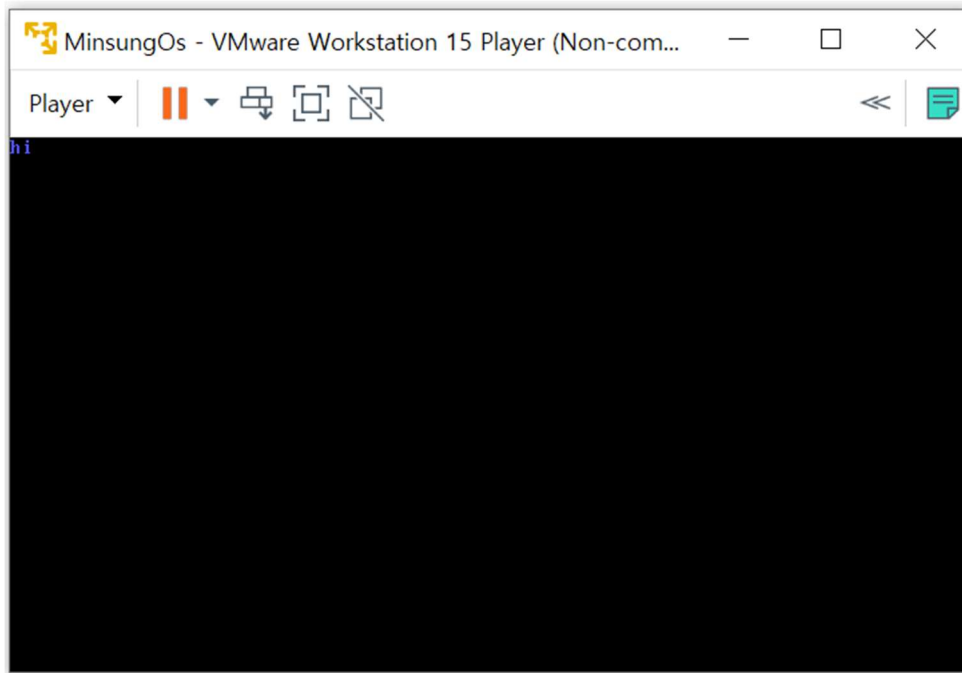
이 다음 두 줄은 두 번째 문자 셀에 'i' 문자를 쓰고, 동일한 속성을 적용합니다

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.

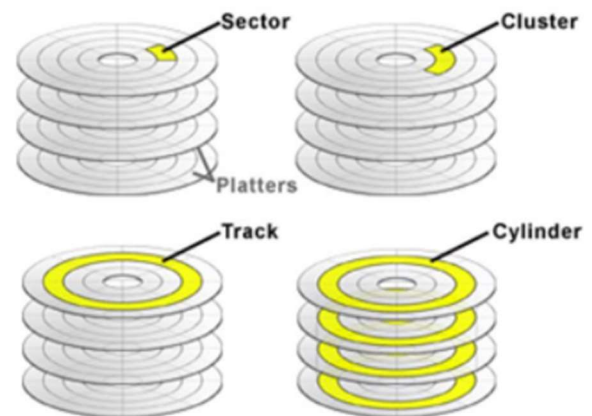
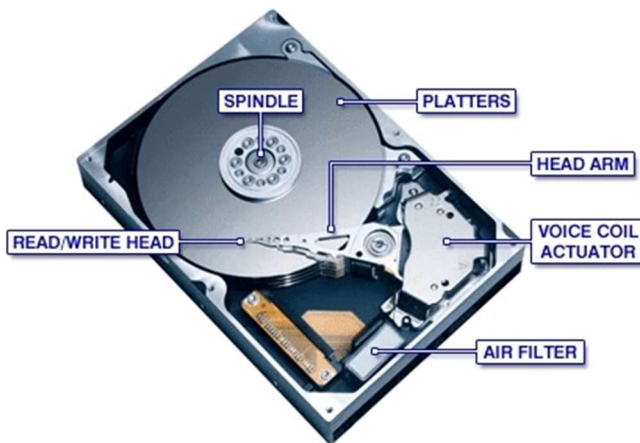


실행하게 된다면 작성한 코드대로 무한 루프에 들어가서 'hi'라는 글자를 하늘색 전경과 검정색 배경으로 표시하는 것이 뜰 것입니다.

이로서 하드디스크에서 512Byte를 읽어와 모니터에 hi를 출력하는 Os를 만들어냈습니다.

3. 하드디스크 읽기 모듈 개발

- 하드디스크의 실린더와 헤드



하드디스크는 여러 개의 원판으로 이루어져 있고, 각 원판은 미세한 트랙들로 나뉘어져 있습니다. 이 트랙들은 일종의 저장 공간으로, 데이터가 저장되는 곳입니다.

실린더: 하드디스크의 모든 원판들은 같은 중심축을 공유하며, 이를 통해 원판들이 회전합니다. 이 때, 모든 원판들을 통과하는 같은 반경에 위치한 트랙들의 집합을 '실린더'라고 부릅니다. 즉, 실린더는 원판들의 수직선을 통과하는 트랙들의 집합입니다.

헤드: 헤드는 데이터를 읽고 쓰는 역할을 하는 장치입니다. 하드디스크의 각 원판마다 헤드가 위치하며, 원판이 회전할 때 헤드는 원판 위에서 떠 있게 되어 원판의 트랙 위를 따라 이동하게 됩니다. 헤드는 트랙 위에 위치한 데이터를 읽거나 쓸 수 있습니다.

- 섹터

섹터란 무엇인가?

섹터: 자기 디스크나 광 디스크의 트랙의 일부로, 고정된 양의, 사용자 접근이 가능한 데이터를 저장하고 있음.

섹터는 몇부터 시작하며 몇까지 있는가?

저장 매체에서 섹터는 0 부터 시작함.

이론적으로 120GB 하드 드라이브에는 약 2 억 3 천만개의 512 바이트 섹터가 있음. 저장 장치의 섹터는 0 부터 시작하여 번호가 매겨지므로, 마지막 섹터의 번호는 총 섹터 수에서 1 을 뺀 값이 됨.

섹터 0 은 뭔가? 섹터 1 은 뭔가?

섹터 0: 시스템을 부팅하기 위한 초기 부트 코드와 파티션 정보를 포함합니다. 이 섹터는 부팅 가능한 미디어에 반드시 필요한 섹터입니다.

섹터 1 이후: 일반적인 데이터 저장 용도로 사용되며, 여기에는 운영 체제, 응용 프로그램, 사용자 파일 등이 저장됩니다.

그냥 섹터 0 에 만들지, 왜 섹터 1 을 또 만들어서 저장하는건가? 이렇게 하면 시간 복잡도와 공간 복잡도가 더 늘어나는 것 아닌가?

크기 제한: 섹터 0, 즉 부트 섹터는 512 바이트의 고정된 크기를 가지고 있습니다. 이 안에는 부트로더의 일부, 파티션 테이블, 그리고 마지막 2 바이트에는 유효한 부트 섹터임을 나타내는 시그니처 값(0x55AA)이 들어갑니다. 현대의 부트로더는 이보다 훨씬 큰 크기를 가지므로, 섹터 0 에는 전체 부트로더를 저장할 수 없습니다.

부팅 과정: 컴퓨터가 부팅할 때 BIOS 또는 UEFI는 섹터 0 을 메모리의 특정 위치(예: 0x7C00)에 로드합니다. 여기에 있는 코드는 매우 기본적인 부팅 작업을 수행하고, 실제로 운영 체제를 메모리로 로드하는 보다 복잡한 부트로더의 나머지 부분은 추가 섹터에 저장됩니다.

모듈화와 관리: 부트로더를 여러 부분으로 나누어 관리함으로써 개발과 유지보수가 용이해집니다. 각 부분은 독립적으로 수정하거나 업데이트할 수 있으며, 이는 전체 코드베이스의 복잡성을 줄여줍니다.

확장성: 섹터 1 이후의 섹터들을 사용함으로써, 부트로더는 필요한 만큼의 추가 공간을 활용할 수 있습니다. 이를 통해 더 많은 기능을 제공하고, 다양한 시스템 환경에 적응할 수 있습니다.

호환성: 과거의 컴퓨터 시스템과의 호환성을 유지하기 위해, 섹터 0 의 구조는 변경하지 않고, 새로운 기능이나 더 큰 용량이 필요할 때 추가 섹터를 사용합니다.

시간 복잡도와 공간 복잡도가 증가하는 것은 맞지만, 이는 부팅 과정의 신뢰성과 유연성을 향상시키기 위한 필요한 트레이드 오프입니다. 부트로더의 기능과 안정성은 시스템의 부팅 과정에 매우 중요하므로, 이러한 구조적인 설계는 정당화됩니다.

컴퓨터의 섹터 번호는 일반적으로 0 부터 시작하지만, BIOS 인터럽트 int 13h를 이용한 디스크 접근에서는 섹터 번호가 1 부터 시작합니다. 이는 BIOS의 디스크 서비스를 사용할 때의 관례입니다.

- 하드디스크 내 특정 섹터 읽기

처음 BIOS가 자동으로 읽었던 섹터를 "섹터 1"이라 합시다. 이 512Byte 코드는 물리주소 0x7C00 에 적재되어 있습니다. (원래는 섹터 0 이지만 여기서는 BIOS 인터럽트 int 13h를 이용한 디스크 접근이므로 섹터 1.)

x86 의 예약된 메모리 구조를 고려해서 물리 주소 0x10000(섹터 2 의 예약된 메모리 주소)에 "섹터 2"를 올려봅시다. (원래는 섹터 1 이지만 여기서는 BIOS 인터럽트 int 13h를 이용한 디스크 접근이므로 섹터 2).

ASM 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 루프**에 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.

(부트 섹터가 아니므로 510 바이트를 0 으로, 2 바이트를 부트 시그니처로 설정하지 않고 512 바이트를 0 으로 채움.)

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.

ASM 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.

bx 레지스터:

x86 아키텍처에서 사용되는 범용 레지스터 중 하나로 메모리 주소를 지정하는데 주로 사용.

여기서는 오프셋 레지스터로 사용되고 있음. 세그먼트:오프셋 방식에서 메모리 주소를 지정할 때, 세그먼트 레지스터(여기서 es)는 메모리의 기본 주소를 가리키고, bx 레지스터는 그 기준 주소로부터 얼마나 떨어진 위치에 있는지를 나타내는 오프셋 제공.

mov bx, 0 명령은 bx 레지스터 값을 0 으로 설정, 이는 es:bx 조합에서 가리키는 세그먼트의 시작지점을 참조하겠다는 의미 → es: 0x1000, bx: 0 → $es * 16 + bx = 0x10000$ (이 주소는 디스크에서 데이터를 읽어올 때 데이터를 저장할 메모리 위치로 사용)

```
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 레지스터: BIOS 인터럽트 서비스를 호출할 때 사용되는 '기능 번호'를 저장하는데 사용.

mov ah, 2 명령은 ah 레지스터에 2 라는 값을 로드함. 여기서 2 는 디스크에서 섹터를 읽어오라는 BIOS 인터럽트 int 13h의 기능번호임.

al 레지스터: BIOS의 디스크 서비스 인터럽트를 사용할 때, 읽어들이 섹터의 수를 지정하는데 사용.

(int 13h는 디스크 관련 작업을 수행하는 BIOS 인터럽트로 여러 기능을 수행할 수 있으며, 그 중 하나가 디스크에서 데이터를 읽어오는 것임. al 레지스터는 이 인터럽트를 사용할 때 얼마나 많은 섹터를 읽을 것인지 BIOS에 알려주기 위해 사용,)

ch 레지스터: BIOS 디스크 서비스에서 데이터를 읽어올 때 실린더 번호를 지정하는데 사용.

(mov ch, 0 에서 ch 레지스터 값을 0 으로 설정함. 여기서 ch는 읽어 올 데이터가 위치한 하드디스크의 실린더 번호를 나타냄. 실린더 번호가 0 이라는 것은 하드디스크의 가장 바깥쪽 트랙에서 데이터를 읽겠다는 뜻임.)

왜 하드디스크의 가장 바깥쪽 트랙에서 데이터를 읽는가?

접근 속도: 하드디스크의 바깥쪽 트랙은 내부 트랙에 비해 선형 속도가 빠릅니다. 하드디스크는 회전 속도가 일정하기 때문에 바깥쪽 트랙이 더 길어서 한 번 회전할 때 더 많은 데이터를 읽거나 쓸 수 있습니다. 이로 인해 데이터 전송률이 높아져서 접근 속도가 향상됩니다.

효율적인 공간 사용: 초기에 하드디스크를 포맷하거나 데이터를 저장할 때 바깥쪽 트랙부터 사용하는 것이 일반적입니다. 이는 바깥쪽 트랙이 더 많은 데이터를 저장할 수 있기 때문에 효율적으로 디스크 공간을 활용할 수 있게 해줍니다.

호환성과 표준: 과거의 BIOS 기반 시스템들에서는 부트 섹터나 중요한 시스템 파일들을 하드디스크의 가장 바깥쪽 실린더에 위치시키는 것이 일반적이었습니다. 이는 시스템이 부팅할 때 더 빠르게 필요한 파일에 접근할 수 있도록 하기 위함입니다.

cl 레지스터: 디스크 읽기 작업을 위한 BIOS 인터럽트 int 13h에서 사용되는 레지스터로 저장장치에서 읽기 작업을 할 때 시작할 섹터의 번호를 지정하는데 사용.

(mov cl, 2 에서 하드디스크의 첫번째 섹터 다음에 오는 섹터 2 를 읽으라고 지시하는 것.)

dh 레지스터: 디스크 읽기 작업을 위한 BIOS 인터럽트 int 13h에서 사용되는 레지스터로 헤드 번호를 지정하는데 사용.

mov dh, 0 명령어는 dh 레지스터를 0 으로 설정합니다. 이 경우에는 하드 드라이브의 첫 번째 헤드를 사용하겠다는 것을 의미합니다. 하드 드라이브에서 데이터를 읽거나 쓸 때, 어떤 플래터의 어느 면을 사용할 것인지를 결정하는 것이 바로 이 헤드 번호입니다.

저장 장치의 각 플래터는 보통 두 개의 면을 가지고 있으며, 각 면에는 헤드가 하나씩 있습니다. 따라서 헤드 번호는 데이터가 위치한 플래터의 면을 지정합니다. int 13h 인터럽트를 사용하여 디스크에서 데이터를 읽거나 쓸 때, DH는 해당 플래터의 어느 면에서 데이터를 처리할 것인지를 BIOS에 알려줍니다.

하드 디스크의 각 플래터는 양쪽 면에 데이터를 저장할 수 있고, 각 면마다 하나의 헤드가 데이터를 읽고 쓰는 역할을 합니다. 따라서 하드 디스크에 플래터가 하나 있고 양면을 사용한다면 헤드는 두 개가 됩니다. 하지만 실제 하드 디스크에는 여러 개의 플래터가 있을 수 있으며, 각각의 플래터는 양면을 사용할 수 있습니다.

예를 들어, 하드 디스크에 3 개의 플래터가 있고 각 플래터가 양면을 사용할 수 있다면, 총 6 개의 헤드가 있을 것입니다. 이 경우 DH 레지스터는 0 부터 5 까지의 값을 가질 수 있으며, 각 값은 하드 디스크의 다른 헤드를 가리킵니다.

mov dh, 0 외에도 다른 헤드를 가리키기 위해 DH 레지스터에 다른 값을 설정할 수 있습니다.

왜 하드 드라이브의 첫 번째 헤드를 사용하는가?

부트스트랩 로딩: 컴퓨터 부팅 과정에서 BIOS는 부트스트랩 로더를 로딩하기 위해 하드디스크의 첫 번째 섹터인 MBR(Master Boot Record)에 접근합니다. MBR은 대개 첫 번째 플래터의 첫 번째 헤드 아래에 위치합니다. 따라서, 부팅 과정에서는 첫 번째 헤드를 사용하는 것이 표준 절차입니다.

효율성: 하드디스크의 첫 번째 트랙은 데이터 접근 속도가 상대적으로 빠른 편입니다. 따라서, 운영체제나 프로그램은 자주 사용되는 데이터나 필수 시스템 파일을 첫 번째 헤드에 위치시킬 수 있어서, 더 빠른 데이터 접근이 가능합니다.

단순성과 호환성: 첫 번째 헤드를 사용하면 다양한 하드웨어 구성과 시스템에서 호환성을 유지하기 쉽습니다. 특히 과거의 시스템에서는 복잡한 하드웨어 구성보다 단순한 접근 방법을 선호했습니다.

dl 레지스터: BIOS 인터럽트 int 13h를 사용하여 디스크 드라이브에서 데이터를 읽을 때, 어떤 드라이브에서 데이터를 읽을 것인지를 지정하는 데 사용.

mov dl, 0 명령어는 DL 레지스터에 0 을 로드함. 이는 첫 번째 드라이브, 일반적으로 플로피 드라이브를 의미함. (컴퓨터 시스템에서는 드라이브 번호가 0 부터 시작하여 하드 드라이브, CD-ROM 드라이브 등에 순차적으로 할당됨. DL 레지스터는 이러한 드라이브 번호를 인터럽트 호출에 전달하여, 어떤 물리적 드라이브를 사용할 것인지를 BIOS에 알려주는 역할을 함.)

인터럽트란?

인터럽트는 컴퓨터에서 발생하는 일종의 신호로서, CPU에게 특정 이벤트가 발생했음을 알리는 역할을 함. 이는 현재 실행 중인 작업을 일시 중단하고, 인터럽트 처리 루틴을 실행한 후 원래의 작업으로 돌아가게 함.

이 코드에서 인터럽트를 사용하는 이유:

이 코드에서 인터럽트는 디스크에서 데이터를 읽어오는 작업을 수행하기 위해 사용. 인터럽트는 하드웨어와 소프트웨어 사이의 인터페이스 역할을 하며, 디스크와 같은 하드웨어 장치와 상호작용하기 위한 방법임.

인터럽트 13h란?

인터럽트 13h는 BIOS에서 제공하는 디스크 관련 서비스를 호출하는 인터럽트임. 디스크에서 데이터를 읽거나 쓰는 작업을 수행할 수 있음.

인터럽트의 종류:

- 인터럽트 10h: 이 인터럽트는 비디오 서비스를 제공합니다. 화면 모드 설정, 문자 출력, 그래픽 출력 등의 기능을 수행할 수 있습니다.
- 인터럽트 16h: 이 인터럽트는 키보드 서비스를 제공합니다. 키보드 버퍼에서 문자를 읽어오거나 키보드 상태를 체크하는 기능을 수행할 수 있습니다.
- 인터럽트 Ah: 이 인터럽트는 시스템 타이머 서비스를 제공합니다. 시스템 시간을 읽어오거나 설정하는 기능을 수행할 수 있습니다.
- 인터럽트 21h: 이 인터럽트는 DOS 서비스를 제공합니다. 파일 열기, 파일에서 읽기, 파일에 쓰기 등의 기능을 수행할 수 있습니다.

이 외에도 다양한 인터럽트가 있으며, 각각은 특정 시스템 서비스를 제공함.

이 코드에서 왜 13h를 사용하는가?

이 코드는 디스크에서 데이터를 읽어오는 작업을 수행하기 위해 인터럽트 13h를 사용함. 인터럽트 13h는 디스크 I/O 작업을 수행하는 서비스를 제공하기 때문. 이 코드에서는 첫 번째 디스크 드라이브의 2 번째 섹터부터 데이터를 읽어오는 작업을 수행하게 됨.

```
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 둡니다.

반면, Sector2_read.asm 파일은 추가적인 부트 시그니처가 필요 없으므로 전체 512 바이트를 데이터 또는 0 으로 채웁니다.

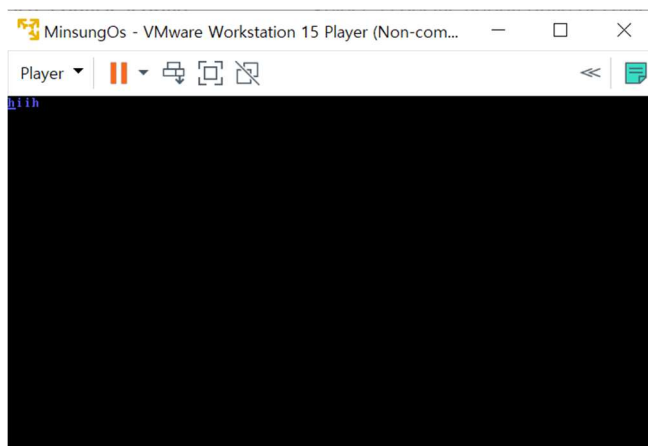
이제 두 개의 파일, Boot_read.asm 과 Sector2_read.asm 이 준비되었습니다. 이 둘을 각각 NASM으로 컴파일하면 각각의 img 파일을 얻을 수 있습니다.

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.

4. 모드 전환 모듈 개발

- 리얼모드(Real Mode)와 보호 모드(Protected Mode)에 대한 기본 개념

리얼 모드(Real Mode):

- 리얼 모드는 8086 CPU 아키텍처의 초기 운영 모드입니다.
- 이 모드는 CPU가 처음 전원이 켜졌을 때의 기본 모드로 설정됩니다.
- 리얼 모드는 16 비트 모드로, CPU는 16 비트 데이터와 명령어를 처리하고, 20 비트 주소선을 통해 최대 1MB의 메모리에 접근할 수 있습니다.
- 세그먼트:오프셋 방식을 사용하여 메모리 주소를 계산합니다.
- 메모리 보호 기능이 없으며, 모든 프로그램은 전체 메모리 공간에 대한 완전한 접근 권한을 가집니다.
- 멀티태스킹, 가상 메모리, 페이징 등의 고급 기능을 지원하지 않습니다.

- 호환성을 위해 여전히 존재하지만, 현대 컴퓨터에서는 운영 체제가 부팅 과정에서 보호 모드로 전환하기 전까지만 잠깐 사용됩니다.

왜 리얼모드는 16 비트인가?

리얼 모드는 1978 년에 인텔이 출시한 8086 프로세서에 처음 도입되었습니다.

이 프로세서는 16 비트 마이크로프로세서였으며,

당시의 컴퓨터 아키텍처와 소프트웨어는 주로 16 비트 데이터와 명령어 세트에 맞춰져 있었습니다.

리얼 모드는 8086 호환성을 유지하기 위해 설계되었기 때문에, 자연스럽게 16 비트로 제한됩니다.

기술적 제약:

8086 CPU는 16 비트 레지스터와 16 비트 데이터 버스를 가지고 있었고, 이것은 CPU가 한 번에 16 비트만 처리할 수 있음을 의미합니다.

또한, 이 CPU는 20 비트 주소선을 가지고 있었는데, 이를 통해 2^{20} , 즉 1MB의 메모리 주소 공간에 접근할 수 있었습니다.

리얼 모드에서는 이 20 비트 주소 공간을 효율적으로 활용하기 위해 세그먼트:오프셋 방식이 사용되었습니다.

이러한 기술적인 제약과 설계 선택은 초기 컴퓨터 하드웨어의 한계와 그 당시의 기술 수준을 반영합니다.

당시에는 16 비트 처리가 일반적이었고, 이는 단순히 오늘날의 관점에서 볼 때의 제약이 아니라, 그 당시의 기술적인 현실과 일치했습니다.

다시 말해, 리얼 모드의 16 비트 제한은 초기 x86 아키텍처의 기본 설계와 기능적 한계에서 비롯된 것으로, 나중에 CPU 아키텍처가 발전함에 따라 32 비트와 64 비트 시스템으로 확장되었습니다.

왜 리얼모드는 CPU가 처음 전원이 켜졌을 때의 기본 모드로 설정되는가?

리얼 모드가 CPU가 처음 전원을 켤 때의 기본 모드로 설정되는 이유는 주로 초기 x86 CPU 아키텍처의 설계 철학 때문입니다. 초기의 x86 프로세서인 8086 과 8088 은 리얼 모드만을 지원했고, 이후의 프로세서들은 하위 호환성을 유지하기 위해 리얼 모드로 시작하는 것을 기본으로 삼았습니다.

하위 호환성을 유지하면, 새로운 프로세서가 출시되어도 기존의 소프트웨어와 운영 체제가 여전히 작동할 수 있도록 보장합니다. 이는 특히, 컴퓨터가 부팅 과정을 시작할 때 중요한데, 대부분의 초기 부팅 코드는 리얼 모드에서 동작하도록 작성되었습니다. 이러한 코드는 BIOS 루틴과 같은 기본적인 시스템 설정이나 부트 로더를 포함하며, 리얼 모드에서의 단순한 메모리 주소 지정 방식을 사용합니다.

전원이 켜지면 CPU는 매우 기본적인 상태로 초기화되며, 리얼 모드에서는 모든 메모리 주소가 실제 물리 메모리 주소와 직접적으로 일치합니다. 이는 복잡한 메모리 관리 기능이나 보호 메커니즘 없이 시스템의 기본적인 부팅 과정을 시작하기 위한 이상적인 환경을 제공합니다. 그 후, 초기화 과정이 진행되고 필요한 시스템 자원이 설정되면, 운영 체제는 보다 고급 기능을 제공하는 보호 모드로 전환합니다.

무시하고 보호모드로 바로 시작하면 안되는가?

아니요, 현재의 x86 아키텍처 기반 CPU는 전원이 켜질 때 자동으로 리얼 모드로 시작합니다. 이는 하드웨어 설계에 내재된 기본 동작으로, CPU 제조사가 의도적으로 설정한 것입니다.

리얼 모드에서 시작하는 것은 여러 가지 이유가 있지만, 그 중 몇 가지는 다음과 같습니다:

호환성: 리얼 모드는 오래된 소프트웨어와 운영 체제가 여전히 기대하는 환경입니다. 이 모드는 8086 CPU의

동작을 에뮬레이트합니다, 그래서 초기 단계에서 리얼 모드로 시작하는 것은 이전 시스템과의 호환성을 보장합니다.

단순성: 리얼 모드는 CPU와 메모리의 관리가 매우 단순합니다. 복잡한 초기화 절차나 설정 없이, BIOS와 같은 기본적인 시스템 소프트웨어가 직접적으로 하드웨어에 접근할 수 있게 해 줍니다.

보호 모드로의 전환 준비: 보호 모드로 진입하기 전에 시스템이 올바른 상태에 있는지 확인하고 필요한 시스템 자원과 환경을 설정해야 합니다. 이를 위해 리얼 모드에서 시작하는 것이 필요합니다.

만약 CPU가 리얼 모드를 건너뛰고 보호 모드로 바로 시작할 수 있다면, 호환성 문제가 발생할 수 있으며, BIOS 루틴과 같은 초기 시스템 소프트웨어가 제대로 동작하지 않을 수 있습니다. 현대의 운영 체제는 이러한 초기 리얼 모드 환경에서 빠르게 시스템을 초기화하고 보호 모드 또는 롱 모드로 전환하여 고급 기능과 보호 기능을 활성화합니다.

리얼모드는 16 비트인데 어떻게 20 비트 주소 공간을 표현하는 방법으로 세그먼트:오프셋 방법을 사용하는가?

리얼 모드에서는 16 비트의 CPU를 사용함. 그래서 레지스터의 크기도 16 비트로 제한되어 있음.

이게 의미하는 건, CPU가 한 번에 처리할 수 있는 주소 범위가 0 부터 65,535(즉, $2^{16} - 1$)까지라는 거임.

하지만 이렇게 되면 메모리 주소 공간이 64KB로 제한되어 버림.

그래서, 인텔은 이 문제를 해결하기 위해 세그먼트:오프셋 방식을 도입함.

이 방식을 사용하면, 16 비트 세그먼트 레지스터와 16 비트 오프셋을 조합하여 실제로는 20 비트의 메모리 주소를 생성할 수 있음. 세그먼트 레지스터는 메모리의 시작 주소를 가리키고, 오프셋은 그 시작 주소로부터의 거리를 나타냄. 이렇게 함으로써, 최대 1MB(2^{20})의 메모리를 사용할 수 있음.

왜 리얼모드는 메모리 보호 기능이 없으며, 모든 프로그램은 전체 메모리 공간에 대한 완전한 접근 권한을 가지는가?

리얼 모드가 메모리 보호 기능을 제공하지 않는 이유는 그것이 설계된 시대의 컴퓨팅 환경과 아키텍처의 한계에 기인합니다.

설계 당시의 필요성: 리얼 모드는 1970년대 후반에 설계된 8086 CPU 아키텍처에 기반을 두고 있습니다. 당시의 컴퓨터는 주로 단일 사용자, 단일 태스크를 처리하는 데 사용되었으며, 오늘날처럼 복잡한 멀티태스킹 환경이 일반적이지 않았습니다. 따라서, 각 프로그램이 메모리의 어느 부분을 사용할지 엄격하게 관리할 필요성이 덜했습니다.

아키텍처의 단순성: 리얼 모드는 CPU 설계의 단순함을 유지하고자 했습니다. 메모리 보호와 같은 고급 기능은 추가적인 하드웨어 지원을 필요로 하며, 이는 당시의 프로세서에 더 많은 복잡성을 추가했을 것입니다. 리얼 모드는 직접적이고 단순한 메모리 접근 방식을 유지함으로써, 하드웨어 설계를 단순하게 유지할 수 있었습니다.

호환성: 리얼 모드는 오래된 소프트웨어와의 호환성을 유지하기 위해 계속해서 사용되었습니다. 이 모드에서는 기존의 8086 소프트웨어가 아무런 수정 없이 실행될 수 있었으며, 이는 초기 x86 호환 시스템에서 중요한 고려사항이었습니다.

하드웨어 자원의 제한: 초기의 마이크로컴퓨터는 제한된 하드웨어 자원을 가지고 있었고, 복잡한 운영 체제보다는 간단한 BIOS를 사용하여 기본적인 입출력과 부트로딩을 처리하는 데 초점을 맞췄습니다. 이러한 환경에서는 메모리 보호 기능이 큰 필요성을 가지지 않았습니다.

하지만 컴퓨터의 사용 환경이 발전하고 멀티태스킹, 네트워킹, 사용자 보안 등이 중요해지면서, 메모리 보호와 같은 기능이 필수적이 되었습니다. 이에 따라, 인텔은 80286 CPU부터 보호 모드를 도입하여 프로그램 간의 메모리 영역을 격리하고, 프로그램이 시스템의 다른 부분에 무분별하게 접근하는 것을 방지하는 등의 기능을 제공하기 시작했습니다.

왜 리얼모드는 멀티태스킹, 가상 메모리, 페이징 등의 고급 기능을 지원하지 않는가?

리얼 모드에서 멀티태스킹, 가상 메모리, 페이징 등의 고급 기능이 지원되지 않는 주된 이유는 그것이 설계된 시점과 목적에 있습니다.

초기 CPU 설계: 리얼 모드는 원래 8086 CPU를 위해 설계되었으며, 이 CPU는 단순한 컴퓨팅 작업을 수행하기 위한 기본적인 기능만 가지고 있었습니다. 이 초기 프로세서는 오늘날의 CPU처럼 복잡한 메모리 관리 기능을 내장하고 있지 않았습니다.

메모리 관리의 단순성: 리얼 모드는 모든 메모리 주소를 실제 물리 메모리 주소로 직접 매핑합니다. 이 단순한 접근 방식은 메모리 보호 구조, 가상 메모리 시스템, 페이징 메커니즘과 같은 복잡한 메모리 관리 기능을 지원하지 않습니다.

시스템의 복잡성과 자원: 당시의 컴퓨터 시스템은 제한된 자원을 가지고 있었고, 고급 기능을 지원하기 위한 충분한 처리 능력이나 메모리가 없었습니다. 멀티태스킹과 가상 메모리는 추가적인 하드웨어 지원과 복잡한 운영 체제의 관리 기능을 필요로 합니다.

운영 체제의 역할: 초기에는 대부분의 컴퓨팅 환경이 단일 작업을 수행하는 데 집중되어 있었으며, 운영 체제는 현재와 같은 복잡한 작업을 다루기 위해 설계되지 않았습니다. 멀티태스킹과 같은 기능은 후에 등장한 보호 모드와 같은 더 발전된 CPU 모드에서 운영 체제에 의해 구현되었습니다.

호환성 유지: 리얼 모드는 하위 호환성을 유지하기 위해 계속 유지되었습니다. 새로운 기능을 추가하는 대신, 리얼 모드는 기본적인 기능을 유지하면서 새로운 CPU 모드에서 고급 기능을 지원하도록 설계되었습니다.

이러한 이유로, 리얼 모드는 간단한 부팅과 기본적인 하드웨어 접근에 사용되며, 복잡한 시스템 관리는 보호 모드 또는 그 이후에 도입된 더 발전된 모드(예: 롱 모드)에서 처리됩니다.

보호 모드(Protected Mode):

- 보호 모드는 인텔 x86 아키텍처의 프로세서가 보다 복잡하고 세련된 운영 체제 기능을 지원하도록 만들어진 모드입니다.
- 처음으로 80286(286) 프로세서에서 도입되었으며, 이때는 16 비트 연산을 지원했습니다. 24 비트 주소 버스를 통해 최대 16MB의 메모리에 접근할 수 있었습니다.
- 80386(386) 프로세서부터는 보호 모드가 32 비트 연산을 지원하게 되었고, 32 비트 주소 버스를 통해 최대 4GB의 메모리에 접근할 수 있게 되었습니다.
- 보호 모드는 메모리 보호를 제공하여, 각 프로그램이나 프로세스가 독립적인 메모리 공간을 가짐으로써 시스템의 안정성을 높입니다.
- 세그먼테이션을 통해 메모리를 관리하며, 페이징 기능을 통해 가상 메모리를 구현합니다.
- 멀티태스킹을 지원하여, 여러 프로세스가 동시에 실행될 수 있도록 하드웨어 수준에서 컨텍스트 스위칭(태스크 스위칭)을 가능하게 합니다.
- 입출력 보호 기능을 통해 특정 프로세스의 시스템 자원 접근을 제어하고 관리합니다.

세그먼테이션이란?

- 세그먼테이션은 메모리를 다양한 크기의 세그먼트로 분할하는 방식입니다.
- 각 세그먼트는 특정 종류의 데이터 또는 프로그램 코드를 위해 할당되며, 시작 주소(base), 크기(limit), 그리고 일련의 권한과 같은 속성을 갖습니다.
- 예를 들어, 코드 세그먼트, 데이터 세그먼트, 스택 세그먼트 등이 있으며, 각각은 프로그램의 실행 코드, 변수, 함수 호출 스택 등을 위해 사용됩니다.
- 세그먼테이션은 메모리 보호를 강화하고, 프로그램 간 또는 프로그램과 운영 체제 간의 메모리 영역을 명확히

구분합니다.

페이징이란?

- 페이징은 메모리를 고정된 크기의 블록(페이지)으로 나누는 방식입니다.
- 시스템은 물리 메모리를 페이지라고 하는 동일한 크기의 블록으로 나누고, 가상 메모리도 동일한 크기의 페이지로 나눕니다.
- 페이지 테이블이라는 구조를 사용해 가상 페이지와 물리 페이지의 매핑(mapping)을 관리합니다.
- 페이징은 프로그램이 실제 물리 메모리의 연속적인 영역을 필요로 하지 않게 하므로 메모리 사용의 유연성을 증가시킵니다.

가상메모리란? 가상메모리를 구현하는 이유는?

가상 메모리는 컴퓨터 운영 체제가 사용하는 메모리 관리의 한 기법입니다. 이 기법은 물리적 메모리(RAM)의 크기를 초과하는 프로그램을 실행할 수 있게 해주고, 멀티태스킹 환경에서 프로세스 간 메모리 충돌을 방지합니다. 가상 메모리는 각 프로그램이 전체 메모리를 독립적으로 사용하는 것처럼 느끼게 하는 추상적인 메모리 레이어를 생성합니다.

가상 메모리를 구현하는 이유는 여러 가지가 있습니다:

1. 메모리 확장: 가상 메모리는 실제 물리적 메모리보다 더 많은 메모리를 필요로 하는 프로그램에게 추가적인 공간을 제공합니다. 이를 통해 물리적 메모리의 제한을 넘어선 작업을 할 수 있습니다.
2. 보안과 격리: 가상 메모리는 각 프로세스에게 독립된 메모리 공간을 제공함으로써, 한 프로세스의 오류가 다른 프로세스에 영향을 미치지 않도록 합니다. 이는 시스템의 안정성을 크게 향상시킵니다.
3. 효율적인 메모리 사용: 가상 메모리는 운영 체제가 메모리 사용을 최적화할 수 있게 해줍니다. 잘 사용되지 않는 메모리 영역을 디스크의 스왑 공간으로 옮기고, 필요할 때 다시 불러오는 방식으로 메모리를 더 효율적으로 사용할 수 있게 합니다.
4. 멀티태스킹: 여러 프로그램이나 프로세스가 동시에 실행될 때, 가상 메모리는 각각에게 충분한 메모리 공간을 제공함으로써, 동시에 여러 작업을 처리할 수 있게 합니다.

멀티태스킹이란?

멀티태스킹은 컴퓨터가 동시에 여러 작업을 실행하는 능력을 말합니다. 보호 모드에서 제공하는 멀티태스킹 기능은 하나의 프로세서가 여러 프로세스나 스레드를 거의 동시에 처리할 수 있도록 하며, 이는 시스템 자원을 효율적으로 사용하고 사용자 경험을 개선하는 데 중요한 역할을 합니다.

멀티태스킹을 가능하게 하는 핵심은 운영 체제의 스케줄러입니다. 스케줄러는 각 프로세스에 CPU 시간을 할당하고, 실행 중인 프로세스 사이를 빠르게 전환함으로써 동시에 여러 작업을 수행하는 것처럼 보이게 합니다. 이러한 프로세스 간 전환을 컨텍스트 스위칭(태스크 스위칭)이라고 합니다.

보호 모드에서의 멀티태스킹은 다음과 같은 특징을 가집니다:

1. 프로세스 격리: 각 프로세스는 독립된 메모리 공간을 할당받아, 다른 프로세스의 작업에 의해 방해받지 않습니다.
2. 효율적인 자원 관리: 운영 체제는 CPU 시간, 메모리, 입출력 장치 등의 자원을 각 프로세스 간에 적절히 분배하여 관리합니다.
3. 사용자 경험 향상: 사용자는 여러 애플리케이션을 동시에 열어두고 작업할 수 있으며, 이는 컴퓨터 사용의 효율성을 크게 높여줍니다.
4. 시스템 성능 최적화: 멀티태스킹은 시스템 자원을 최대한 활용하여, 프로세스 실행 대기 시간을 줄이고 전반적인 처리 능력을 향상시킵니다.

이러한 멀티태스킹의 구현은 현대 컴퓨팅 환경에서 필수적인 기능으로, 사용자가 여러 프로그램을 효율적으로 동시에 사용할 수 있게 하고, 서버와 같은 고성능 시스템에서는 많은 사용자 요청을 동시에 처리할 수 있도록 합니다.

입출력 보호 기능이란?

입출력 보호 기능은 프로그램이 컴퓨터의 하드웨어 자원을 잘못 사용하는 것을 막는 중요한 기능입니다. 이 기능은 프로그램이 특정 하드웨어에 접근하기 전에 적절한 권한을 가지고 있는지 확인합니다. 이는 하드웨어 자원을 보호하고, 프로그램 간의 간섭을 방지하여 시스템의 안정성과 보안을 유지하는 데 필수적입니다.

리얼 모드에서는 고급 기능을 지원하지 않는데, 왜 보호 모드에서는 고급 기능을 지원하는가?

리얼 모드는 초기 x86 프로세서의 운영 모드로, 8086 CPU와 호환성을 유지하기 위해 설계되었습니다. 이 모드는 단순하고 직접적인 메모리 접근 방식을 가지고 있으며, CPU의 기능을 1MB의 메모리 공간과 16 비트 처리에 제한합니다. 리얼 모드는 주로 단일 사용자, 단일 태스크 시스템에서 사용되었기 때문에, 복잡한 메모리 관리나 보안 기능은 필요하지 않았습니다.

반면에, 보호 모드는 80286 CPU와 함께 도입되어, 멀티태스킹과 같은 고급 기능을 지원하도록 설계되었습니다. 보호 모드의 주요 목적은 시스템의 안정성과 보안을 향상시키기 위한 메모리 보호와, 더 넓은 주소 공간을 통한 더 큰 메모리 용량에 접근할 수 있게 하는 것이었습니다. 따라서 이 모드에서는 고급 기능을 지원하게 됩니다.

64 비트 확장 모드(룽 모드):

64 비트 x86 프로세서에서 도입된 룽 모드는 64 비트 연산과 주소 지정을 지원합니다.

룽 모드를 통해 프로세서는 이론상 2^{64} 바이트의 주소 공간에 접근할 수 있지만, 실제로는 프로세서와 운영 체제, 그리고 시스템의 하드웨어 구성에 의해 제한됩니다.

룽 모드는 보호 모드의 모든 기능을 포함하면서도, 64 비트 운영 체제가 필요로 하는 더 큰 메모리 공간과 더 빠른 데이터 처리 능력을 제공합니다.

왜 이론상 2^{64} 바이트의 주소 공간에 접근할 수 있지만, 실제로는 프로세서와 운영 체제, 그리고 시스템의 하드웨어 구성에 의해 제한되는가?

64 비트 확장 모드(룽 모드)에서 이론적으로 가능한 주소 공간은 2^{64} 바이트입니다. 이는 엄청난 양의 메모리(약 18 엑사바이트)에 접근할 수 있음을 의미합니다. 그러나 실제 시스템에서 이 모든 주소 공간을 사용할 수 없는 여러 가지 이유가 있습니다:

1. 하드웨어 제한: 현재 시장에 나와 있는 대부분의 64 비트 CPU는 전체 64 비트 주소 공간을 물리적으로

지원하지 않습니다. CPU 제조사는 시스템의 실제 요구와 기술적인 제약을 고려하여, 주소 가능한 물리적 메모리의 양을 제한합니다. 대부분의 시스템은 48 비트에서 52 비트 정도의 주소 공간만을 사용하며, 이는 여전히 수 테라바이트(TB)에서 수백 테라바이트의 메모리를 지원하는 양입니다.

2. 운영 체제의 제한: 운영 체제는 하드웨어의 능력을 기반으로 메모리 관리 기능을 구현합니다. 대부분의 운영 체제는 실제 시스템에서 필요로 하는 메모리 양에 맞게 메모리 관리 기능을 최적화하기 때문에, 전체 64 비트 주소 공간을 사용하지 않습니다.

3. 실제 메모리의 제한: 시스템의 메인보드와 메모리 슬롯의 물리적 한계로 인해, 실제로 시스템에 장착할 수 있는 메모리 양에는 제한이 있습니다. 현재 기술로는 시스템에 수백 테라바이트의 메모리를 설치하는 것이 불가능합니다.

4. 경제적인 제한: 심지어 기술적으로 가능하더라도, 엄청난 양의 메모리를 시스템에 장착하는 것은 경제적으로 타당하지 않을 수 있습니다. 대부분의 애플리케이션과 시스템은 그렇게 많은 메모리를 필요로 하지 않으며, 이에 따라 이론적인 최대치보다 훨씬 적은 양의 메모리가 실제로 사용됩니다.

따라서 실제 시스템에서는 하드웨어의 물리적 제한, 운영 체제의 설계, 실제 메모리 요구 사항, 그리고 비용 효율성 등 여러 요인이 64 비트 주소 공간의 사용을 제한하게 됩니다.

결론적으로, 보호 모드는 리얼 모드의 제한을 넘어서는 다양한 고급 기능을 제공하며, 컴퓨터의 부팅 과정에서 리얼 모드에서 보호 모드로 전환함으로써, 현대 운영 체제가 안정적이고 효율적으로 작동할 수 있는 환경을 조성합니다. 286 에서 시작된 16 비트 보호 모드는 386 에서 32 비트로 확장되었으며, 현대의 64 비트 프로세서에서는 더욱 발전된 롱 모드로 더 큰 메모리와 더 빠른 처리 능력을 지원합니다.

- 리얼모드(16 비트) 환경에서의 세그먼트:오프셋 구조

"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.

ASM 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 ; 에러라면 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.

데이터 세그먼트: 프로그램의 데이터를 저장하는 메모리 영역, 일반적으로 변수, 배열, 구조체 등 프로그램 실행 중 생성되고 사용되는 데이터를 저장하는데 사용

(16 비트 리얼모드에서는 메모리 주소를 세그먼트와 오프셋으로 나누어 참조하며, 세그먼트는 메모리의 큰 블록을 가리키고, 오프셋은 해당 세그먼트 내의 특정 주소 지정)

ds 레지스터: 데이터 세그먼트 레지스터, 데이터 세그먼트의 세그먼트 부분을 가리키는 데 사용되며, 프로그램이 데이터 메모리에 접근할 때 기준으로 활용

ds를 사용하는 이유:

1. 세그먼트 초기화: 부트스트랩 로더가 실행될 때, 코드는 cs (코드 세그먼트 레지스터)에 이미 로드되어 있음. ds를 cs와 같은 값으로 설정함으로써, 프로그램의 코드와 데이터가 동일한 세그먼트 내에 있음을 보장함. 이렇게 하면 어셈블리어 프로그램이 데이터에 접근할 때 ds를 기반으로 정확한 주소를 계산할 수 있음.

2. 데이터 참조:

코드 세그먼트 내에서 실행되는 명령어들이 데이터를 참조할 때, ds 레지스터는 세그먼트의 기준 주소를 제공함. 예를 들어, mov ax, [someData] 와 같은 명령은 ds:someData 메모리 위치에서 값을 ax 레지스터로 로드하게 됨.

이 코드에서는 ds를 명시적으로 사용하는 부분은 없지만, 데이터에 접근하는 다른 명령어들이 내부적으로 ds

값을 사용하여 올바른 메모리 위치를 찾습니다. ds가 제대로 설정되지 않으면, 데이터에 접근할 때 잘못된 메모리 위치를 참조하게 되어 프로그램이 올바르게 실행되지 않을 수 있습니다.

간단히 말해, 데이터 세그먼트는 프로그램 데이터를 위한 메모리 영역이고, ds 레지스터는 그 영역에 접근하기 위한 기준점을 제공합니다.

위의 `Boot_read.asm`에서는 ds를 사용하지 않다가 `Boot_protect.asm`에서는 왜 ds를 사용할까?

왜냐하면 이는 추가적인 메모리 접근이 필요하기 때문입니다. 여기서는 GDT(Global Descriptor Table)를 초기화하고 보호 모드로 전환하는 과정을 수행합니다. 이 과정에서 DS와 다른 세그먼트 레지스터들(ES, FS, GS, SS)을 새로 설정한 GDT에 맞춰 초기화하는 작업이 필요합니다.

세그먼트 레지스터들은 메모리의 특정 부분을 가리키는데 사용되며, x86 아키텍처에서는 코드, 데이터, 스택, 그리고 추가 세그먼트들을 위해 각기 다른 세그먼트 레지스터를 사용합니다. 보호 모드에서는 메모리 접근을 위한 규칙이 더 엄격하므로 DS와 다른 세그먼트 레지스터들을 적절히 설정해야 합니다.

결론적으로, DS를 설정하는 것은 GDT를 초기화하고 보호 모드로 전환하는 복잡한 과정을 수행하기 위해 필요하기 때문입니다.

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

여기서 `mov ax, 0xB800` 명령은 ax 레지스터에 0xB800 값을 넣습니다. 0xB800은 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 2번 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 0x7C00으로 설정되어 있을 것입니다. 그 결과, 코드는 의도하지 않은 메모리 위치에 데이터를 쓰게 되어 예상치 못한 동작이나 시스템의 충돌을 일으킬 수 있습니다.

따라서 es를 명시적으로 0xB800으로 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 0x09를 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 ; 에러라면 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.

```
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.

왜 Boot_read.asm에서는 플로피 디스크 모터를 안켰는데 Boot_protect.asm에서는 끈건가?

Boot_read.asm 코드는 메모리로 데이터를 로드하는 단계에 초점을 맞추고 있습니다. 이 코드는 "h"와 "i" 문자를 화면에 표시하고, 디스크에서 데이터를 로드하는 기본적인 작업을 수행합니다. 이 단계에서는 아직 플로피 디스크 모터를 끌 필요가 없습니다. 일반적으로 부트로더의 초기 단계에서는 필요한 모든 데이터를 빠르게 읽어들이고 바로 다음 단계로 넘어가야 하므로 모터를 즉시 끄지 않습니다.

반면에, Boot_protect.asm 코드는 보호 모드로의 전환을 준비하는 고급 단계입니다. 여기서는 GDT(Global Descriptor Table)를 설정하고 CPU를 리얼 모드에서 보호 모드로 전환하는 등 더 복잡한 작업을 수행합니다. 보호 모드로 전환하는 과정은 시스템의 큰 변화를 수반하므로, 이 단계에서는 더 이상 플로피 디스크의 사용이 필요하지 않다고 판단되면 모터를 끄는 것이 일반적입니다.

```
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.

GDTR이 CPU 내에 있는 레지스터니까 lgdt를 이용하여 GDT의 시작주소와 한계를 GDTR(CPU)로 보내줌

[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.

(GDT의 한계 값은 0부터 시작하여 셀 때의 마지막 바이트를 가리킴)

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. CR0에는 시스템의 동작 방식을 제어하는 여러 플래그들이 있는데, 그 중에서도 가장 낮은 비트(0번 비트)는 PE(Protection Enable) 플래그입니다.

이 코드는 리얼모드에서 보호모드로 전환하는 과정이고 아직 리얼모드이기에 cr0은 0입니다.

CR0 레지스터란?

CR0는 x86 아키텍처에서 사용하는 32비트 제어 레지스터(Control Register 0)입니다. 이 레지스터는 운영 체제가 시스템을 제어하는 데 중요한 여러 가지 기능을 활성화하거나 비활성화하는 데 사용됩니다.

CR0 레지스터의 비트 중 보호모드 관련 기능은 다음과 같습니다:

PE (Protection Enable) 비트 (0번 비트):

이 비트를 설정하면 시스템이 보호 모드로 전환됩니다. 리얼 모드에서는 이 비트가 0으로 설정되어 있으며, 보호 모드로 진입하기 위해 이 비트를 1로 설정해야 합니다.

EAX 레지스터란?

EAX 레지스터는 x86 아키텍처의 범용 레지스터 중 하나입니다. 원래 16비트 x86 아키텍처에서는 AX 레지스터로 사용되었으며, 32비트 확장인 IA-32 아키텍처에서 EAX로 확장되었습니다. 여기서 'E'는 'Extended'를 의미합니다. 이 레지스터는 여러 용도로 사용될 수 있지만, 특히 다음과 같은 상황에서 중요한 역할을 합니다:

데이터 연산: 산술 연산과 논리 연산을 수행할 때 EAX는 종종 연산의 주 대상 레지스터로 사용됩니다.

함수의 반환 값: 많은 호출 규약에서 함수의 반환 값은 EAX 레지스터에 저장됩니다.

시스템 호출: 시스템 호출을 수행할 때 EAX는 일반적으로 요청된 시스템 호출 번호를 저장하는 데 사용됩니다.

반복문 제어: 일부 반복문 명령어는 EAX 레지스터에 저장된 값을 사용하여 반복 횟수를 제어합니다.

64 비트 확장인 x86-64 아키텍처에서는 EAX 레지스터가 RAX로 더 확장되어, 64 비트 값을 처리할 수 있게 되었습니다. 여기서 'R'은 'Register'를 의미합니다.

레지스터는 CPU 내부의 매우 빠른 메모리이며, 프로그램이 수행되는 동안 데이터를 저장하거나 연산에 사용하기 위해 사용됩니다. EAX 레지스터는 이러한 범용 레지스터들 중 가장 자주 사용되는 레지스터 중 하나입니다.

```
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 \$+2는 현재 명령어의 주소(\$)에서 2바이트 뒤로 점프하라는 명령어입니다. \$는 현재 명령어의 주소를 나타내며, +2는 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 명령어는 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, bx`는 데이터 세그먼트 레지스터 `ds`를 `bx`에 저장된 값으로 설정합니다. 데이터 세그먼트는 일반적으로 데이터를 저장하는 데 사용됩니다.

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

`mov ss, bx`는 스택 세그먼트 레지스터 `ss`를 설정합니다. 스택 세그먼트는 함수 호출, 지역 변수 저장 등 스택에 관련된 데이터를 저장하는 데 사용됩니다.

bx 레지스터란?

`BX` 레지스터는 `x86` 아키텍처에서 사용되는 16 비트 범용 레지스터 중 하나입니다. 초기 `8086` 과 `8088` 프로세서에서 도입되었으며, 이후의 확장된 아키텍처인 `IA-32`(32 비트)와 `x86-64`(64 비트)에서도 사용됩니다.

범용 레지스터는 다양한 용도로 사용될 수 있으며, `BX` 레지스터는 특히 메모리 주소를 지정하는 데 자주 사용됩니다.

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

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

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

`gdt:`은 GDTR에 로드할 데이터를 정의하는 레이블입니다.

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

`dd gdt+0x7C00`은 GDT의 베이스 주소를 정의합니다. `dd`는 Define Doubleword로, 32 비트의 크기를 가진 데이터를 정의합니다. 여기서 `gdt`는 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.

왜 GDT의 첫 번째 항목이 항상 비어있어야 하는가?

GDT(Global Descriptor Table)의 첫 번째 항목이 항상 비어 있어야 하는 이유는 x86 아키텍처의 설계 규약입니다. 이 규약에 따르면, GDT의 첫 번째 엔트리는 NULL 세그먼트 디스크립터로 설정되어야 합니다. NULL 세그먼트 디스크립터는 실제로 어떠한 메모리 세그먼트도 가리키지 않습니다.

NULL 세그먼트 디스크립터가 존재하는 이유는 몇 가지가 있습니다:

오류 감지: 프로그램이나 운영 체제가 실수로 0 번 세그먼트 선택터를 사용하여 메모리 접근을 시도할 경우, NULL 디스크립터는 유효한 메모리 세그먼트를 가리키지 않기 때문에, 이는 즉시 오류(General Protection Fault)를 발생시켜 버그를 감지하고 시스템을 보호하는 데 도움이 됩니다.

프로그램의 적법성 검증: 어떤 프로그램이 0 번 세그먼트 선택터를 사용한다면, 이는 프로그램 내에 버그가 있을 가능성이 높습니다. NULL 디스크립터는 이런 경우를 쉽게 식별하도록 도와줍니다.

시스템의 초기화: 시스템이 부팅될 때 CPU의 세그먼트 레지스터들은 0 으로 초기화됩니다. 이때 NULL 디스크립터가 존재하지 않으면, 시스템이 초기화되자마자 잘못된 메모리 접근을 시도할 위험이 있습니다.

따라서, GDT의 첫 번째 항목을 NULL로 설정함으로써, 시스템은 세그먼트 레지스터가 실수로 0 으로 설정된 경우에도 안전하게 동작할 수 있습니다. 이는 시스템의 안정성과 신뢰성을 높이는 중요한 설계 결정입니다.

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

왜 코드 세그먼트의 선택자를 0x08 로 설정하는가?

GDT(Global Descriptor Table)에서 각 세그먼트 디스크립터는 8 바이트를 차지합니다. 선택터는 GDT 내에 있는 세그먼트 디스크립터를 가리키는 인덱스로 사용됩니다. 이 인덱스는 GDT의 시작에서부터 해당 세그먼트 디스크립터까지의 오프셋을 기반으로 계산됩니다.

GDT의 첫 번째 엔트리는 항상 NULL 디스크립터로, 어떠한 세그먼트도 가리키지 않아야 합니다. 따라서 첫 번째 세그먼트 디스크립터(이 경우 코드 세그먼트)는 GDT의 두 번째 위치에서 시작합니다.

GDT에서의 위치를 바이트 단위로 계산할 때, 첫 번째 세그먼트 디스크립터는 0x08(8 바이트) 오프셋에 위치합니다. 왜냐하면 NULL 디스크립터가 0x00 부터 0x07 까지의 오프셋을 차지하기 때문입니다. 따라서 코드 세그먼트의 선택터는 0x08 로 설정됩니다.

이렇게 설정함으로써, CPU에게 코드 세그먼트 디스크립터가 GDT 내의 어디에 있는지 정확히 알려줄 수 있습니다. 선택터는 세그먼트 레지스터에 로드될 때 사용되며, CPU가 메모리에 접근할 때 해당 세그먼트의 속성을 확인하는 데 사용됩니다.

요약하면, 코드 세그먼트의 선택자를 0x08 로 설정하는 것은 GDT에서 NULL 디스크립터 다음에 위치한 첫 번째 유효한 세그먼트 디스크립터를 가리키기 위한 계산된 오프셋 값입니다.

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

dd 0x0000FFFF, 0x00CF9A00 라인은 GDT(Global Descriptor Table) 내에 코드 세그먼트 디스크립터를 정의하는 부분입니다. 여기서 dd는 'Define Doubleword'의 약자로, 32 비트(4 바이트)의 값을 정의하는데 사용됩니다. 세그먼트 디스크립터는 총 8 바이트로 구성되며, 이중 앞의 4 바이트와 뒤의 4 바이트에는 세그먼트의 다양한 속성이 지정됩니다.

세그먼트 디스크립터의 각 필드는 다음과 같은 정보를 담고 있습니다:

세그먼트의 한계(Limit): 메모리 세그먼트의 크기를 지정합니다. 0x0000FFFF는 세그먼트가 65535 바이트(즉, 64KB)까지의 주소 범위를 가지고 있음을 의미합니다.

베이스 주소(Base Address): 세그먼트의 시작 주소를 지정합니다. 여기서는 디스크립터가 두 부분으로 나뉘어 있기 때문에, 베이스 주소는 두 부분의 조합으로 지정됩니다.

접근 권한(Access Rights): 세그먼트의 타입, 권한 레벨(Privilege Level, 또는 Ring Level), 그리고 세그먼트의 상태(예: 실행 가능, 읽기 가능 등)를 나타냅니다.

0x00CF9A00 에서 각 비트가 가지는 의미는 다음과 같습니다:

CF는 세그먼트 한계의 상위 바이트를 나타냅니다.

9A는 세그먼트의 접근 권한을 나타내며, 다음과 같이 분해할 수 있습니다:

9 는 이진수로 1001 이며, 세그먼트가 코드 세그먼트임을 나타내고, 실행/읽기 가능하며, conforming 속성이 없고, 액세스되었음을 나타내지 않습니다.

A는 이진수로 1010 이며, DPL(Descriptor Privilege Level)을 나타내는데 여기서는 Ring 2 를 의미합니다.

00 은 세그먼트의 베이스 주소의 상위 바이트들을 나타냅니다.

이러한 설정은 세그먼트가 메모리에서 어떻게 동작하는지를 CPU에게 알려주는 역할을 합니다. 예를 들어, 코드 세그먼트 디스크립터는 프로그램이 실행할 코드가 들어 있는 메모리 영역의 속성을 정의합니다.

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

왜 데이터 세그먼트의 선택자를 0x10 로 설정하는가?

데이터 세그먼트의 선택자를 0x10 으로 설정하는 이유는 GDT(Global Descriptor Table) 내에서 세그먼트 디스크립터의 위치를 기반으로 합니다.

GDT 내의 각 세그먼트 디스크립터는 8 바이트를 차지합니다. 선택자(selector)는 이 테이블 내에서 세그먼트 디스크립터의 시작 주소를 가리키는 인덱스입니다. 선택자는 GDT의 시작부터 해당 세그먼트 디스크립터까지의 바이트 오프셋으로 계산되며, 이는 GDT에 엔트리를 추가할 때의 순서에 의해 결정됩니다.

일반적으로 GDT의 첫 번째 엔트리(0 번째 인덱스)는 NULL 디스크립터로 비워두며, 이는 실제 메모리 세그먼트를 가리키지 않습니다. 이렇게 NULL 디스크립터를 사용하는 것은 보안과 안정성을 위한 규약입니다. NULL 디스크립터를 사용하려고 하면 프로세서에서 오류(General Protection Fault)를 발생시키도록 합니다.

GDT의 두 번째 엔트리(1 번째 인덱스)는 보통 코드 세그먼트를 위해 사용됩니다. 이 세그먼트 디스크립터는

GDT의 시작에서 8 바이트 떨어진 곳에 위치하므로, 해당 세그먼트의 선택자는 0x08 이 됩니다.

다음으로, GDT의 세 번째 엔트리(2 번째 인덱스)는 데이터 세그먼트를 위해 사용됩니다. 이 세그먼트 디스크립터는 GDT의 시작에서 16 바이트(0x08 + 0x08) 떨어진 곳에 위치하므로, 해당 세그먼트의 선택자는 0x10 이 됩니다.

따라서, 데이터 세그먼트의 선택자를 0x10 으로 설정하는 것은 GDT 내에서 세 번째 위치(두 번째 유효한 세그먼트 디스크립터)를 가리키는 것과 일치합니다. 이렇게 선택자를 설정함으로써 CPU는 데이터 세그먼트에 대한 올바른 메모리 접근 권한과 속성을 알 수 있게 됩니다.

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

dd 0x0000FFFF, 0x00CF9200 구문은 Global Descriptor Table(GDT)에 데이터 세그먼트 디스크립터를 정의하는 어셈블리 명령어입니다. 이 명령어는 세그먼트의 한계와 속성을 설정합니다. 각각의 세그먼트 디스크립터는 8 바이트로 구성되며, 여기서 dd는 'Define Doubleword'의 약자로, 4 바이트 값을 정의하는 데 사용됩니다.

이 구문에서 정의되는 값들은 다음과 같습니다:

0x0000FFFF: 세그먼트의 한계(Limit)를 정의합니다. 이 값은 세그먼트의 크기를 나타내며, 여기서 0xFFFF는 16 비트 최대 값으로, 세그먼트의 크기가 64KB임을 의미합니다. 이는 세그먼트가 0 부터 시작해서 0xFFFF(즉, 65535)까지의 오프셋을 가질 수 있음을 나타냅니다.

0x00CF9200: 세그먼트의 베이스 주소(Base Address)와 접근 권한(Access Rights) 및 기타 플래그를 설정하는 부분입니다. 이 4 바이트 값은 다음과 같이 세분화됩니다:

00: 세그먼트 베이스 주소의 상위 바이트입니다.

CF9: 세그먼트의 베이스 주소 중간 바이트와 한계의 상위 바이트입니다.

2: 세그먼트의 접근 권한을 나타내는 바이트입니다. 여기서 2 는 세그먼트가 읽기/쓰기 가능한 데이터 세그먼트임을 나타냅니다. 9 는 이진수로 1001 이며, 이는 세그먼트가 읽기/쓰기 가능함을 뜻하며, 'expand-down' 특성은 없고, 'writeable'이며, 'accessed' 비트는 클리어되어 있음을 의미합니다.

0: 세그먼트 베이스 주소의 하위 바이트입니다.

이렇게 정의된 데이터 세그먼트 디스크립터는 메모리의 데이터 세그먼트에 대한 CPU의 접근 방식을 결정합니다. 이는 운영 체제가 메모리 보호, 권한 검사, 그리고 다른 메모리 관리 기능을 구현하는 데 필수적입니다.

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

왜 비디오 세그먼트의 선택자를 0x18 로 설정하는가?

비디오 세그먼트의 선택자를 0x18 로 설정하는 이유는 GDT(Global Descriptor Table) 내에서 세그먼트 디스크립터의 순서와 위치를 기반으로 합니다. GDT 내의 각 세그먼트 디스크립터는 일반적으로 8 바이트의 크기를 가지며, 각 세그먼트 디스크립터는 GDT 내에서 고유한 오프셋을 가집니다.

여기서 비디오 세그먼트의 선택자가 0x18 이라는 것은, 비디오 세그먼트 디스크립터가 GDT 내에서 세 번째 유효한 엔트리(Null 디스크립터를 제외하고)에 위치한다는 것을 의미합니다. 선택자 값은 GDT 시작점으로부터 세그먼트 디스크립터까지의 바이트 단위 오프셋입니다.

예를 들어, 만약 GDT가 다음과 같이 구성되어 있다면:

0x00: Null 디스크립터 (8 바이트)

0x08: 코드 세그먼트 디스크립터 (8 바이트)

0x10: 데이터 세그먼트 디스크립터 (8 바이트)

0x18: 비디오 세그먼트 디스크립터 (8 바이트)

위의 예에서 Null 디스크립터는 언제나 첫 번째 위치에 있으며, 아무것도 가리키지 않는 엔트리로 남겨둡니다. 코드 세그먼트 디스크립터는 두 번째 위치에 있으므로 선택자는 0x08 이 됩니다. 데이터 세그먼트 디스크립터는 세 번째 위치에 있으므로 선택자는 0x10 이 됩니다. 그리고 비디오 세그먼트 디스크립터가 네 번째 위치에 있으므로 선택자는 0x18 이 됩니다.

비디오 세그먼트는 일반적으로 텍스트 모드 화면이나 그래픽 모드 화면 버퍼에 접근할 때 사용되는 세그먼트입니다. 이 선택자를 사용하여 비디오 버퍼에 대한 세그먼트 오버라이드 명령을 수행할 수 있습니다(예: `mov es, 0x18` 후에 `es` 레지스터를 사용하여 비디오 메모리에 접근).

`dd 0x8000FFFF, 0x0040920B`는 비디오 메모리 세그먼트 디스크립터를 정의합니다. 이는 비디오 메모리에 접근할 때 사용됩니다.

`dd 0x8000FFFF, 0x0040920B` 는 Global Descriptor Table(GDT)에 비디오 메모리 세그먼트 디스크립터를 정의하는 어셈블리 명령어입니다. 이 구문은 비디오 세그먼트의 한계(limit), 베이스 주소(base address), 접근 권한(access rights) 등을 설정합니다.

세그먼트 디스크립터는 메모리 세그먼트의 속성을 정의하며, 보통 8 바이트로 구성됩니다. 여기서 `dd`는 Define Doubleword의 약자로, 32 비트(4 바이트)의 값을 정의하는데 사용됩니다.

0x8000FFFF와 0x0040920B 두 개의 더블워드(doubleword)는 다음과 같은 정보를 담고 있습니다:

0x8000FFFF:

앞의 FFFF는 세그먼트의 한계를 나타냅니다. 이 경우 최대 값(65535 바이트)으로 설정되어 있으며, 세그먼트의 크기가 64KB라는 것을 의미합니다.

8000 은 한계 필드의 상위 바이트를 포함합니다. 여기서 8 은 GDT 세그먼트 디스크립터의 'Granularity' 비트를 설정하여, 세그먼트의 한계가 4KB 블록 단위로 해석되도록 합니다. 즉, 실제 메모리 한계는 $FFFF * 4KB = 4GB$ 입니다.

0x0040920B:

00 은 세그먼트 베이스 주소의 상위 바이트입니다.

40 은 세그먼트 베이스 주소의 중간 바이트입니다.

92 는 세그먼트의 접근 권한을 설정하는 바이트입니다. 여기서 9 는 세그먼트가 읽기 가능임을 나타내며, 2 는 세그먼트가 시스템 세그먼트임을 나타냅니다. P 비트(존재 비트)는 세그먼트가 메모리에 실제로 존재한다는 것을 나타냅니다.

0B는 세그먼트의 베이스 주소의 하위 바이트입니다. 그리고 여기서 B는 세그먼트가 읽기 및 쓰기 가능하다는 것을 나타냅니다.

이렇게 정의된 비디오 세그먼트 디스크립터는 메모리의 특정 부분(일반적으로 B0000h, B8000h, A0000h 등의 비디오 메모리 영역)에 대한 접근을 설정하며, CPU가 비디오 버퍼를 올바르게 해석하고 사용할 수 있도록 합니다. 보통 이러한 설정은 텍스트 또는 그래픽 모드의 비디오 메모리에 접근할 때 사용됩니다.

gdt_end:

gdt_end:는 GDT의 끝을 나타내는 레이블입니다. 이는 gdt에서 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.

ASM 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.

CodeSegment, DataSegment, VideoSegment는 이미 Boot_protect.asm에서 정의되었는데 왜 Sector2_protect.asm에서 또 정의하는가?

CodeSegment, DataSegment, 그리고 VideoSegment 같은 세그먼트 셀렉터 값을 정의하는 것은 그 섹터 내의 코드가 어셈블리 명령을 수행할 때 참조할 세그먼트를 명확히 지정하기 위함입니다.

이러한 값들이 Boot_protect.asm에서 이미 정의되었음에도 불구하고 Sector2.asm에서 다시 정의하는 이유는 다음과 같을 수 있습니다:

독립성: 각 어셈블리 파일은 독립적으로 컴파일되며, 다른 파일에서 정의된 심볼이나 레이블에 의존하지 않도록 설계될 수 있습니다. 이렇게 함으로써, 각 파일은 자체적으로 완전하며, 다른 파일의 내용에 대해 걱정할 필요가 없습니다.

명확성과 유지보수: 파일마다 세그먼트 셀렉터를 명시함으로써, 그 파일을 읽는 사람은 세그먼트 셀렉터 값이 무엇을 의미하는지 즉시 알 수 있습니다. 이는 코드의 가독성과 유지보수성을 향상시킬 수 있습니다.

재사용과 모듈화: 하나의 파일 내에서 정의된 심볼이나 레이블을 다른 파일에서 재사용하는 것은 모듈화의 일환입니다. 만약 이 파일들이 다른 프로젝트에서도 사용될 수 있다면, 필요한 모든 정의를 포함하고 있어야 재사용이 용이해집니다.

링크 과정: 여러 어셈블리 파일들이 하나의 실행 파일로 링크될 때, 링커는 각 파일에 정의된 심볼들을 해석하여 최종 실행 파일 내에서 올바른 참조를 생성합니다. 각 파일이 필요한 모든 심볼을 포함하고 있다면 링크 과정이 더욱 간단해집니다.

따라서, Sector2.asm 파일 내에서 이러한 세그먼트 셀렉터를 다시 정의하는 것은 위와 같은 이유로, 파일이 독립적으로 컴파일되고 링크될 수 있도록 하기 위함입니다.

링크란?

프로그래밍에서 링크(link)는 컴파일 과정에서 생성된 하나 이상의 오브젝트 파일(object file)들을 결합하여 단일 실행 파일(executable) 또는 라이브러리(library)를 생성하는 과정입니다. 이 과정은 링커(linker)라고 하는 특별한 프로그램에 의해 수행됩니다.

링크 과정은 크게 두 가지 유형으로 나뉩니다:

정적 링크(Static Linking):

정적 링크는 모든 필요한 코드와 라이브러리를 최종 실행 파일에 포함시키는 것입니다.

결과적으로 독립적인 실행 파일이 생성되며, 외부 라이브러리에 대한 의존성이 없습니다.

이 실행 파일은 다른 컴퓨터로 옮겨져도 추가적인 라이브러리 설치 없이 실행될 수 있습니다.

동적 링크(Dynamic Linking):

동적 링크는 실행 파일이나 라이브러리가 실제로 실행될 때 필요한 코드를 찾아 연결하는 방식입니다.

이 방식에서 실행 파일은 동적 라이브러리(.dll, .so 파일 등)에 정의된 코드를 참조합니다.

동적 라이브러리는 디스크 공간과 메모리를 절약할 수 있으며, 라이브러리를 업데이트하면 해당 라이브러리를

사용하는 모든 프로그램이 혜택을 받을 수 있습니다.

링크 과정 중에 링커는 다음과 같은 작업을 수행합니다:

심볼 해석(Symbol Resolution): 프로그램에서 정의하거나 참조하는 모든 심볼(함수, 변수 등)의 주소를 찾아서 결정합니다.

주소 할당(Address Assignment): 각 심볼에 메모리 주소를 할당하고, 참조되는 위치를 이 주소로 업데이트합니다.

재배치(Relocation): 오브젝트 파일 내의 각 코드와 데이터 섹션을 실행 파일의 주소 공간 내로 재배치합니다.

라이브러리 링킹(Library Linking): 외부 라이브러리 함수를 참조하는 경우, 해당 라이브러리와 연결합니다.

링커는 컴파일러와 함께 소프트웨어 개발 툴체인의 중요한 부분을 이루며, 최종적인 실행 가능한 소프트웨어 제품을 만들어내는 데 필수적인 역할을 수행합니다.

```
[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
```

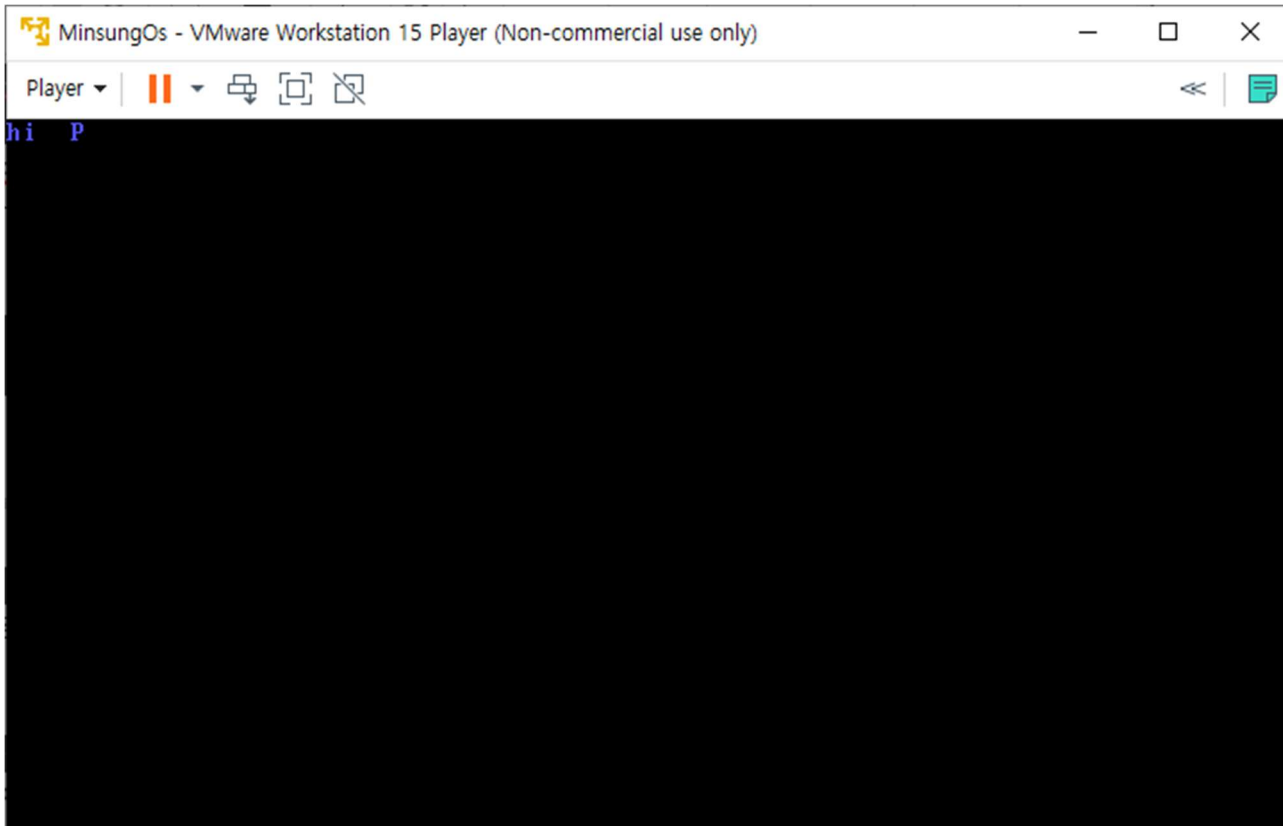
이 줄은 부트 섹터를 512 바이트로 채우기 위해 남은 공간을 0 으로 채우는 명령입니다. \$는 현재 주소를 의미하고, \$\$는 섹션의 시작 주소를 의미합니다. 따라서 (\$-\$\$)는 현재까지의 코드 길이를 나타내며, 512-(\$-\$\$)는 필요한 0의 수를 계산합니다.

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.