주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.01.15 – 2024.01.19

세부 사항

1. 업무 내역 요약 정리

Plan	To-do
1. 부트로더 개발 - 정의 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 2. 하드디스트 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 리얼모드, 보호모드에 대해서 복기 후 함수 만들기, 인터럽트 핸들러 개발에 대해 공부해보았습니다.** 1. 1/12 리뷰 복기 2. 모드 전환 모듈 개발 - 리얼모드와 보호모드에 대한 기본 개념 - 리얼모드 환경에서의 세그먼트:오프셋 구조 - 리얼모드에서 보호모드로의 전환 3. 함수 만들기 - 어셈블리어로 함수 만들기 - C언어로 함수 만들기 - Makefile 만들기 - C언어로 함수 만들기 2 4. 인터럽트 핸들러 개발 - PIC 세팅 - IDT 선언

8. 쉘 개발

- 쉘과 cli의 차이점

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

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

쉘은 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. 1/12 리뷰 복기

1. 방에서 내가 수학 문제를 푼다
컴퓨터 구조에서 가장 중요한 것: cpu, mem, disk
cpu: 나
mem: 책상
disk: 책장

2. CPU(연산장치)는 1 쿨럭당 1 연산(1 cycle)을 한다.
쿨럭이 높을 수록 연산을 많이 할 수 있어서 성능이 좋다.
1 쿨럭당 CPU(연산장치)가 처리 할 수 있는 건 적음 -> 32bit, 64bit

용량이 크면 처리하기 위해서 여러 사이클을 돈다.

3. 리얼모드에서 보호모드 전환 다시 공부
리얼모드는 왜 16 비트인가에서 레지스터와 데이터 버스가 뭔가?

프로젝트 종료시 내가 알아야 할 가장 중요한 것: assembly로 코드를 짜서 nasm을 이용하여 이미지파일로 컴파일하여 VMware에 넣으면 왜 실행이 되는가? (메모리 구조 등을 이용하여 실행되는 과정을 다 알아야 함.)

2. 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? 왜 최대 1MB의 메모리에 접근할 수 있나?
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.

레지스터란?
CPU 내부에 위치한 매우 빠른 메모리 유형으로, CPU가 데이터를 빠르게 읽고 쓸 수 있도록 함.

데이터 버스란?
컴퓨터 시스템의 다른 부분 사이에서 데이터를 전송하는 데 사용되는 통신 시스템.
이는 CPU, 메모리, 그리고 I/O 장치(입출력 장치) 등 시스템의 주요 부분을 연결함.

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?
안됩니다, 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 1970 년대 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 덜 했습니다.
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.

태스크란?

"태스크(Task)"는 컴퓨터에서 실행되는 프로그램이나 작업을 말합니다. 운영 체제에서는 각각의 태스크를 관리하여 CPU 시간을 할당하고, 필요한 자원을 제공하는 등의 작업을 수행합니다. 멀티태스킹 환경에서는 여러 태스크가 동시에 또는 거의 동시에 실행될 수 있으며, 이를 위해 운영 체제가 CPU 시간을 여러 태스크에 공평하게 분배합니다.

왜 리얼모드는 멀티테스킹, 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.

세그멘테이션이란?

- 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 프로세서(CPU)가 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.

프로세스란?

컴퓨터에서 실행중인 프로그램. 프로세스는 서로 완벽히 독립적인 공간을 가짐.

스레드란?

프로세스 내에서 실제로 작업을 수행하는 실행 단위. 스택은 따로따로이지만, 코드 영역과 데이터 영역은 하나를 공유함.

Ex) 프로세스: 각각의 은행 지점
    스레드: 은행 지점 하나에 속한 고객 창구 여러 개

CPU 시간이란?

프로세스 또는 스레드가 CPU를 사용하는 시간.
운영 체제는 스케줄링 알고리즘을 사용하여 프로세스들에게 CPU 시간을 할당 -> 이는 각 작업이 실행될 순서와 실행될 시간을 결정

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 지원하하게 됩니다.

64 비트 확장 모드(롱 모드):
64 비트 x86 프로세서에서 도입된 롱 모드는 64 비트 연산과 주소 지정을 지원합니다.
롱 모드를 통해 프로세서는 이론상 2^64 바이트의 주소 공간에 접근할 수 있지만, 실제로는 프로세서와 운영 체제, 그리고 시스템의 하드웨어 구성에 의해 제한됩니다.
롱 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(롱 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.

시작 주소는 내가 직접 찾아서 코드에 작성해야되는가?

여기서 말하는 '시작 주소'는 코드를 작성할 때 직접 입력하는 값이 아닙니다. 프로그래머는 일반적으로 세그먼트 선택자와 오프셋만을 코드에 명시합니다. '시작 주소'는 세그먼트 선택자에 의해 참조되는 세그먼트 디스크립터 테이블 내에 정의되어 있으며, 이는 운영 체제 또는 시스템이 설정하는 부분입니다.

32 비트 보호 모드에서의 메모리 접근 과정은 다음과 같습니다:

1. 코드 작성: 프로그래머는 코드 내에서 세그먼트 선택자와 오프셋을 사용하여 메모리 주소를 참조합니다. 예를 들어, 0x0008:0050 과 같이 명시할 수 있습니다.
2. 선택자 사용: 프로그램이 실행될 때 CPU는 코드에 명시된 세그먼트 선택자를 사용하여 세그먼트 디스크립터 테이블(GDT 또는 LDT)을 참조합니다.
3. 세그먼트 정보 조회: 선택자는 세그먼트 디스크립터 테이블 내의 특정 항목(인덱스)을 가리키고, CPU는 해당 인덱스에 저장된 세그먼트의 시작 주소(base address), 길이(limit), 접근 권한(access rights) 등의 정보를 읽습니다.
4. 물리 주소 계산: CPU는 세그먼트 시작 주소에 오프셋을 더하여 최종적인 물리 메모리 주소를 계산합니다.

따라서, 프로그래머는 '시작 주소'를 직접 코드에 작성할 필요 없이, 올바른 세그먼트 선택자와 오프셋 값을 사용해서 메모리 주소를 지정하면 됩니다. 나머지 부분은 운영 체제와 CPU가 처리합니다.

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.

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 기준점으로 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.asm에서는 ds를 사용하지 않다가 Boot_protect.asm에서는 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. 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.

```
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 다루는데 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 안껐는데 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.

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

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

dw (Define Word)는 16 비트 값을 정의합니다. 여기서는 GDT의 크기에서 1 을 뺀 값을 나타냅니다.
(GDT의 한계 값은 0 부터 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. CR0 에는 시스템의 동작 방식을 제어하는 여러 플래그들이 있는데, 그 중에서도 가장 낮은 비트(0 번 비트)는 PE(Protection Enable) 플래그입니다.
이 코드는 리얼모드에서 보호모드로 전환하는 과정이고 아직 리얼모드이기에 cr0 은 0 입니다.

CR0 레지스터란?

CR0 는 x86 아키텍처에서 사용하는 32 비트 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 (0 번 비트):
이 비트를 설정하면 시스템이 보호 모드로 전환됩니다. 리얼 모드에서는 이 비트가 0 으로 설정되어 있으며, 보호 모드로 진입하기 위해 이 비트를 1 로 설정해야 합니다.

EAX 레지스터란?

EAX 레지스터는 x86 아키텍처의 범용 레지스터 중 하나입니다. 원래 16 비트 x86 아키텍처에서는 AX 레지스터로 사용되었으며, 32 비트 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 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.

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. 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.

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 셀렉터를 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 셀렉터를 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. 셀렉터는 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 셀렉터는 0x08 로 설정됩니다.

이렇게 설정함으로써, CPU에게 코드 세그먼트 디스크립터가 GDT 내의 어디에 있는지 정확히 알려줄 수 있습니다. 셀렉터는 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, 0x0040920B는 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 dd는 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 = 4GB입니다.

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 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.

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.

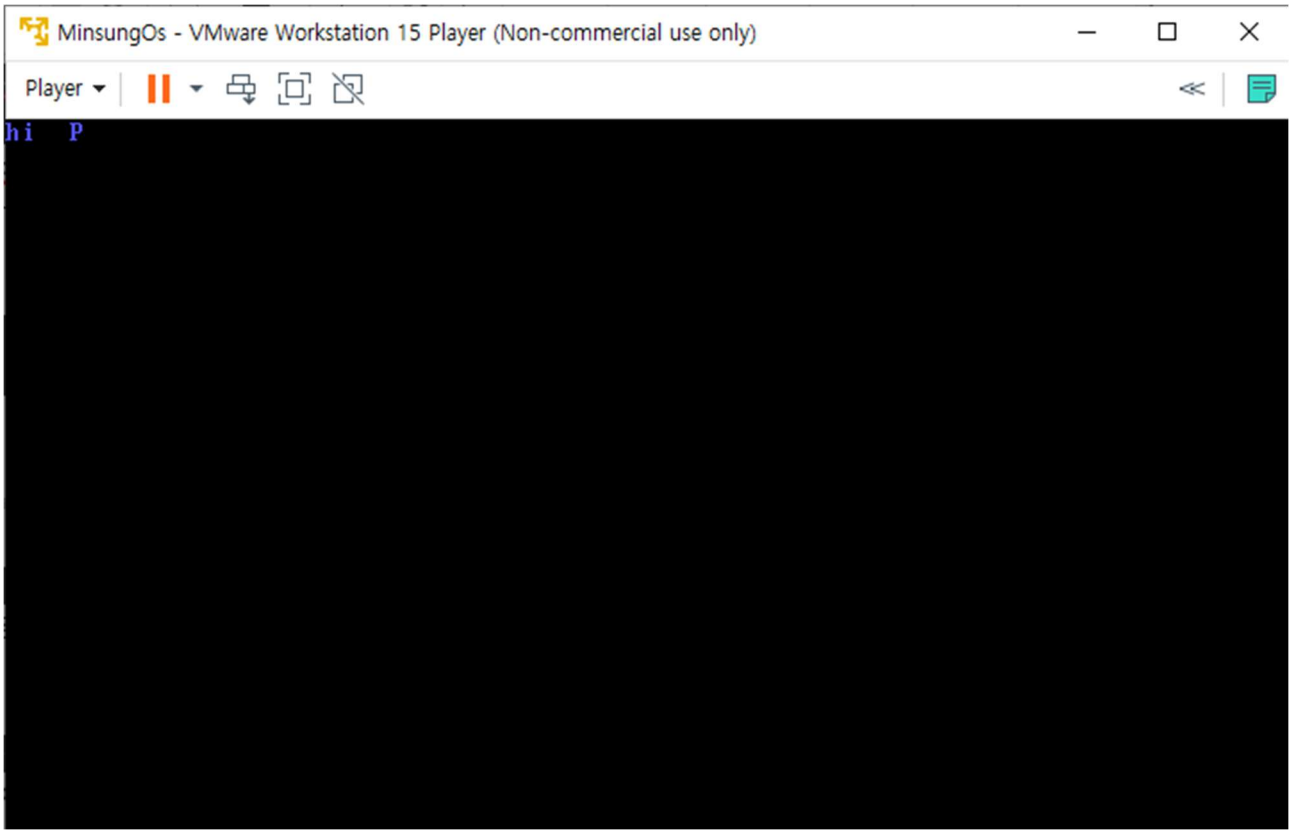
```
times 512-($-$$) db 0
```

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 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.

3. 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.

esp 레지스터가 뭐고 왜 esp 레지스터에 START 레이블의 주소를 로드하는가?

ESP는 Extended Stack Pointer의 약자로, 32 비트 x86 아키텍처에서 사용되는 레지스터입니다. 스택은 프로그램에서 데이터를 저장하고 추출하는 데 사용되는 자료 구조로, ESP는 이 스택의 최상위 위치를 가리키는 역할을 합니다.

스택은 '푸시(push)'와 '팝(pop)' 두 가지 주요 작업을 수행합니다. 푸시 작업은 데이터를 스택에 넣는 것을 말하고, 팝 작업은 스택의 최상위에서 데이터를 가져오는 것을 말합니다. 이러한 작업들은 함수 호출, 지역 변수의 할당 및 제거 등에서 사용됩니다.

따라서 'lea esp, [START]' 명령어는 스택 포인터를 현재 코드의 시작 위치로 설정하여, 이후의 스택 관련 작업들이 이 위치를 기준으로 수행되도록 하는 것입니다.

```
        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.

edi 레지스터란? edi 레지스터를 0 으로 초기화하는 이유는?

EDI 레지스터는 x86 아키텍처의 CPU에서 사용하는 32 비트 레지스터 중 하나입니다. EDI는 'Destination Index'의 약자로, 주로 데이터의 목적지 인덱스를 나타내며, 데이터 복사, 이동 등의 연산에서 사용됩니다.

EDI 레지스터를 0 으로 초기화하는 이유는, 이후에 이루어질 데이터 조작의 목적지를 메모리의 특정 위치, 여기서는 0 번 주소로 설정하기 위함입니다. 이렇게 함으로써, 이후에 데이터를 이동하거나 복사할 때, 해당 데이터는 0 번 주소부터 시작하는 위치에 저장됩니다.

이 코드에서는 'printf' 함수가 문자열을 화면에 출력하는 작업을 수행하는데, 이때 출력될 문자열의 시작 위치를 화면의 시작점, 즉 0 번 주소로 설정하겠다는 의미로 EDI 레지스터를 0 으로 초기화하였습니다.

esi 레지스터란? esi 레지스터에 msg를 로드하는 이유는?

ESI 레지스터는 x86 아키텍처의 CPU에서 사용하는 32 비트 레지스터 중 하나입니다. ESI는 'Source Index'의 약자로, 주로 데이터의 원본 인덱스를 나타내며, 데이터 복사, 이동 등의 연산에서 사용됩니다.

ESI 레지스터에 'msg'를 로드하는 이유는, 이후에 이루어질 데이터 조작의 원본을 'msg'라는 레이블이 가리키는 메모리 위치로 설정하기 위함입니다. 'msg' 레이블은 'Call printf'라는 문자열을 가리키고 있습니다.

이 코드에서는 'printf' 함수가 'msg' 레이블이 가리키는 문자열을 화면에 출력하는 작업을 수행하는데, 이때 출력할 문자열의 시작 위치를 'msg'가 가리키는 위치로 설정하겠다는 의미로 ESI 레지스터에 'msg'를 로드하였습니다. 따라서 'printf' 함수는 ESI 레지스터가 가리키는 위치에서 시작하는 문자열, 즉 'Call printf'를 화면에 출력하게 됩니다.

```
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.

eax, es가뭐고 여기서 왜 사용하는가?

EAX와 ES는 x86 아키텍처의 CPU에서 사용하는 레지스터입니다.

- EAX: 이는 'Extended Accumulator Register'의 약자로, 주로 산술연산이나 데이터 이동 등의 연산에 사용되는 32 비트 레지스터입니다.
- ES: 이는 'Extra Segment'의 약자로, 메모리 세그먼트를 가리키는 데 사용되는 16 비트 세그먼트 레지스터입니다. 이 레지스터는 주로 문자열이나 배열 등의 데이터 블록을 처리할 때 사용됩니다.

'printf' 함수에서 'push eax'와 'push es' 명령은 각각 EAX와 ES 레지스터의 현재 값을 스택에 저장하는 역할을 합니다. 이는 함수의 실행 도중 레지스터의 값이 변경되더라도, 함수가 종료된 후에는 원래의 값으로 복원될 수 있도록 보장하기 위한 것입니다.

즉, 'printf' 함수가 호출되면 레지스터의 값이 변경될 수 있기 때문에, 함수 호출 이전의 레지스터 값을 보존하기 위한 목적으로 'push eax'와 'push es' 명령이 사용됩니다. 이렇게 함으로써, 함수가 종료된 이후에도 레지스터의 원래 값이 유지될 수 있습니다. 이는 함수 호출이 프로그램의 상태를 변경하지 않도록 하는 중요한 방법 중 하나입니다.

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

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

VideoSegment란?

'VideoSegment'는 이 코드에서 사용된 사용자 정의 식별자입니다. 'equ'는 어셈블리 언어에서 'equals'의 약자로, 이 식별자에 특정 값을 할당하는 역할을 합니다. 따라서 'VideoSegment equ 0x18'는 'VideoSegment'라는 이름을 가진 식별자에 0x18 이라는 값을 할당하는 것을 의미합니다.

이 코드에서 'VideoSegment'는 화면(비디오) 메모리의 세그먼트를 가리키는데 사용됩니다. 즉, 'VideoSegment'는 화면 출력을 위한 메모리 영역을 가리키는 주소값을 담고 있습니다.

이 값은 'printf' 함수에서 화면에 문자열을 출력하기 위해 사용됩니다. 'printf' 함수는 ESI 레지스터가 가리키는 문자열을 읽어, 그 문자열을 'VideoSegment'가 가리키는 화면 메모리 영역에 출력하는 작업을 수행합니다. 따라서 'VideoSegment'는 화면 출력을 위한 메모리 영역의 주소를 지정하는 역할을 합니다.

```
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.

db란?

DB는 어셈블리 언어에서 사용하는 지시어로, 'Define Byte'의 약자입니다. 이는 특정 메모리 위치에 1 바이트의 데이터를 정의하거나 저장하는 데 사용됩니다.

즉, 'db' 지시어를 사용하면, 지정된 이름(여기서는 'msg')에 원하는 바이트 값을 할당할 수 있습니다. 이 때 할당되는 값은 숫자일 수도 있고, 문자열일 수도 있습니다.

'Call printf',0'는 'Call printf'라는 문자열과 그 뒤에 이어지는 null 문자(0)를 나타냅니다. 문자열의 끝을 나타내는 null 문자는 문자열 처리에서 중요한 역할을 합니다.

따라서 'msg db 'Call printf',0'는 'msg'라는 이름의 메모리 위치에 'Call printf'라는 문자열을 저장하고, 문자열의 끝에 null 문자를 추가하는 것을 의미합니다. 이렇게 저장된 'msg'는 이후에 'printf' 함수에서 화면에 출력될 문자열로 사용됩니다.

```
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, 보호 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.

```c
        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.

어셈블이란? 링크란?

- 어셈블(Assemble):

어셈블은 고급 프로그래밍 언어로 작성된 소스 코드를 컴퓨터가 이해할 수 있는 저급 언어인 어셈블리 언어로 변환하는 과정을 말합니다. 이 과정은 일반적으로 컴파일러에 의해 수행되며, 결과적으로 어셈블리 코드가 생성됩니다. 이후, 이 어셈블리 코드는 다시 기계어 코드로 변환되어 실행 파일이 만들어집니다.

- 링크(Link):

링크는 여러 개의 오브젝트 파일(.o 파일)이나 라이브러리를 하나의 실행 파일로 합치는 과정을 말합니다. 이 과정에서 링커(linker)라는 도구가 사용되며, 링커는 여러 오브젝트 파일들 사이에서 정의된 함수나 변수 등의 참조를 해결하고, 모든 코드와 데이터를 하나의 실행 파일로 합칩니다.

'-c' 옵션을 사용하면, 소스 코드는 컴파일과 어셈블 과정을 거쳐 오브젝트 파일이 생성되지만, 이 오브젝트 파일은 링크 과정을 거치지 않습니다. 따라서 이 오브젝트 파일은 독립적으로 실행할 수 없으며, 실행 가능한 프로그램을 만들기 위해서는 다른 오브젝트 파일들과 링크 과정을 거쳐야 합니다.

-m32: 이 옵션은 컴파일러에게 32 비트 코드를 생성하도록 지시합니다. 기본적으로, 운영체제의 아키텍처에 따라

컴파일러는 32 비트 또는 64 비트 코드를 생성합니다. 그러나 -m32 옵션을 사용하면, 컴파일러는 32 비트 코드를 생성하도록 강제할 수 있습니다. 이 옵션은 주로 64 비트 시스템에서 32 비트 응용 프로그램을 컴파일할 때 사용됩니다.

-ffreestanding: 일반적으로 C 프로그램을 작성하고 컴파일할 때는 표준 라이브러리를 사용합니다. 표준 라이브러리에는 print 함수나 math 함수 등 우리가 프로그래밍할 때 자주 사용하는 기능들이 포함되어 있습니다. 또한 일반적인 C 프로그램은 'main'이라는 함수에서 시작합니다.

그런데 'freestanding' 환경이라는 것은 이러한 표준 라이브러리를 사용하지 않는 환경을 말합니다. 이 환경에서는 표준 라이브러리에 있는 기능들을 사용할 수 없기 때문에, 프로그램을 작성할 때 필요한 모든 기능을 직접 구현해야 합니다.

또한 'freestanding' 환경에서는 프로그램의 시작점이 반드시 'main'이 아니어도 됩니다. 즉, 프로그램이 시작되는 위치를 우리가 원하는 곳으로 지정할 수 있습니다.

'-ffreestanding' 옵션은 이러한 'freestanding' 환경에서 프로그램을 컴파일하겠다는 것을 컴파일러에게 알리는 역할을 합니다. 이는 주로 운영 체제의 핵심 부분인 커널이나, 임베디드 시스템(내장형 시스템) 등 특수한 경우에 사용됩니다. 왜냐하면 이러한 시스템들은 표준 라이브러리를 사용하지 않고, 프로그램의 시작점도 'main'이 아닌 경우가 많기 때문입니다.

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.

- Makefile 만들기

이제 C언어로 운영체제를 개발할 수 있게 되었습니다.
하지만 소스 몇 줄만 수정해도 복잡한 명령어를 일일이 입력해서 다시 이미지(img) 파일을 얻어야만 합니다.
이를 해결하기 위해 make 명령어를 이용합니다. 그러기 위해서 Makefile을 아래와 같이 작성한 후, 소스 파일들(Boot.asm, Sector2.asm, main.c)가 있는 폴더에 넣어둡니다.

make 명령어란? Makefile 이란?

make 명령어는 소스 코드 파일을 컴파일하고 애플리케이션을 빌드하는 과정을 자동화하기 위해 사용되는 유틸리티입니다. 이 명령어는 주로 Unix와 Unix 계열 시스템에서 사용되며, Makefile이라는 이름의 설정 파일에 기반하여 작동합니다.

Makefile 내부에는 다음과 같은 요소들이 포함됩니다:

목표(Targets): 빌드 과정의 결과물을 나타내며, 보통 실행 파일이나 오브젝트 파일 등이 됩니다.
의존성(Dependencies): 각 목표를 생성하기 위해 필요한 파일이나 다른 목표들입니다. 의존성이 변경되었을 때만 해당 목표의 빌드가 다시 실행됩니다.
명령(Commands): 목표를 만들기 위해 실행되어야 하는 실제 커맨드 라인 명령어들입니다. 이 명령어들은 의존성에 변화가 있을 때 실행됩니다.

make 명령어 사용의 기본적인 형태는 아래와 같습니다:

```bash
make [옵션] [목표]
```

make: 이 명령어를 단독으로 사용하면, Makefile에서 첫 번째 목표를 빌드합니다.
make 목표: 특정 목표를 지정하여 해당 목표에 대한 빌드 과정만을 실행합니다.
make 옵션: 다양한 옵션을 사용하여 make의 동작을 조정할 수 있습니다. 예를 들어, -f 옵션을 통해 특정 이름의 Makefile을 지정할 수 있습니다.

make 명령어는 개발 과정에서 매우 유용하며, 소스 코드의 변경이 있을 때마다 전체 프로젝트를 처음부터 다시 컴파일하지 않고도 변경된 부분만을 재컴파일함으로써 시간을 절약해 줍니다.

Makefile

```
CC = gcc

final.img : Boot.img Sector2.img disk.img
    cat Boot.img Sector2.img disk.img > final.img

disk.img : main.img
    objcopy -O binary main.img disk.img

main.img :main.o
    ld -melf_i386 -Ttext 0x10200 -nostdlib main.o -o main.img

main.o : main.c
    gcc -c -m32 -ffreestanding main.c -o main.o

Boot.img : Boot.asm
    nasm -f bin -o Boot.img Boot.asm

Sector2.img : Sector2.asm
    nasm -f bin -o Sector2.img Sector2.asm

clean :
    rm *.o
```

Makefile은 아래에서부터 읽어야 합니다. 각각의 문장은 콜론(:)으로 구분되며, 왼쪽에는 생성하려는 파일을 명시하고, 오른쪽에는 해당 파일을 생성하기 위해 필요한 의존 파일을 작성합니다. 또한, 실행하고자 하는

명령어는 아래줄에 작성해야 합니다. 순서를 잘 지켜서 Makefile을 작성해야 합니다.

예를 들어, "nasm -f bin -o Sector2.img Sector2.asm" (A 명령어)를 통해 sector2.img 파일을 생성한 후에 "objcopy -O binary main.img disk.img" (B 명령어)를 실행할 수 있습니다. 따라서, A 명령어는 B 명령어보다 아래에 작성되어야 합니다. 이러한 순서를 지켜서 Makefile을 작성해야 합니다.

```
CC = gcc
```

CC는 컴파일러를 위한 변수입니다. 여기서 gcc (GNU Compiler Collection)를 기본 C 컴파일러로 사용하겠다는 것을 정의합니다.

```
final.img : Boot.img Sector2.img disk.img
	cat Boot.img Sector2.img disk.img > final.img
```

final.img는 최종적으로 만들어질 이미지 파일입니다. 이 파일은 Boot.img, Sector2.img, disk.img 세 파일을 cat 명령어를 사용해 연결(concatenate)하여 생성됩니다.

```
disk.img : main.img
	objcopy -O binary main.img disk.img
```

disk.img 파일은 main.img에 의존합니다. objcopy 명령어를 사용하여 main.img에서 disk.img로 포맷을 변환합니다. -O binary 옵션은 출력 파일을 이진 형식으로 만듭니다.

```
main.img : main.o
	ld -melf_i386 -Ttext 0x10200 -nostdlib main.o -o main.img
```

main.img는 main.o 오브젝트 파일로부터 링크되어 생성됩니다. ld는 링커로, -melf_i386 옵션은 ELF(i386 아키텍처) 형식의 실행 파일을 생성합니다. -Ttext 0x10200 는 실행 파일의 텍스트 세그먼트 시작 주소를 0x10200 으로 설정합니다. -nostdlib는 표준 라이브러리를 링크하지 않도록 합니다.

```
main.o : main.c
	$(CC) -c -m32 -ffreestanding main.c -o main.o
```

main.o는 main.c 소스 파일로부터 컴파일됩니다. 여기서 $(CC)는 위에서 정의한 gcc를 사용합니다. -c 옵션은 컴파일만 수행하고 링크는 하지 않습니다. -m32 는 32 비트 코드를 생성하며, -ffreestanding은 표준 라이브러리가 없는 환경에서 컴파일됨을 의미합니다.

```
Boot.img : Boot.asm
	nasm -f bin -o Boot.img Boot.asm
```

Boot.img는 Boot.asm 어셈블리 파일로부터 NASM 어셈블러를 사용하여 생성됩니다. -f bin 옵션은 이진 형식의 출력 파일을 생성합니다.

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

Sector2.img도 마찬가지로 Sector2.asm 어셈블리 파일로부터 이진 형식의 이미지 파일을 생성합니다.

```
clean :
	rm *.o
```

clean은 특별한 목표로, make clean 명령을 실행하면 오브젝트 파일(*.o)을 삭제하여 빌드 과정에서 생성된 중간 파일들을 정리합니다.

이 스크립트는 전체 운영체제의 이미지를 만드는 복잡한 과정을 단순화하여, 소스 코드나 어셈블리 파일이 변경될 때마다 필요한 부분만 재빌드하도록 합니다. 이로써 개발자는 빌드 프로세스에 드는 시간과 노력을 크게 줄일 수 있습니다.

Makefile은 확장자가 없나?

Makefile은 확장자가 없는 것이 일반적인 관례입니다. 이는 make 유틸리티가 기본적으로 'Makefile'이라는 이름의 파일을 찾기 때문입니다. make 명령어를 실행할 때, 별도의 옵션 없이 해당 디렉토리에 'Makefile' 또는 'makefile'이라는 이름의 파일이 있으면, make는 그 파일을 빌드 지침으로 사용합니다.

확장자가 없는 Makefile의 관습은 여러 이유로 인해 형성되었습니다:

간결함: 확장자가 없으므로 파일 이름이 간결해져서 사용자가 쉽게 인식하고 기억할 수 있습니다.
유닉스 전통: 유닉스 시스템에서는 확장자가 파일의 종류를 지정하는 주된 방법이 아니었으며, 파일의 내용과 권한이 파일의 행동을 결정했습니다. Makefile은 이러한 전통을 따릅니다.
표준화: 모든 유닉스 기반 시스템에서 make를 사용할 때 일관된 파일 이름을 갖는 것이 중요했고, 'Makefile'이라는 이름은 이러한 표준을 제공합니다.
명확성: 확장자가 없는 Makefile은 빌드 과정에 사용되는 특별한 파일임을 명확하게 하며, 다른 소스 파일이나 스크립트 파일과 구분되게 합니다.

확장자가 필요한 특별한 상황이 아니라면, 일반적으로 Makefile은 확장자 없이 사용됩니다. 만약 다른 이름이나 확장자를 가진 Makefile을 사용하고 싶다면, make 명령어와 함께 -f 또는 --file 옵션을 사용하여 해당 파일을 지정할 수 있습니다. 예를 들어, make -f custom.mk와 같은 방식으로 사용할 수 있습니다.

터미널을 열고 파일이 있는 곳으로 이동하여 "make"라는 명령어를 칩니다.

```
minsung@ubuntu:~/Dev/Os$ ls -ltr
total 16
-rw-rw-r-- 1 minsung minsung 199 Jan  3 22:49 HelloWorld.c
-rw-rw-r-- 1 minsung minsung 803 Jan  4 00:02 Boot_c.asm
-rw-rw-r-- 1 minsung minsung 580 Jan  4 00:34 Sector2_c.asm
-rwxrw-r-- 1 minsung minsung 539 Jan 16 18:40 Makefile
minsung@ubuntu:~/Dev/Os$ make
nasm -f bin -o Boot_c.img Boot_c.asm
nasm -f bin -o Sector2_c.img Sector2_c.asm
gcc -c -m32 -ffreestanding HelloWorld.c -o HelloWorld.o
ld -melf_i386 -Ttext 0x10200 -nostdlib HelloWorld.o -o HelloWorld.img
ld: warning: cannot find entry symbol _start; defaulting to 0000000000010200
objcopy -O binary HelloWorld.img disk.img
cat Boot_c.img Sector2_c.img disk.img > final.img
```

입력하고자 했던 명령어들이 순식간에 다 실행되었습니다. 이제 파일이 추가된다면 Makefile만 수정하면 됩니다.

- C언어로 함수 만들기 2

명령어를 여러 번 칠 일도 줄여줄 Makefile도 만들었으니 C언어를 이용해서 printf와 비슷한 기능을 하는 kprintf를 만들어 봅시다.

먼저 function.h를 만들고 다음을 선언합니다.

```
#pragma once

void kprintf(char*, int, int);

// str: 출력할 스트링 주소, int: 몇 번째 줄에 출력할 것인지, int: 몇 번째 행에 출력할 것인지
```

```
#pragma once
```

이는 헤더 파일이 프로그램에 한 번만 포함되도록 보장하는 전처리 지시어입니다. 중복 포함을 방지하여 컴파일 에러와 불필요한 컴파일 시간 증가를 예방하는 역할을 합니다.

```
void kprintf(char*, int, int);
```

이 선언은 kprintf라는 함수의 원형(prototype)을 나타냅니다. 함수의 반환 타입, 이름, 매개변수 타입을 정의합니다.

kprintf 함수는 세 개의 매개변수를 받습니다:

첫 번째 char* 타입의 매개변수는 출력하고자 하는 문자열의 메모리 주소를 가리킵니다. 실제 문자열 데이터는 이 포인터에 의해 참조됩니다.
두 번째 int 타입의 매개변수는 문자열을 출력할 줄 번호를 지정합니다. 이는 화면 또는 출력 매체에서 문자열이 나타날 세로 위치를 결정합니다.
세 번째 int 타입의 매개변수는 문자열을 출력할 행 번호를 지정합니다. 이는 화면 또는 출력 매체에서 문자열이 나타날 가로 위치를 결정합니다.

이 함수는 printf와 같은 표준 출력 함수의 기능을 사용자 정의 방식으로 구현하려는 목적으로 설계되었습니다. 즉, kprintf는 지정된 줄과 행 위치에 문자열을 출력하는 기능을 수행할 것으로 예상됩니다. 이를 통해 개발자는 특정 출력 위치를 제어할 수 있는 추가적인 유연성을 가질 수 있습니다. 실제 kprintf 함수의 동작은 이 헤더 파일에 선언된 대로 구현된 소스 코드에서 정의되어야 합니다.

그 다음, function.c를 만들고 kprintf를 구현합니다.

```
void kprintf(char* str, int line, int col) // str 글자를 line 행 col 열에 출력하는 함수
{
    char *video = (char*)(0xB8000 + 2 * (line * 80 + col));

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

    return;
}
```

그 다음, main.c를 만들어봅니다.

```
#include "function.h"

void main()
{
    kprintf("Use function in C!", 10, 5);
}
```

function.h를 include하고 kprintf를 써봅니다.

이제 소스는 모두 만들었으나 새로운 파일들이 추가되었으니 Makefile을 수정해서 컴파일 환경을 구축해봅니다.

```
CC = gcc

final.img : Boot_c.img Sector2_c.img disk.img
    cat Boot_c.img Sector2_c.img disk.img > final.img

disk.img : main.img
    objcopy -O binary main.img disk.img

main.img : main.o function.o
    ld -melf_i386 -Ttext 0x10200 -nostdlib main.o function.o -o main.img

main.o : main.c
    gcc -c -m32 -ffreestanding main.c -o main.o

function.o : function.c
    gcc -c -m32 -ffreestanding function.c -o function.o

Boot_c.img : Boot_c.asm
    nasm -f bin -o Boot_c.img Boot_c.asm

Sector2_c.img : Sector2_c.asm
    nasm -f bin -o Sector2_c.img Sector2_c.asm

clean :
    rm *.o
```

```
CC = gcc
```

CC 변수에 gcc 컴파일러를 할당합니다. 이 변수는 나중에 C 파일을 컴파일할 때 사용됩니다.

```
final.img : Boot_c.img Sector2_c.img disk.img
cat Boot_c.img Sector2_c.img disk.img > final.img
```

final.img 타깃은 Boot_c.img, Sector2_c.img, disk.img 파일들 의존합니다. 이 파일들을 순서대로 결합하여 final.img를 생성합니다. cat 명령은 여러 파일의 내용을 연결하여 표준 출력으로 보내는 데 사용되며, 여기서는 리다이렉션(>)을 사용하여 결과를 final.img 파일로 저장합니다.

```
disk.img : main.img
     objcopy -O binary main.img disk.img
```

disk.img는 main.img에 의존합니다. objcopy 명령은 main.img를 이진(binary) 형식으로 변환하여 disk.img를 생성합니다.

```
main.img : main.o function.o
    ld -melf_i386 -Ttext 0x10200 -nostdlib main.o function.o -o main.img
```

main.img 타깃은 main.o와 function.o 오브젝트 파일에 의존합니다. ld는 링커로, 여기서는 main.o와 function.o를 링크하여 main.img를 생성합니다. -melf_i386 는 32 비트 ELF 형식을 지정하고, -Ttext 0x10200 는 실행 코드가 로드될 메모리 주소를 설정합니다. -nostdlib 옵션은 표준 라이브러리를 링크하지 않도록 합니다.

```
main.o : main.c
    $(CC) -c -m32 -ffreestanding main.c -o main.o
```

main.o는 main.c 소스 파일에 의존합니다. $(CC)는 앞서 정의한 gcc 컴파일러를 사용하며, -c는 컴파일만 수행하고 링크는 하지 않습니다. -m32 는 32 비트 아키텍처로 컴파일하고, -ffreestanding은 운영체제에 의존하지 않는 환경을 가정하여 컴파일합니다.

```
function.o : function.c
    $(CC) -c -m32 -ffreestanding function.c -o function.o
```

function.o는 function.c에 의존합니다. 이 파일도 main.c와 같은 방식으로 컴파일됩니다.

```
Boot_c.img : Boot_c.asm
    nasm -f bin -o Boot_c.img Boot_c.asm
```

Boot_c.img는 Boot_c.asm 어셈블리 파일에 의존하며, nasm 어셈블러를 이용해 바이너리 형식으로 컴파일합니다.

```
Sector2_c.img : Sector2_c.asm
    nasm -f bin -o Sector2_c.img Sector2_c.asm
```

Sector2_c.img도 Sector2_c.asm에 의존하여 같은 방식으로 컴파일됩니다.

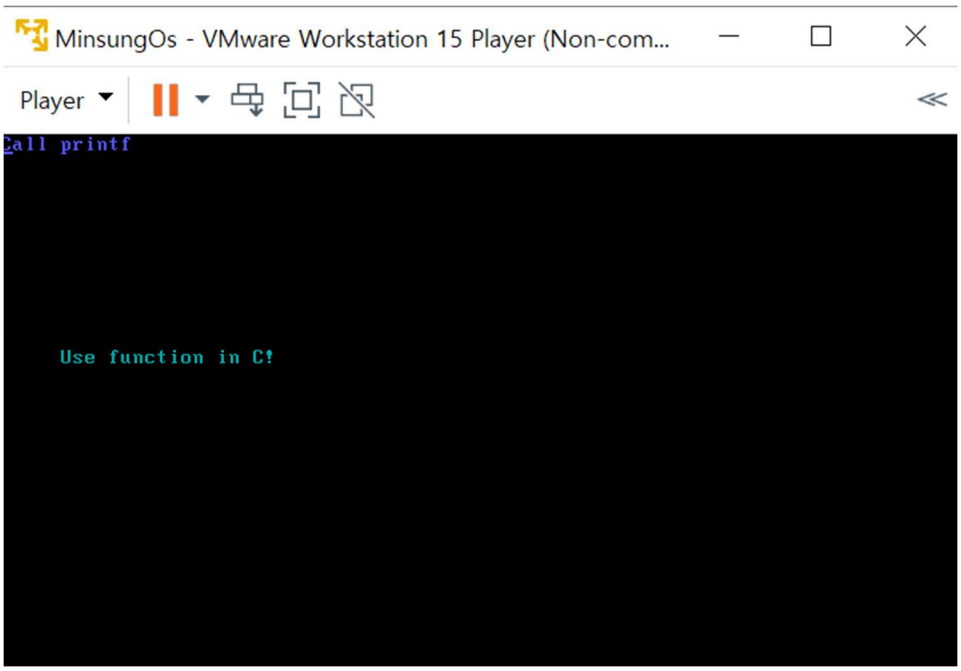
```
clean :
    rm *.o
```

clean 타깃은 실제 파일을 생성하지 않고, make clean 명령을 실행할 때 사용됩니다. 이 명령은 모든 오브젝트 파일(*.o)을 삭제하여 빌드 과정에서 생성된 중간 파일들을 정리합니다.

우분투 환경에서 make 명령어를 입력해 컴파일을 합니다.

```
minsung@ubuntu:~/Dev/Os/함수 만들기/C언어로 함수 만들기 2$ ls -ltr
total 24
-rwxrw-r-- 1 minsung minsung 803 Jan  4 00:02 Boot_c.asm
-rwxrw-r-- 1 minsung minsung 580 Jan  4 00:34 Sector2_c.asm
-rw-rw-r-- 1 minsung minsung 169 Jan 16 20:02 function.h
-rw-rw-r-- 1 minsung minsung  78 Jan 16 20:23 main.c
-rwxrw-r-- 1 minsung minsung 585 Jan 16 20:31 Makefile
-rw-rw-r-- 1 minsung minsung 247 Jan 16 21:06 function.c
minsung@ubuntu:~/Dev/Os/함수 만들기/C언어로 함수 만들기 2$ make
nasm -f bin -o Boot_c.img Boot_c.asm
nasm -f bin -o Sector2_c.img Sector2_c.asm
gcc -c -m32 -ffreestanding main.c -o main.o
gcc -c -m32 -ffreestanding function.c -o function.o
ld -melf_i386 -Ttext 0x10200 -nostdlib main.o function.o -o main.img
ld: warning: cannot find entry symbol _start; defaulting to 0000000000010200
objcopy -O binary main.img disk.img
cat Boot_c.img Sector2_c.img disk.img > final.img
```

이제 생성된 이미지 파일(final.img)을 가상머신에 돌려봅니다.



C언어로 함수를 만들고, 또 분리해서 소스를 작성할 수 있게 되었습니다.

4. 인터럽트 핸들러 개발

- PIC 세팅

PIC란?

PIC(Programmable Interrupt Controller)는 인터럽트를 관리하는 하드웨어 장치입니다. 컴퓨터에서 여러 장치들이 동시에 CPU의 주의를 요구할 수 있는데, PIC는 이러한 인터럽트 요청을 조정하고, CPU가 순차적으로 각 인터럽트를 처리할 수 있도록 도와줍니다.

초기의 IBM PC 호환 컴퓨터들은 주로 두 개의 PIC 칩셋, 즉 주 PIC(Primary PIC 또는 Master PIC)와 부 PIC(Secondary PIC 또는 Slave PIC)를 사용했습니다. 이들은 각각 8 개의 라인(IRQ0-IRQ7 및 IRQ8-IRQ15)을 관리할 수 있었고, 이를 통해 총 16 개의 인터럽트 요청을 처리할 수 있었습니다.

PIC의 기능:

1. 인터럽트 결합: 여러 인터럽트 라인을 하나의 시스템 인터럽트로 결합합니다. 이는 CPU가 단일 인터럽트 라인만을 감시하면서 여러 장치의 인터럽트를 처리할 수 있게 해줍니다.
2. 우선순위 지정: 인터럽트에 우선순위를 부여하여 중요한 작업을 먼저 처리할 수 있게 합니다.
3. 인터럽트 마스킹: 특정 인터럽트 라인을 활성화하거나 비활성화할 수 있습니다. 이를 통해 시스템은 특정 인터럽트를 무시하고 다른 인터럽트에 집중할 수 있습니다.
4. 인터럽트 벡터 리매핑: 인터럽트 벡터 주소를 변경할 수 있습니다. 이는 특히 x86 시스템에서 중요한데, 기본적으로 PIC의 인터럽트 벡터는 CPU의 예약된 영역과 충돌하기 때문에, 일반적으로 이들을 0x20-0x2F (IRQ0-IRQ7)와 0x70-0x7F (IRQ8-IRQ15)로 리매핑합니다.

PIC의 프로그래밍:

PIC의 세팅은 보통 부트 시에 시스템의 다른 부분들과 함께 초기화됩니다. 이 과정에서 인터럽트 벡터 테이블을 설정하고, PIC 칩셋을 초기화하여 인터럽트를 적절히 처리할 수 있도록 합니다.

PIC를 프로그래밍할 때 사용하는 I/O 포트는 다음과 같습니다:

- 주 PIC: 명령 포트 0x20, 데이터 포트 0x21
- 부 PIC: 명령 포트 0xA0, 데이터 포트 0xA1

이 포트들을 사용하여 PIC에 명령을 보내고, PIC의 동작을 설정하며, 인터럽트 라인을 활성화하거나 비활성화할 수 있습니다.

인터럽트 핸들러 개발에서 PIC를 세팅하는 것은 시스템이 인터럽트를 적절하게 처리하고, 올바른 인터럽트 핸들러를 호출할 수 있도록 보장하는 중요한 단계입니다.

Boot.asm

```
[org 0]
[bits 16]

28 references
jmp 0x07C0:start

15 references
start:
206 references
mov ax, cs
206 references
mov ds, ax
206 references
mov es, ax

11 references
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, 20 ; 20섹터를 읽을 것이다 MAX = 64
    mov ch, 0 ; 0번째 실린더 MAX 1024
    mov cl, 2 ; 2번째 섹터부터 읽기 시작한다
    mov dh, 0 ; 헤드는 0 MAX 16
    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

; 인터럽트 코드 시작

mov al, 0x11    ;pic 초기화
out 0x20, al    ;마스터 PIC
dw 0x00eb, 0x00eb   ;jmp $+2, jmp $+2
out 0xA0, al    ;슬레이브 PIC
dw 0x00eb, 0x00eb

mov al, 0x20;마스터 PIC  인터럽트 시작점
out 0x21, al
dw 0x00eb, 0x00eb
mov al, 0x28
out 0xA1, al
dw 0x00eb, 0x00eb
```

```
mov al, 0x04    ;마스터 PIC의 IRQ 2번에
out 0x21, al    ;슬레이브 PIC이 연결되어 있다.
dw 0x00eb, 0x00eb
mov al, 0x02    ;슬레이브 PIC이 마스터 PIC이
out 0xA1, al    ;IRQ 2번에 연결되어 있다.
dw 0x00eb, 0x00eb

mov al, 0x01    ; 8086모드를 사용한다.
out 0x21, al
dw 0x00eb, 0x00eb
out 0xA1, al
dw 0x00eb, 0x00eb

mov al, 0xFF    ;슬레이브 PIC의 모든 인터럽트를
out 0xA1, al    ;막아둔다.
dw 0x00eb, 0x00eb
mov al, 0xFB    ;마스터 PIC의 IRQ 2번을 제외한
out 0x21, al    ;모든 인터럽터를 막아둔다.

; 인터럽트 코드 끝
```

```
lgdt[gdtr]

206 references
mov eax, cr0
7 references
or eax, 1
206 references
mov cr0, eax

28 references
jmp $+2
9 references
nop
9 references
nop

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

```
28 references
jmp dword CodeSegment:0x10000

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

14 references
gdt:
    dd 0,0
    CodeSegment equ 0x08
    dd 0x0000FFFF, 0x00CF9A00
    DataSegment equ 0x10
    dd 0x0000FFFF, 0x00CF9200
    VideoSegment equ 0x18
    dd 0x8000FFFF, 0x0040920B

9 references
gdt_end:

12 references
times 510-($-$$) db 0
21 references
dw 0xAA55
```

```
mov al, 0x11   ; PIC 초기화 명령어 (ICW1)
```

PIC에 초기화 명령을 전송합니다. 여기서 0x11 은 PIC가 초기화 시퀀스를 받아들일 준비가 되었음을 알리는 값입니다.

al 레지스터란?

al은 x86 아키텍처의 CPU에서 사용하는 레지스터 중 하나입니다. al은 ax 레지스터의 하위 8 비트를 나타내며, ax는 16 비트 레지스터입니다.

x86 아키텍처의 일반 레지스터는 다음과 같은 구조를 가지고 있습니다:

- ax : 16 비트 레지스터 (Accumulator register)
  - ah : ax의 상위 8 비트 (high byte)
  - al : ax의 하위 8 비트 (low byte)

ax 레지스터는 주로 산술 연산에 사용되며, ah와 al은 ax를 더 작은 단위로 다룰 때 사용됩니다. 특히 I/O 포트와의 통신에서 8 비트 데이터를 전송하거나 받을 때 al 레지스터가 자주 사용됩니다.

예를 들어, mov al, 0x11 명령은 al 레지스터에 0x11 값을 로드하는 것입니다. 이 값은 이후에 I/O 포트로 전송되어 PIC(Programmable Interrupt Controller)를 초기화하는 데 사용됩니다.

```
out 0x20, al   ; 마스터 PIC(커맨드 레지스터)에 명령어 전송
```

앞서 al 레지스터에 저장한 초기화 명령어를 마스터 PIC의 커맨드 레지스터 포트(0x20)로 전송합니다.

마스터 PIC란?

마스터 PIC(Programmable Interrupt Controller)는 컴퓨터 시스템에서 다양한 장치로부터 발생하는 인터럽트 신호를 관리하는 하드웨어 컴포넌트입니다. 이 컨트롤러는 인터럽트 신호를 CPU로 전달하기 전에 우선 순위를 지정하고, 필요한 경우 인터럽트를 마스크(차단)하거나 특정 인터럽트를 다룰 준비를 합니다.

초기 PC 시스템에서는 두 개의 PIC가 사용되었는데, 하나는 마스터 PIC이고 다른 하나는 슬레이브 PIC입니다. 마스터 PIC는 보통 IRQ0 부터 IRQ7 까지의 인터럽트 라인을 관리하며, 슬레이브 PIC는 IRQ8 부터 IRQ15 까지를 관리합니다. 슬레이브 PIC는 마스터 PIC에 연결되어 있으며, 마스터 PIC를 통해 CPU와 통신합니다.

마스터 PIC의 커맨드 레지스터 포트(0x20)란?

마스터 PIC의 커맨드 레지스터 포트(0x20)는 PIC에 명령을 보내기 위한 I/O 포트 주소입니다. 이 포트를 통해 CPU는 마스터 PIC에 다양한 명령을 전송할 수 있습니다. 이런 명령에는 초기화 명령, 인터럽트 마스킹/언마스킹, 인터럽트 요청의 우선 순위 변경 등이 포함될 수 있습니다.

예를 들어, 시스템을 부팅할 때 마스터 PIC를 초기화하기 위해 0x11 이라는 특정 값을 마스터 PIC의 커맨드 레지스터 포트로 보내야 합니다. 이 값은 PIC가 초기화 명령을 받아들일 준비가 되었음을 알리며, 이후에 PIC는 추가적인 초기화 명령어를 기다립니다.

간단히 말해, 마스터 PIC는 인터럽트를 관리하는 주된 장치이며, I/O 포트 0x20 은 마스터 PIC에 명령을 보내는 데 사용되는 "우편함" 같은 것입니다.

```
dw 0x00eb, 0x00eb   ; 소프트웨어 딜레이 (jmp $+2)
```

어셈블리 코드에서 dw 지시어는 "define word"의 약자로, 메모리에 2 바이트(16 비트)의 데이터를 정의하는 데 사용됩니다. dw 0x00eb, 0x00eb 라인은 두 개의 16 비트 값 0x00eb를 메모리에 연속으로 배치하라는 의미입니다.

그러나 여기서 0x00eb는 단순한 데이터 값이 아니라 실제로는 작은 기계어 명령어입니다. 0x00eb는 어셈블리 언어에서 jmp $+2 에 해당하는 기계어 코드로, 프로그램 카운터를 현재 위치에서 2 바이트만큼 앞으로 이동시키라는 뜻입니다. 즉, 다음 명령어로 바로 건너뛰어 실행하라는 명령입니다.

jmp $+2 는 CPU가 다음 명령어로 즉시 이동하게 함으로써, 실질적으로 아무 일도 하지 않게 하는 효과를 낳습니다. 이는 nop (no operation) 명령어와 유사한 결과를 가져오는데, nop 명령어는 CPU에게 아무런 연산도 수행하지 말고 다음 명령어로 넘어가라고 지시합니다.

이러한 "빈" 명령어는 I/O 명령어 사이에 작은 지연을 만들기 위해 사용됩니다. I/O 포트로 데이터를 보낼 때, PIC와 같은 하드웨어 장치가 다음 명령어를 받기 위해 준비될 수 있는 충분한 시간이 필요할 수 있기 때문입니다. 이런 지연을 통해 CPU와 하드웨어 장치 사이의 타이밍 문제를 방지할 수 있습니다.

```
out 0xA0, al   ; 슬레이브 PIC(커맨드 레지스터)에 명령어 전송
```

동일한 초기화 명령어를 슬레이브 PIC의 커맨드 레지스터 포트(0xA0)로 전송합니다.

슬레이브 PIC란?

- 슬레이브 PIC는 추가 인터럽트 라인을 시스템에 제공하며, 마스터 PIC에 연결되어 있습니다.
- 마스터 PIC는 IRQ(Interrupt Request Line) 0 부터 7 까지를 처리하고, 슬레이브 PIC는 IRQ 8 부터 15 까지를 처리합니다.
- 슬레이브 PIC는 마스터 PIC의 특정 IRQ 라인(보통 IRQ2)에 연결되어, 마스터 PIC를 통해 CPU에 인터럽트 신호를 전달합니다.

슬레이브 PIC의 커맨드 레지스터 포트(0xA0)란?

- 슬레이브 PIC의 커맨드 레지스터 포트는 I/O 포트 주소 0xA0 에 위치합니다.
- 이 포트를 통해 CPU는 슬레이브 PIC에 명령어를 전송할 수 있습니다.
- 이 포트는 PIC를 초기화하거나, 인터럽트 처리 방식을 설정하는 등의 작업에 사용됩니다.

어셈블리 코드에서 out 명령어는 CPU에서 하드웨어 장치로 데이터를 전송하는 데 사용됩니다. out 0xA0, al 명령은 al 레지스터에 저장된 데이터(이 경우에는 초기화 명령어 0x11)를 슬레이브 PIC의 커맨드 레지스터로 보내어 슬레이브 PIC를 초기화하거나 설정하는 데 사용됩니다.

```
mov al, 0x20   ; 마스터 PIC 인터럽트 벡터 시작점 (IRQs 0-7)
```

마스터 PIC에 대한 인터럽트 벡터 시작점을 설정합니다. 여기서 0x20 은 IRQ 0 부터 IRQ 7 까지의 인터럽트를 처리할 때 사용할 인터럽트 벡터의 시작점(오프셋)입니다.

인터럽트 서비스 루틴(ISR)이란?

인터럽트 서비스 루틴(Interrupt Service Routine, ISR)은 컴퓨터 시스템에서 특정 인터럽트를 처리하기 위해 실행되는 함수 또는 프로시저입니다. 각종 하드웨어 장치나 소프트웨어에서 발생하는 인터럽트에 반응하여 실행되며, 해당 인터럽트에 적절히 대응하는 코드를 포함하고 있습니다.

ISR의 주요 역할은 다음과 같습니다:

1. 인터럽트 처리: ISR은 인터럽트가 발생했을 때 수행해야 할 작업을 정의합니다. 예를 들어, 키보드에서 키 입력이 발생하면 키보드 인터럽트의 ISR이 호출되어 입력된 키를 처리합니다.
2. 상태 저장 및 복원: 인터럽트가 발생하면, ISR은 현재 실행 중인 코드의 상태를 저장하고 인터럽트 처리가 끝난 후에 원래 코드로 복귀하기 전에 상태를 복원합니다.
3. 인터럽트 우선 순위 관리: 다중 인터럽트 환경에서는 ISR이 우선 순위에 따라 인터럽트를 처리합니다. 중요한 인터럽트가 먼저 처리되어야 할 때 ISR은 이를 보장합니다.
4. 하드웨어 제어: ISR은 하드웨어 장치의 상태를 읽고, 장치를 제어하며, 필요한 경우 데이터를 전송합니다.

각 인터럽트는 고유한 번호(IRQ 번호)를 갖고 있으며, 인터럽트 벡터 테이블(IVT)에는 이 번호에 해당하는 ISR의 주소가 저장되어 있습니다. 인터럽트가 발생하면 CPU는 IVT를 참조하여 관련 ISR을 찾아 실행합니다.
ISR은 시스템의 반응성과 신뢰성을 높이는 데 매우 중요합니다. 그러나 ISR은 가능한 한 빠르게 실행되어야 하며, 시스템의 다른 부분에 영향을 주지 않도록 주의해야 합니다. ISR 내에서는 복잡하거나 시간이 많이 걸리는 작업을 피하고, 가능한 한 빨리 인터럽트를 처리한 후 원래 프로세스로 복귀하는 것이 좋습니다.

인터럽트 벡터란?

인터럽트 벡터는 인터럽트 서비스 루틴(ISR)의 주소를 가리키는 포인터입니다. 각 인터럽트 유형에는 고유한 인터럽트 벡터가 있으며, 이는 해당 인터럽트가 발생했을 때 수행해야 할 코드의 시작점을 CPU에 알려줍니다.

- 인터럽트 벡터 테이블(Interrupt Vector Table, IVT): 시스템의 모든 인터럽트 벡터가 저장되어 있는 테이블입니다. 이 테이블은 메모리 상의 고정된 위치에 있으며, 시스템이 부팅될 때 설정됩니다.
- 인터럽트 벡터 주소: 각 인터럽트 벡터는 인터럽트 핸들러의 메모리 주소를 가지고 있습니다. CPU가 인터럽트를 받으면 IVT를 참조하여 해당 인터럽트에 대한 ISR의 위치를 찾아 실행합니다.

인터럽트 벡터 시작점이란?

- 인터럽트 벡터 시작점: 특정한 인터럽트 컨트롤러의 인터럽트에 대한 벡터의 시작 주소를 나타냅니다.
- 예를 들어, 마스터 PIC의 경우 0x20 은 IVT 내에서 마스터 PIC가 관리하는 첫 번째 인터럽트(IRQ 0)의 벡터 주소의 시작점입니다. 실제 벡터 주소는 0x20 + IRQ 번호로 계산됩니다.

이 주소는 인터럽트 신호가 PIC로부터 CPU에 전달될 때 사용되는 오프셋으로, CPU는 이 오프셋을 기준으로 ISR의 실제 메모리 주소를 결정합니다. 이렇게 인터럽트 벡터 시작점을 설정함으로써, 운영 체제는 인터럽트 번호에 따라 적절한 ISR을 찾아 제어를 넘길 수 있습니다.
0x20 으로 설정하는 것은 일반적으로 보호 모드에서 사용하기 위해 IVT의 기본 범위(0x00-0x1F)를 벗어나는 범위로 인터럽트를 리매핑하는 것입니다. 이는 기본적으로 CPU에 의해 사용되는 인터럽트와 충돌을 방지하기 위한 것입니다.

```
out 0x21, al   ; 마스터 PIC(데이터 레지스터)에 오프셋 전송
```

설정된 인터럽트 벡터 오프셋을 마스터 PIC의 데이터 레지스터 포트(0x21)로 전송합니다.

마스터 PIC의 데이터 레지스터 포트(0x21)란?

마스터 PIC(Programmable Interrupt Controller)의 데이터 레지스터 포트는 0x21 에 위치해 있습니다. 이 데이터 레지스터 포트는 주로 PIC의 설정을 변경하는 데 사용됩니다.

주요 사용 사례는 다음과 같습니다:

인터럽트 마스크 설정: PIC의 데이터 레지스터를 통해 특정 인터럽트를 활성화하거나 비활성화할 수 있습니다. 각 비트는 해당 인터럽트 라인의 마스크 상태를 나타내며, 1 은 마스크 되어 있음(비활성화)을, 0 은 마스크가 해제됨(활성화)을 나타냅니다.
인터럽트 벡터 기본 주소 설정: 초기화 과정에서 ICW2 라는 명령어를 통해 인터럽트 벡터 테이블에서의 PIC 인터럽트의 시작 주소를 설정할 수 있습니다. 이 값은 데이터 레지스터를 통해 전달됩니다.
특수 명령어 실행: PIC에는 특정 동작을 제어하는 데 사용되는 특수 명령어가 있습니다. 예를 들어, End of Interrupt(EoI) 명령어는 현재 처리 중인 인터럽트가 종료되었음을 PIC에 알리는 데 사용되며, 이 명령어는 데이터 레지스터를 통해 전달됩니다.

따라서, 마스터 PIC의 데이터 레지스터 포트는 인터럽트 처리와 관련된 중요한 역할을 담당하고 있습니다.

왜 초기화는 마스터 PIC의 커맨드 레지스터 포트(0x20)에 하고 시작점은 데이터 레지스터 포트(0x21)에 하는가?
PIC(Programmable Interrupt Controller)의 초기화 및 설정 과정은 특정한 순서를 따르도록 설계되어 있습니다. 이는 PIC가 제어를 위한 다양한 명령어를 받아들이고 처리할 수 있는 방법을 정의하는데 필요합니다.

초기화 작업은 PIC의 커맨드 레지스터를 통해 수행됩니다. 이는 PIC에게 '초기화를 시작하라'는 명령을 내리는 것으로, ICW1(Initialization Command Word 1)라는 특별한 명령어를 전달하는 것입니다.

이 초기화 명령은 PIC을 초기화 상태로 전환하고, 이후에 전달되는 명령어들을 초기화 명령어로 해석하게 합니다. 따라서 이 단계에서는 PIC의 동작을 제어하는 데 필요한 일련의 초기화 명령어를 전달할 수 있게 됩니다.

그 다음 단계에서는 데이터 레지스터를 통해 ICW2, ICW3, ICW4 등의 추가 초기화 명령어를 전달합니다. ICW2 는 인터럽트 벡터 테이블에서의 시작 주소를 설정하며, ICW3 는 마스터와 슬레이브 PIC의 연결 상태를 설정하고, ICW4 는 PIC의 동작 모드를 설정합니다.

따라서, 초기화는 커맨드 레지스터를 통해 수행되며, 이후의 설정 작업은 데이터 레지스터를 통해 이루어지는 것입니다. 이런 방식은 PIC의 동작을 세밀하게 제어할 수 있도록 하며, PIC의 초기화 및 설정 과정을 명확하게 정의하고 있습니다.

```
mov al, 0x28   ; 슬레이브 PIC 인터럽트 벡터 시작점 (IRQs 8-15)
```

슬레이브 PIC에 대한 인터럽트 벡터 시작점을 설정합니다. 0x28 은 IRQ 8 부터 IRQ 15 까지의 인터럽트를 처리할 때 사용할 인터럽트 벡터의 시작점입니다.

```
out 0xA1, al   ; 슬레이브 PIC(데이터 레지스터)에 오프셋 전송
```

설정된 인터럽트 벡터 오프셋을 슬레이브 PIC의 데이터 레지스터 포트(0xA1)로 전송합니다.

```
mov al, 0x04   ; 마스터 PIC 의 IRQ 2 번에 슬레이브 PIC 이 연결되어 있다.
```

마스터 PIC의 IRQ 2 번 라인이 슬레이브 PIC와 연결되어 있음을 설정합니다. 이는 마스터 PIC가 슬레이브 PIC로부터 인터럽트 신호를 받을 수 있게 합니다.

0x04 주소값이란?

마스터 PIC의 관점에서, 슬레이브 PIC가 자신의 IRQ 2 번 라인에 연결되어 있음을 설정.

왜 마스터 PIC의 2 번 라인에 슬레이브 PIC를 연결하는건가?

마스터 PIC의 2 번 라인에 슬레이브 PIC를 연결하는 것은 초기 x86 시스템의 설계상의 제약 때문입니다.

초기의 IBM PC는 8 개의 인터럽트만을 처리할 수 있는 단일 PIC (8259A)를 사용하였습니다. 이는 기본적인 시스템 요구사항을 충족시키지만, 시간이 지나며 추가 장치와 함께 인터럽트 요구사항이 증가하였습니다. 이를 해결하기 위해 추가적인 PIC, 즉 슬레이브 PIC를 도입하였고, 이를 마스터 PIC에 연결해야 했습니다.

이 연결을 위해선스터 PIC의 임의의 라인이 필했는데, 여기서 선택된 것이 2 번 라인이었습니다. 이는 단순히 하드웨어 설계상의 결정이었습니다. 이렇게 연결된 슬레이브 PIC는 마스터 PIC를 통해 CPU에 인터럽트를 전달할 수 있게 됩니다.

따라서, 마스터 PIC의 2 번 라인에 슬레이브 PIC를 연결하는 것은 이러한 역사적인 배경과 하드웨어 설계상의 결정에 기인한 것입니다.

```
out 0x21, al   ; 마스터 PIC(데이터 레지스터)에 설정 전송
```

마스터 PIC에 슬레이브 PIC 연결 설정을 전송합니다.

```
mov al, 0x02   ; 슬레이브 PIC 이 마스터 PIC 의 IRQ 2 번에 연결되어 있다.
```

슬레이브 PIC가 마스터 PIC의 어느 IRQ 라인에 연결되어 있는지 설정합니다. 여기서는 IRQ 2 번에 연결되어 있음을 나타냅니다.

0x02 란?

슬레이브 PIC의 관점에서, 자신이 마스터 PIC의 IRQ 2 번 라인에 연결되어 있음을 설정.

mov al, 0x04 에서 마스터 PIC의 관점에서 2 번 라인에 연결하였는데, 왜 슬레이브 PIC의 관점에서 또 2 번 라인에 연결하였나?

'mov al, 0x04'는 마스터 PIC의 관점에서, 슬레이브 PIC가 자신의 IRQ 2 번 라인에 연결되어 있음을 설정합니다. 반면 'mov al, 0x02'는 슬레이브 PIC의 관점에서, 자신이 마스터 PIC의 IRQ 2 번 라인에 연결되어 있음을 설정합니다.

이렇게 두 번 설정하는 이유는 인터럽트 체인을 정확하게 구성하기 위해서입니다. 마스터 PIC는 슬레이브 PIC로부터 인터럽트를 받아야 하며, 슬레이브 PIC는 마스터 PIC에 인터럽트를 보낼 수 있어야 합니다. 이를 위해 양쪽 모두에서 연결을 설정해야 합니다.

```
out 0xA1, al   ; 슬레이브 PIC(데이터 레지스터)에 설정 전송
```

슬레이브 PIC에 마스터 PIC와의 연결 설정을 전송합니다.

```
mov al, 0x01   ; 8086/88 (MCS-80/85) 모드 설정
```

PIC를 8086/88 모드로 설정합니다. 이는 PIC가 8086 호환 모드에서 인터럽트를 처리하도록 설정하는 것입니다.

8068/88 모드가 뭔가? PIC가 8068/88 모드가 되면 어떻게 되고 왜 하는 건가?

8086/88 모드는 Intel 8086 및 8088 마이크로프로세서에 최적화된 모드를 말합니다. 이 모드가 설정되면, PIC(Programmable Interrupt Controller)는 이들 마이크로프로세서와 호환되는 방식으로 인터럽트를 처리하게 됩니다.

8086/88 모드가 설정되면, PIC는 인터럽트를 처리할 때 특정 프로토콜을 따르게 됩니다. 이 프로토콜은 8086/88 마이크로프로세서와 통신하기 위해 필요한 메커니즘을 정의합니다. 예를 들어, 인터럽트 요청(IRQ)이 발생하면, PIC는 CPU에 인터럽트 벡터를 전달합니다. 이 벡터는 CPU가 어떤 인터럽트 서비스 루틴(ISR)을 호출해야 하는지를 알려줍니다.

8086/88 모드를 설정하는 이유는 시스템의 마이크로프로세서가 8086/88 인 경우, PIC와 마이크로프로세서 사이의 통신을 원활하게 하기 위해서입니다. 이렇게 함으로써, PIC는 마이크로프로세서와 효율적으로 통신하며 인터럽트를 정확하게 처리할 수 있습니다.

마이크로프로세서란?

마이크로프로세서는 컴퓨터의 중앙 처리 장치(CPU)의 기능을 수행하는 집적 회로입니다. 이는 컴퓨터의 두뇌라고 볼 수 있으며, 데이터를 처리하고 명령어를 실행하는 역할을 담당합니다.

마이크로프로세서는 연산 장치, 제어 장치, 메모리 등 여러 컴포넌트를 하나의 칩에 집적하여 만들어졌습니다. 이로 인해 크기가 작고 전력 소모가 적으며, 빠른 처리 속도를 가질 수 있습니다.

마이크로프로세서의 종류는 다양하며, 그 중에서도 Intel의 8086, 8088 등은 초기 개인용 컴퓨터(PC)에 널리 사용되었습니다. 이러한 마이크로프로세서는 하드웨어와 소프트웨어, 그리고 주변 장치들과 통신하기 위한 여러 모드를 지원하며, 이 중 하나가 앞서 언급한 8086/88 모드입니다.

```
out 0x21, al   ; 마스터 PIC 에 전송
out 0xA1, al   ; 슬레이브 PIC 에 전송
```

설정된 모드(8086/88 모드)를 마스터 PIC와 슬레이브 PIC에 전송합니다.

```
mov al, 0xFF   ; 슬레이브 PIC 의 모든 인터럽트를 막아둔다.
```

슬레이브 PIC의 모든 인터럽트를 비활성화합니다. 이는 모든 인터럽트 라인을 마스크하여 신호를 받지 않게 합니다.

```
out 0xA1, al   ; 슬레이브 PIC 인터럽트 마스킹
```

슬레이브 PIC의 모든 인터럽트를 비활성화하는 설정을 전송합니다.

```
mov al, 0xFB   ; 마스터 PIC 의 IRQ 2 번을 제외한 모든 인터럽터를 막아둔다.
```

마스터 PIC의 모든 인터럽트를 비활성화하되, IRQ 2 번 라인(슬레이브 PIC로부터의 인터럽트를 받기 위해)은 활성화 상태로 둡니다.

```
out 0x21, al   ; 마스터 PIC 인터럽트 마스킹
```

마스터 PIC의 인터럽트 마스킹 설정을 전송합니다.

왜 인터럽트 마스킹을 하는가?

인터럽트 마스킹은 특정 인터럽트를 임시적으로 무시하거나 차단하는 기능을 말합니다. 이는 CPU가 특정 작업에 집중하거나, 중요한 작업을 안전하게 수행할 수 있도록 도와줍니다.

예를 들어, CPU가 임계 영역의 코드를 실행하거나, 복잡한 연산을 수행하는 중일 때, 인터럽트가 발생하면 원치 않는 결과를 초래할 수 있습니다. 이러한 상황을 피하기 위해, 운영 체제나 특정 프로그램은 일시적으로 인터럽트를 마스킹하여 차단할 수 있습니다.
위의 코드에서는 'mov al, 0xFF'와 'mov al, 0xFB' 명령어로 슬레이브 PIC의 모든 인터럽트를 차단하고, 마스터 PIC의 2 번 IRQ를 제외한 모든 인터럽트를 차단하고 있습니다. 이는 시스템 초기화 과정 중에 원하지 않는 인터럽트가 시스템의 안정성을 해치는 것을 방지하기 위한 것입니다. 이후에 필요에 따라 개별 인터럽트를 다시 활성화할 수 있습니다.

∴ 이렇게 PIC를 초기화하고 설정함으로써, 시스템은 인터럽트를 받을 준비가 되며, 인터럽트 핸들러를 통해 인터럽트를 처리할 수 있게 됩니다.

이 코드는 컴퓨터 시스템에서 인터럽트를 관리하는 Programmable Interrupt Controller(PIC)를 초기화하고 설정하는 역할을 하는 Assembly 언어 코드입니다.
Assembly 언어는 컴퓨터의 하드웨어와 가까운 저수준 프로그래밍 언어로, 이 코드는 아래와 같은 일련의 작업을 수행합니다:
1. PIC 초기화
2. 인터럽트 벡터 시작점 설정
3. 슬레이브 PIC와 마스터 PIC 사이의 연결 설정
4. 8086/88 모드 설정
5. 인터럽트 마스킹
이 작업들을 통해, 시스템은 각종 장치로부터 발생하는 인터럽트를 적절히 처리할 수 있습니다. 이는 컴퓨터가 여러 작업을 동시에 처리하거나, 특정 작업에 집중하도록 하는 데 필요한 과정입니다.

- IDT 선언

IDT란?

IDT(Interrupt Descriptor Table)는 인터럽트 처리를 관리하는데 사용되는 데이터 구조입니다. 인터럽트 발생시 CPU는 IDT를 참조하여 해당 인터럽트에 대한 처리루틴(ISR, Interrupt Service Routine)의 주소를 찾아냅니다. IDT는 각 인터럽트 벡터에 대해 인터럽트 게이트, 트랩 게이트, 태스크 게이트 등의 정보를 포함하고 있습니다.

IDT와 GDT는 뭐가 다른가?

GDT는 시스템의 메모리 세그먼트에 대한 정보를 저장하는 테이블입니다. 이는 각 세그먼트의 베이스 주소, 크기, 액세스 권한 등의 정보를 포함하고 있습니다. GDT를 통해 운영체제는 메모리 보호를 구현하고, 프로세스 간의 메모리 공간을 분리합니다.
반면, IDT는 인터럽트 처리에 필요한 정보를 저장하는 테이블입니다. 이는 각 인터럽트 벡터에 대응하는 인터럽트 처리루틴의 주소와, 그 처리루틴의 액세스 권한 등의 정보를 포함하고 있습니다. IDT를 통해 운영체제는 인터럽트를 효율적으로 처리하고, 시스템의 안정성을 유지합니다.
따라서, GDT와 IDT는 둘 다 중요한 시스템 테이블이지만, 그 용도와 저장하는 정보의 종류가 다릅니다. GDT는 메모리 관리에, IDT는 인터럽트 처리에 초점을 둡니다.

PIC, IDT, ISR의 관계는?

1. PIC(Programmable Interrupt Controller): PIC는 컴퓨터 시스템에서 발생하는 여러 인터럽트를 관리하는 하드웨어 장치입니다. 여러 장치로부터 발생하는 인터럽트를 받아 처리하거나 CPU로 전달하는 역할을

합니다. PIC는 마스터 PIC와 슬레이브 PIC로 구성되며, 이들은 인터럽트를 받아서 CPU에 알리는 역할을 합니다.

2. IDT(Interrupt Descriptor Table): IDT는 인터럽트 처리를 위한 정보를 저장하는 테이블입니다. 인터럽트가 발생하면 CPU는 IDT를 참조하여 인터럽트 처리루틴(ISR)의 주소를 찾습니다. IDT는 각 인터럽트 별로 인터럽트 게이트, 트랩 게이트, 태스크 게이트 등의 정보를 담고 있습니다.
3. ISR(Interrupt Service Routine): ISR은 특정 인터럽트에 대한 처리를 수행하는 함수 또는 루틴입니다. 인터럽트가 발생하면 CPU는 IDT를 참조하여 해당 인터럽트에 대응하는 ISR을 호출합니다. ISR은 인터럽트를 처리한 후에 CPU의 실행 흐름을 원래대로 복구합니다.

PIC, IDT, ISR의 관계는 다음과 같습니다:

- PIC는 여러 인터럽트 중에서 CPU가 처리해야 할 인터럽트를 결정하고, 그 인터럽트를 CPU에게 알립니다.
- CPU는 PIC로부터 인터럽트를 받으면 IDT를 참조하여 해당 인터럽트에 대응하는 ISR의 주소를 찾습니다.
- 찾은 ISR을 실행하여 인터럽트를 처리하고, 처리가 완료되면 원래의 작업으로 돌아갑니다.

이렇게 PIC, IDT, ISR은 함께 작동하여 인터럽트를 효율적으로 처리합니다.

IDT를 구현하는 방식은 GDT를 구현했던 방식과 똑같습니다. 각 인터럽트에 해당하는 정보들을 담아주고 이를 lidt라는 특정 레지스터에 등록해주면 됩니다.

```
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:
```

예전에 구현했던 GDT 테이블 코드를 살펴보면, 똑같이 gdt: ~ gdt_end: 부분을 만들어주고, gdtr: 부분에다 크기와 주소를 각각 저장해놓은 후 이를 lgdt가 하는 것처럼 등록해 놓으면 끝입니다. GDT를 등록하는 원리와 IDT를 등록하는 원리가 완벽하게 같다는 뜻입니다.

단지 차이점이 있다면 다음과 같습니다.

1. GDT와 다르게 IDT의 각 오프셋은 실제 인터럽트 발생 시 실행될 주소(ISR)를 가지고 있어야 한다.
   인터럽트가 발생하게 되면 CPU는 IDT를 참고하게 되고 저장되어 있는 주소(ISR)로 이동합니다. 그 주소에는 실제 인터럽트 발생 시 실행하고 싶은 명령어들이 있어야 합니다. 우리는 이를 함수로 만들어 놔서 해당 IDT 테이블들이 이 함수를 가리키게 만들어야 합니다. 그리고 각각의 함수에 코드를 작성해 놓으면 인터럽트 발생시 이 함수로 이동해서 함수를 실행할 수 있습니다.

2. IDT는 물리주소 0x0 번지에 위치해야 한다.
   IDT는 보호 모드에서의 인터럽트 처리를 위해 필요한 시스템 테이블입니다. 이 테이블은 특정 메모리 위치에 배치되어야 하는데, 이 코드에서는 이를 물리 주소 0x0 번지에 배치하도록 하고 있습니다. 이는 인터럽트 처리를 위한 초기 설정 과정의 일부로, IDT의 내용을 생성한 후에 이를 물리 주소 0x0 번지 위치로 복사하는 작업이 필요합니다.

   ### 왜 IDT는 물리주소 0x0 번지에 위치해야 하는가?
   IDT가 물리 주소 0x0 번지에 위치해야 하는유는 아키텍처나 운영 체제의 설계에 따라 다릅니다. 일반적으로, 초기 시스템 설정 과정에서 IDT를 특정 위치에 배치하는 것이 필요하며, 이 위치는 시스템의 요구사항에 따라 달라집니다.

3. IDT 크기는 256 개다.
   모든 가능한 인터럽트 벡터를 처리할 수 있도록 IDT는 256 개의 엔트리를 가지고 있습니다. 그러나 모든 인터럽트를 개별적으로 처리하는 것은 비효율적일 수 있으므로, 대부분의 인터럽트는 기본 처리 루틴(idt_ignore)을 가리키도록 설정됩니다. 이후에 필요에 따라 특정 인터럽트(예: 타이머, 키보드 인터럽트)에 대한 처리 루틴을 개별적으로 정의하고, 해당 IDT 엔트리를 각각의 처리 루틴을 가리키도록 변경합니다. 이렇게 하면, 필요한 인터럽트만 효과적으로 처리할 수 있습니다.

   ### 왜 IDT 크기는 256 인가?
   IDT의 크기가 256 인 이유는 x86 아키텍처에서 인터럽트 벡터의 개수가 256 개이기 때문입니다. 즉, 각각의 인터럽트 벡터에 대응하는 IDT 엔트리가 필요하므로, IDT의 크기는 256 이 됩니다.

   ### 엔트리가 뭔가?
   엔트리(entry)는 테이블이나 목록, 배열 등의 개별 요소를 의미하는 컴퓨팅 용어입니다.
   예를 들어, 인터럽트 디스크립터 테이블(Interrupt Descriptor Table, IDT)이나 글로벌 디스크립터 테이블(Global Descriptor Table, GDT) 같은 시스템 테이블에서 엔트리는 테이블의 한 줄을 차지하는 정보 단위를 의미합니다.
   각 엔트리는 특정 정보를 저장하고 있으며, 이 정보는 테이블의 용도에 따라 다릅니다. 예를 들어, IDT의 엔트리는 인터럽트 서비스 루틴(ISR)의 주소, 인터럽트 타입 등의 정보를 담고 있습니다. 이런 방식으로 테이블은 여러 개의 엔트리로 구성되어 있으며, 각 엔트리는 테이블의 특정 정보를 나타냅니다.

이제 우리는 이를 C언어로 구현하려고 합니다. 그렇다면 ... gdt: ~ gdt_end: 같이 반복되는 타입을 저장하는 것을 C언어에서는 어떻게 구현할 수 있을까요? 구조체로 구현할 수 있습니다.

Interrupt.h 파일을 하나 만들고 다음과 같이 적어봅니다.

```
#pragma once

void init_intdesc();
void idt_ignore();
void idt_timer();
void idt_keyboard();

struct IDT {
    unsigned short offsetl;
    unsigned short selector;
    unsigned short type;
    unsigned short offseth;
} __attribute__((packed));

struct IDTR {
    unsigned short size;
    unsigned int addr;
} __attribute__((packed));
```

IDT에는 offset_low, selector, type, offset_high 같이 IDT 스펙을 그대로 따르는 부분을 정의하였습니다. 여기서 short는 2 바이트, 즉 16 비트를 저장할 수 있는 자료형입니다. 그런데 그냥 short라고 선언하면 음수가 저장되서 엉뚱한 값이 나올수도 있어서 unsigned를 붙였습니다.

IDTR은 전체 IDT 테이블의 개수, 그리고 주소를 저장할 수 있습니다. Int형은 4 바이트, 즉 32 비트를 저장할 수 있는 자료형입니다.

IDT 스펙이 뭔가? offset_low, selector, type, offset_high이 IDT 스펙인가?

IDT(Interrupt Descriptor Table) 스펙은 인터럽트 처리를 위해 x86 아키텍처에서 정의한 표준입니다. 이 스펙에 따르면, 각 IDT 엔트리는 특정한 구조를 가지며, 이 구조는 다음과 같은 필드를 포함합니다:

1. Offset: 인터럽트 서비스 루틴(ISR)의 주소를 나타냅니다. 이 주소는 32 비트로 표현되지만, IDT 엔트리 내에서는 두 개의 16 비트 필드(offset_low와 offset_high)로 나뉘어 저장됩니다.
2. Selector: 세그먼트 셀렉터를 나타냅니다. 이는 ISR이 위치한 세그먼트를 가리키는 값입니다.
3. Type: 인터럽트 게이트의 타입과 속성을 나타냅니다. 이 필드는 인터럽트가 발생했을 때 어떤 동작을 해야하는지를 정의합니다.

따라서, IDT 스펙은 IDT 엔트리의 구조와 각 필드의 의미를 정의하는 표준입니다. 이 스펙은 인터럽트 처리의 일관성과 효율성을 보장하는 데 중요한 역할을 합니다.

왜 2 바이트(16 비트)로 자료형을 결정한건가?

2 바이트(16 비트)로 자료형을 결정한 이유는 IDT 스펙이 그렇게 정의되어 있기 때문입니다. 즉, IDT 엔트리의 각 필드는 특정 크기를 가지고 있어야 하며, 이 크기는 x86 아키텍처의 인터럽트 처리 방식에 따라 결정됩니다.

IDT 테이블이 뭔가?

IDT(Interrupt Descriptor Table) 테이블은 시스템의 모든 인터럽트 벡터에 대응하는 IDT 엔트리를 모아 놓은 테이블입니다. IDT 테이블은 인터럽트 처리를 위한 중요한 자료구조로, 인터럽트가 발생하면 CPU는 이 테이블을

참조하여 해당 인터럽트에 대응하는 인터럽트 서비스 루틴(ISR)의 주소를 찾아 이동하고, ISR에 정의된 명령어들을 실행합니다.
IDT 테이블의 각 엔트리는 다음과 같은 정보를 담고 있습니다:
Offset: 인터럽트 서비스 루틴(ISR)의 주소를 나타냅니다. 이 주소는 32 비트로 표현되지만, IDT 엔트리 내에서는 두 개의 16 비트 필드(offset_low와 offset_high)로 나뉘어 저장됩니다.
Selector: 세그먼트 셀렉터를 나타냅니다. 이는 ISR이 위치한 세그먼트를 가리킵니다.
Type: 인터럽트 게이트의 타입과 속성을 나타냅니다.
따라서, IDT 테이블은 시스템의 인터럽트 처리를 관리하는 중요한 역할을 수행합니다.

IDT(Interrupt Descriptor Table)와 IDT 테이블은 같은 것을 지칭합니다. 둘 다 시스템의 모든 인터럽트 벡터에 대응하는 인터럽트 서비스 루틴(ISR)의 정보를 가진 테이블을 의미합니다.
즉, IDT는 인터럽트 벡터를 ISR의 주소로 매핑하는 역할을 하는 테이블이며, 이는 인터럽트가 발생했을 때 CPU가 실행해야 할 코드를 찾는 데 사용됩니다. 이러한 테이블을 일반적으로 IDT라고 부르지만, 테이블이라는 의미를 명확히 하기 위해 IDT 테이블이라고 부르기도 합니다.
따라서, IDT와 IDT 테이블은 동일한 개념을 나타내며, 둘 사이에는 차이가 없습니다.

C interrupt.h

```
#pragma once
```

이는 헤더 파일이 한 번만 포함되도록 하는 전처리 지시자입니다. 이를 통해 헤더 파일의 중복 포함을 방지할 수 있습니다.

```
void init_intdesc();
```

이 함수는 인터럽트 디스크립터 테이블(IDT)을 초기화하는 역할을 합니다.
인터럽트 디스크립터 테이블(IDT)은 컴퓨터 시스템에서 발생하는 다양한 인터럽트에 대응하는 처리 함수, 즉 인터럽트 서비스 루틴(ISR)을 가리키는 포인터를 저장하고 있는 테이블입니다.
IDT를 초기화한다는 것은, 시스템에서 발생 가능한 각각의 인터럽트에 대해 그에 대응하는 ISR을 IDT에 등록하는 것을 의미합니다. 이렇게 함으로써 실제로 인터럽트가 발생했을 때, 시스템이 어떤 동작을 수행해야 하는지를 빠르게 찾아낼 수 있습니다.
따라서, 이 함수는 시스템의 인터럽트 처리를 위한 준비 작업의 일환으로, IDT에 각 인터럽트의 ISR을 등록하는 초기화 작업을 수행합니다. 이 함수를 통해 시스템은 인터럽트가 발생하면 어떤 동작을 해야 하는지를 미리 정의된 ISR에 따라 수행하게 됩니다.

```
void idt_ignore();
```

이 함수는 인터럽트가 발생했을 때 무시하고 싶은 특정 인터럽트를 처리하는 역할을 합니다. 예를 들어, 시스템에서 중요하지 않은 인터럽트가 발생했을 때 이를 무시하고 싶다면 'idt_ignore' 함수를 해당 인터럽트의 처리 함수로 설정할 수 있습니다.

```
void idt_timer();
```

이 함수는 시스템 타이머 관련 인터럽트를 처리하는 역할을 합니다. 시스템 타이머는 컴퓨터 시스템의 핵심 구성 요소로, 정해진 시간 간격마다 인터럽트를 발생시켜 시스템의 다른 작업을 정기적으로 수행하게 합니다. 'idt_timer' 함수는 이런 타이머 인터럽트가 발생했을 때 수행할 동작을 정의합니다.

```
void idt_keyboard();
```
이 함수는 키보드 관련 인터럽트를 처리하는 역할을 합니다. 사용자가 키보드를 누르면 키보드 인터럽트가 발생하고, 이를 'idt_keyboard' 함수가 처리하여 해당 키에 대한 동작을 수행하게 됩니다.

```
struct IDT {
```
이는 인터럽트 디스크립터 테이블(IDT)의 각 항목을 나타내는 구조체를 정의하는 부분입니다. IDT는 시스템의 모든 인터럽트에 대한 처리 함수를 저장한 테이블입니다.

```
    unsigned short offsetl;
```
이 멤버는 인터럽트 처리 함수의 주소의 하위 16 비트를 저장합니다.

```
    unsigned short selector;
```
이 멤버는 인터럽트 처리 함수가 위치한 세그먼트의 세그먼트 선택자를 저장합니다.

```
    unsigned short type;
```
이 멤버는 해당 인터럽트 게이트의 타입을 저장합니다.

```
    unsigned short offseth;
```
이 멤버는 인터럽트 처리 함수의 주소의 상위 16 비트를 저장합니다.

```
} __attribute__((packed));
```
이 코드는 'IDT' 구조체의 정의를 마치며, 'attribute((packed))'는 이 구조체가 메모리에서 패딩 없이 가장 작은 공간을 차지하도록 지정합니다.

```
struct IDTR {
```
이는 인터럽트 디스크립터 테이블 레지스터(IDTR)를 나타내는 구조체를 정의하는 부분입니다. IDTR는 IDT의 크기와 시작 주소를 저장하는 레지스터입니다.

```
    unsigned short size;
```
이 멤버는 IDT의 크기를 저장합니다.

```
    unsigned int addr;
```
이 멤버는 IDT의 시작 주소를 저장합니다.

```
} __attribute__((packed));
```
이 코드는 'IDTR' 구조체의 정의를 마치며, 'attribute((packed))'는 이 구조체가 메모리에서 패딩 없이 가장 작은 공간을 차지하도록 지정합니다.

패딩이 뭔가?

패딩이란 컴퓨터 과학에서 데이터 구조나 네트워크 통신에서 데이터를 특정 단위로 정렬하기 위해 사용하는 기술입니다.

예를 들어, 메모리에서 구조체를 저장할 때, 각 멤버가 특정 바이트 경계에 맞춰 저장되도록 하기 위해 추가되는

빈 공간이 패딩입니다. 이렇게 함으로써 메모리 접근 속도를 향상시킬 수 있습니다.
그러나 이런 패딩은 때때로 메모리를 불필요하게 차지하는 문제를 일으킬 수 있습니다. 따라서 __attribute__((packed)) 지시자를 사용하여 구조체의 멤버들 사이에 패딩이 추가되지 않도록 할 수 있습니다. 이렇게 함으로써 구조체가 메모리에서 차지하는 공간을 최소화할 수 있습니다.
즉, __attribute__((packed))는 패딩 없이 구조체의 멤버들을 가장 밀집하게 메모리에 배치하도록 지시하는 것입니다.

이 코드에서 __attribute__((packed))를 하는 이유는?
이 코드에서 __attribute__((packed))는 패딩을 제거하려는 목적으로 사용되었습니다.
구조체의 패딩은 일반적으로 컴퓨터 아키텍처에서 데이터에 접근하는 데 있어 성능을 향상시키기 위한 목적으로 사용됩니다. 컴퓨터는 일반적으로 특정 크기(예: 4 바이트, 8 바이트 등)의 배수로 된 주소에서 데이터를 읽는 것이 더 빠릅니다. 따라서 구조체의 각 필드가 이러한 주소 경계에 맞춰져 있도록 패딩이 추가됩니다.
그러나 이렇게 패딩이 추가되면, 구조체가 메모리에서 차지하는 크기가 더 커질 수 있습니다. 특히, 이 구조체를 네트워크를 통해 전송하거나 디스크에 저장해야 하는 경우, 이러한 추가적인 공간은 비효율적일 수 있습니다.
따라서 이 코드에서는 __attribute__((packed))를 사용하여 구조체의 패딩을 제거하였습니다. 이렇게 함으로써 구조체가 차지하는 메모리 공간을 최소화하고, 메모리 사용을 최적화하려는 목적이 있습니다. 이는 특히 시스템 수준의 프로그래밍에서 자주 볼 수 있는 기법입니다.

∴ 이 코드는 시스템의 인터럽트 처리를 위한 인터럽트 서비스 루틴을 초기화하고, 각 인터럽트에 대한 처리 함수를 설정하고, 이를 인터럽트 디스크립터 테이블에 등록하는 작업을 수행합니다. 이를 통해 시스템은 인터럽트가 발생하면 어떤 동작을 수행해야 하는지를 알 수 있습니다.

이제 main.c에다가 interrupt.h와 위에서 했던 function.h 헤더와 함수 호출을 추가해주면 됩니다.

```c
#include "function.h"
#include "interrupt.h"

void main()
{
    kprintf("We are now in C!", 10, 10);

    init_intdesc();
}
```