주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.02.14 – 2024.02.16

세부 사항

1. 업무 내역 요약 정리

Plan	To-do
1. 부트로더 개발 2. VGA 드라이버 개발 3. 키보드 드라이버 개발 4. 쉘 개발 5. 동적 메모리 관리 개발	1. 오류 수정 2. 부트로더 개발

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

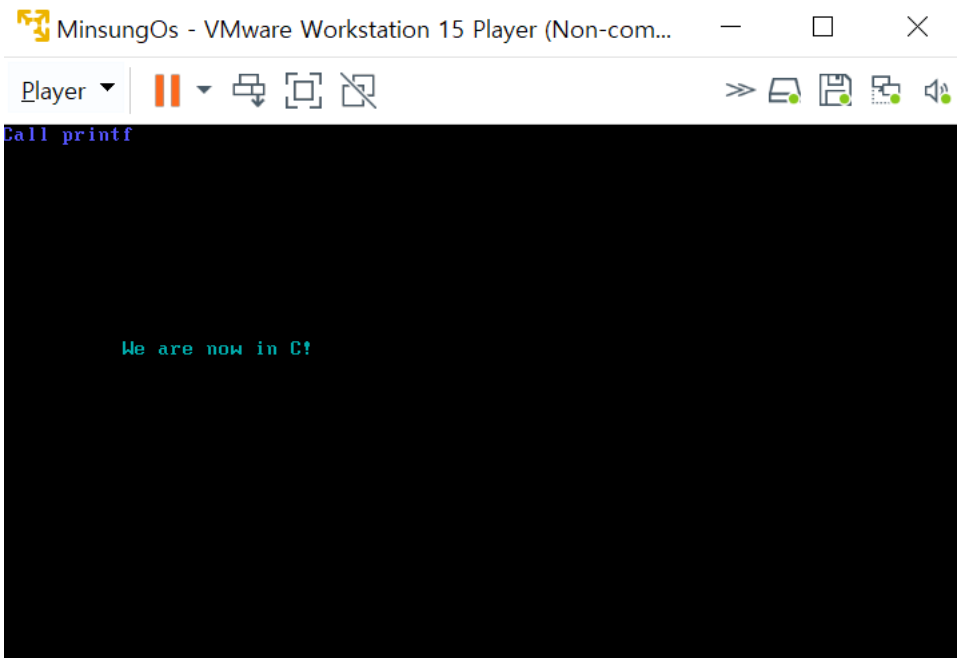
1. 오류 수정

작주에 kprintf 함수 오류라고 판단하여 인터럽트 구현에서 kprintf를 사용하지 않고 kprintf 코드를 그대로 사용하였더니 오류를 수정하였습니다.

그래서 키보드 인터럽트 구현에서도 kprintf를 사용하지 않고 kprintf 코드를 그대로 사용하였습니다.

```
__asm__ __volatile__("mov %0, al;" :"=r"(keybuf) );

{
int line = 8; // Update line number as needed
int col = 40; // Update column number as needed

// Convert coordinates to linear address
char *video = (char*)(0xB8000 + 2 * (line * 80 + col));

// Display the value of keybuf
*video++ = keybuf;
*video++ = 0x03; // Assuming text color is 0x03
}

__asm__ __volatile__
(
    "mov al, 0x20;"
    "out 0x20, al;"
);
```

오류가 수정될 줄 알았으나 아래와 같이 아무것도 출력되지 않는 오류가 발생하였습니다.



Ubuntu의 버전 문제인가 싶어서 버전을 변경하고 컴파일 해봤더니 결과가 마찬가지였습니다.

오류를 수정할 수 없다고 판단하여 그냥 무시하고 진행하려고 하였으나 블로그에 나와있는 커리큘럼 상 이를 해결하지 않으면 앞으로의 진행은 불가능하다고 판단하였습니다.

그래서 오류를 계속하여 검색하던 중 아래와 같은 참고 문헌을 찾았습니다. 기본 개념도 많이 부족한 것 같아 보충하면서 아래를 따라서 처음부터 다시 공부하며 진행해보려고 합니다.

- 참고문헌

https://dev.to/frosnerd/writing-my-own-boot-loader-3mld

https://github.com/cfenollosa/os-tutorial

https://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf

2. 부트로더 개발

Mac OS에서 x86 어셈블리어를 사용하여 간단한 부트 로더를 개발하겠습니다.

사용 도구:

QEMU(OS를 실행하기 위한 에뮬레이터), NASM, gcc(C 컴파일러), ld(링커)

어셈블러 및 에뮬레이터 설치:

```
brew install qemu nasm
```

커널 컴파일러 설치:

```
minsung@iminseocBookAir ~ $ brew install i386-elf-binutils
brew install i386-elf-gcc
brew install i386-elf-gdb
```

환경 변수 PATH 수정:

```
export PATH="/usr/local/Cellar/x86_64-elf-binutils/<version>/bin/:/usr/local/Cellar/x86_64-elf-gcc/<version>/bin/:/usr/local/Cellar/i386-elf-gdb/<version>/bin:$PATH"
```

x86 시스템에서 BIOS는 부팅 장치를 선택한 다음 장치의 첫 번째 섹터를 메모리 주소 0x7C00 의 물리적 메모리에 복사합니다. 우리의 경우 소위 부트 섹터라고 불리는 이 섹터는 512 바이트를 보유합니다. 이 512 바이트에는 부트로더 코드, 파티션 테이블, 디스크 서명은 물론 부트 섹터가 아닌 것을 실수로 로드하는 것을 방지하기 위해 BIOS에서 확인하는 "매직 넘버"가 포함되어 있습니다. 그런 다음 BIOS는 CPU에 부트로더 코드의 시작 부분으로 점프하도록 지시하여 기본적으로 부트로더에 제어권을 전달합니다.

커널을 시작하려면 부트로더가 다음 작업을 수행해야 합니다:

1. 커널을 디스트에서 메모리로 로드합니다.
2. 전역 설명자 테이블(GDT)을 설정합니다.
3. 16 비트 리얼 모드에서 32 비트 보호 모드로 전환하고 제어권을 커널에 전달합니다.

커널: 컴퓨터 시스템의 핵심 부분으로, 하드웨어와 소프트웨어 간의 상호 작용을 관리하고 시스템의 다양한 자원에 대한 액세스를 조정합니다. 이를 통해 프로세스 및 리소스 관리, 입출력 관리, 메모리 관리, 보안 및

사용자 인터페이스 등의 중요한 기능을 수행합니다.

커널의 주요 역할은 다음과 같습니다:

1. 프로세스 관리: 커널은 프로세스 생성, 종료, 일시 중단 및 스케줄링과 같은 프로세스 관리 기능을 수행합니다. 이는 멀티태스킹 및 멀티스레딩을 지원하여 여러 프로세스가 동시에 실행될 수 있도록 합니다.
2. 메모리 관리: 커널은 시스템의 물리적 및 가상 메모리를 관리합니다. 이는 메모리 할당 및 해제, 가상 메모리 관리, 페이지 교체 등을 포함합니다.
3. 입출력 관리: 커널은 입출력 장치와의 상호 작용을 관리합니다. 이는 파일 시스템, 네트워킹, 그래픽 디스플레이, 키보드 및 마우스 등 다양한 입출력 장치와의 통신을 처리합니다.
4. 시스템 호출 및 서비스 제공: 커널은 응용 프로그램이 하드웨어 및 다른 시스템 리소스에 액세스하기 위해 사용하는 시스템 호출(System Call) 인터페이스를 제공합니다. 이는 파일 I/O, 네트워킹, 프로세스 생성 및 관리 등과 같은 기능을 제공합니다.
5. 시스템 보안: 커널은 시스템의 보안을 유지하고 관리합니다. 이는 사용자 인증, 자원 액세스 제어, 프로세스 격리 등을 포함합니다.
6. 하드웨어 추상화: 커널은 하드웨어와의 상호 작용을 추상화하여 응용 프로그램이 특정 하드웨어에 종속되지 않도록 합니다. 이를 통해 다양한 하드웨어 구성에서 동일한 운영 체제를 실행할 수 있습니다.

커널은 시스템의 핵심 부분으로서, 운영 체제의 안정성, 효율성 및 기능을 보장합니다. 다양한 운영 체제들은 서로 다른 커널을 사용하며, 커널의 구현은 운영 체제의 성능과 기능에 큰 영향을 미칩니다.

코드베이스 구성:

NASM을 사용하여 x86 어셈블리에 부트로더를 작성하겠습니다. 커널은 C로 작성됩니다. 독성과 모듈성을 높이기 위해 코드를 여러 파일로 구성합니다. 다음 파일은 최소 설정과 관련이 있습니다.

- mbr.asm: 마스터 부트 기록(512 byte 부트 섹터)를 정의하는 기본 파일
- disk.asm: BIOS를 사용하여 디스크에서 읽는 코드가 포함되어 있음.
- gdt.asm: GDT를 설정
- switch-to-32bit.asm: 32 비트 보호 모드로 전환하는 코드가 포함되어 있음.
- kernel-entry.asm: kernel.c의 main 함수에 전달할 어셈블리 코드가 포함되어 있음.
- kernel.c: 커널의 주요 기능을 포함되어 있음.
- Makefile: 운영 체제를 부팅할 수 있도록 컴파일러, 링커, 어셈블리 및 에뮬레이터를 함께 연결함.

부트로더 작성:

- 마스터 부트 레코드 파일:

부트로더의 기본 어셈블리 파일에는 부트 레코드의 정의와 모든 관련 도우미 모듈에 대한 include문이 포함되어 있습니다. 먼저 파일 전체를 살펴본 다음 각 세션을 개별적으로 살펴보겠습니다.

```asm
[bits 16]
[org 0x7c00]

; where to load the kernel to
KERNEL_OFFSET equ 0x1000

; BIOS sets boot drive in 'dl'; store for later use
mov [BOOT_DRIVE], dl

; setup stack
mov bp, 0x9000
mov sp, bp

call load_kernel
call switch_to_32bit

jmp $

%include "disk.asm"
%include "gdt.asm"
%include "switch-to-32bit.asm"

[bits 16]
load_kernel:
    mov bx, KERNEL_OFFSET ; bx → destination
    mov dh, 2             ; dh → num sectors
    mov dl, [BOOT_DRIVE]  ; dl → disk
    call disk_load
    ret

[bits 32]
BEGIN_32BIT:
    call KERNEL_OFFSET ; give control to the kernel
    jmp $ ; loop in case kernel returns

; boot drive variable
BOOT_DRIVE db 0

; padding
times 510 - ($-$$) db 0

; magic number
dw 0xaa55
```

가장 먼저 주목해야 할 것은 16 비트 실제 모드와 32 비트 보호 모드 사이를 전환할 것이므로 어셈블러에 16 비트 또는 32 비트 명령어를 생성할지 여부를 알려줘야 한다는 것입니다. 이는 각각 [bits 16] 및 [bits 32] 지시어를 사용하여 수행할 수 있습니다. CPU가 여전히 16 비트 모드인 상태에서 BIOS가 부트 로더로 점프하기 때문에 16 비트 명령어로 시작합니다.

NASM에서 [org 0x7c00] 지시어는 어셈블러 위치 카운터를 설정합니다. BIOS가 부트 로더를 배치하는 메모리 주소를 지정합니다. 이는 머신 코드를 생성할 때 레이블을 메모리 주소로 변환해야 하고 해당 주소에 올바른 오프셋이 있어야 하므로 레이블을 사용할 때 중요합니다.

KERNEL_OFFSET equ 0x1000 문은 나중에 커널을 메모리에 로드하고 진입점으로 점프할 때 사용할 값 0x1000 을 가진 KERNEL_OFFSET이라는 어셈블러 상수를 정의합니다.

부트 로더를 호출하기 전에 BIOS는 선택한 부팅 드라이브를 dl 레지스터에 저장합니다. 이 정보를 덮어쓸 위험 없이 다른 용도로 dl 레지스터를 사용할 수 있도록 이 정보를 BOOT_DRIVE 변수 내부의 메모리에 저장합니다.

커널 로딩 프로시저를 호출하기 전에 스택 포인터 레지스터 sp(스택의 상단, 아래쪽으로 증가)와 bp(스택의 하단)를 설정하여 스택을 설정해야 합니다. 스택의 하단을 0x9000 에 배치하여 충돌을 피하기 위해 다른 부트로더 관련 메모리와 충분히 멀리 떨어져 있는지 확인합니다. 스택은 예를 들어 어셈블리 프로시저를 실행할 때 메모리 주소를 추적하기 위해 call 및 ret 문에서 사용됩니다.

이제 작업을 할 때가 왔습니다. 먼저 load_kernel 프로시저를 호출하여 커널을 디스크에서 KERNEL_OFFSET 주소의 메모리로 로드하도록 BIOS에 지시할 것입니다. load_kernel은 나중에 작성할 disk_load 프로시저를 사용합니다. 이 프로시저에는 세 가지 입력 매개변수가 필요합니다:

읽은 데이터를 저장할 메모리 위치(bx)
읽을 섹터 수(dh)
읽을 디스크(dl)
완료되면 곧바로 다음 명령어인 switch_to_32bit로 돌아가서 나중에 작성할 다른 도우미 프로시저를 호출합니다. 32 비트 보호 모드로 전환하는 데 필요한 모든 것을 준비하고, 전환을 수행하며, 완료되면 BEGIN_32BIT 레이블로 이동하여 효과적으로 커널에 제어권을 넘깁니다.

이것으로 기본 부트 로더 코드를 마칩니다. 유효한 마스터 부트 레코드를 생성하려면 남은 공간을 0 바이트 곱하기 510 - ($-$$) db 0 과 매직넘버 dw 0xaa55 로 채워서 패딩을 포함시켜야 합니다.

다음으로, 디스크에서 커널을 읽을 수 있도록 disk_load 프로시저가 어떻게 정의되는지 살펴봅시다.

- 디스크에서 읽기:
16 비트 모드에서 작업할 때는 인터럽트를 전송하여 BIOS 기능을 활용할 수 있기 때문에 디스크에서 읽는 것이 다소 쉽습니다. BIOS의 도움 없이는 하드 디스크나 플로피 드라이브와 같은 I/O 장치와 직접 인터페이스해야 하므로 부트 로더가 훨씬 더 복잡해집니다.

디스크에서 데이터를 읽으려면 읽기 시작 위치, 읽기 양, 메모리 내 데이터 저장 위치를 지정해야 합니다. 그런 다음 인터럽트 신호(int 0x13)를 보내면 BIOS가 각 레지스터에서 다음 파라미터를 읽으면서 작업을 수행합니다:

Register	Parameter
`ah`	Mode (0x02 = read from disk)
`al`	Number of sectors to read
`ch`	Cylinder
`cl`	Sector
`dh`	Head
`dl`	Drive
`es:bx`	Memory address to load into (buffer address pointer)

디스크 오류가 있는 경우 BIOS가 캐리 비트를 설정합니다. 이 경우 일반적으로 사용자에게 오류 메시지를

표시해야 하지만 문자열을 인쇄하는 방법을 다루지 않았고 이 게시물에서는 다루지 않을 것이므로 단순히 무한 반복하겠습니다.

이제 disk.asm의 내용을 살펴봅시다.

```
disk_load:
    pusha
    push dx

    mov ah, 0x02 ; read mode
    mov al, dh   ; read dh number of sectors
    mov cl, 0x02 ; start from sector 2
                 ; (as sector 1 is our boot sector)
    mov ch, 0x00 ; cylinder 0
    mov dh, 0x00 ; head 0

    ; dl = drive number is set as input to disk_load
    ; es:bx = buffer pointer is set as input as well

    int 0x13      ; BIOS interrupt
    jc disk_error ; check carry bit for error

    pop dx     ; get back original number of sectors to read
    cmp al, dh ; BIOS sets 'al' to the # of sectors actually read
               ; compare it to 'dh' and error out if they are ≠
    jne sectors_error
    popa
    ret

disk_error:
    jmp disk_loop

sectors_error:
    jmp disk_loop

disk_loop:
    jmp $
```

이 파일의 주요 부분은 disk_load 프로시저입니다. mbr.asm에서 설정한 입력 파라미터를 기억합니다.

1. 읽은 데이터를 저장할 메모리 위치(bx)
2. 읽을 섹터 수(dh)
3. 읽을 디스크(dl)

모든 프로시저에서 가장 먼저 해야 할 일은 프로시저의 부작용을 피하기 위해 반환하기 전에 모든 범용 레지스터(ax, bx, cx, dx)를 pusha를 사용하여 스택으로 푸시하여 다시 팝할 수 있도록 하는 것입니다.

또한 BIOS 인터럽트 신호를 보내기 전에 dh를 헤드 번호로 설정해야 하고, 완료 시 오류를 감지하기 위해 예상 읽은 섹터 수와 BIOS에서 보고한 실제 섹터 수를 비교하기 위해 읽을 섹터 수(dx 레지스터의 높은 부분에 저장됨)를 스택에 푸시하고 있습니다.

이제 각 레지스터에서 필요한 모든 입력 파라미터를 설정하고 인터럽트를 전송할 수 있습니다. bx와 dl은 호출자에 의해 이미 올바르게 설정되어 있습니다. 목표는 디스크의 다음 섹터를 읽는 것이므로 부팅 섹터 바로 뒤에 있는 섹터 2, 실린더 0, 헤드 0 부터 부팅 드라이브에서 읽습니다.

int 0x13 이 실행되고 나면 커널이 메모리에 로드되어야 합니다. 문제가 없는지 확인하려면 두 가지를 확인해야

합니다: 첫째, 캐리 비트 jc disk_error에 기반한 조건부 점프를 사용해 디스크 에러(캐리 비트로 표시)가 있었는지 여부입니다. 둘째, 읽은 섹터 수(알에서 인터럽트의 반환 값으로 설정)가 비교 명령어 cmp al, dh와 같지 않은 경우 조건부 점프 jne sectors_error를 사용하여 읽으려고 시도한 섹터 수(스택에서 dh로 팝)와 일치하는지 여부입니다.

문제가 발생하면 무한 루프에 빠지게 됩니다. 모든 것이 잘 되었다면 프로시저에서 다시 메인 함수로 돌아갑니다. 다음 작업은 32 비트 보호 모드로 전환할 수 있도록 GDT를 준비하는 것입니다.

- 전역 설명자 테이블(GDT):
16 비트 실수 모드에서 벗어나면 메모리 분할은 약간 다르게 작동합니다. 보호 모드에서 메모리 세그먼트는 GDT의 일부인 세그먼트 설명자에 의해 정의됩니다.

부트 로더의 경우 플랫 메모리 모델과 유사한 가장 간단한 GDT를 설정하겠습니다. 코드와 데이터 세그먼트는 완전히 겹치며 전체 4GB의 주소 지정 가능한 메모리에 걸쳐 있습니다. GDT의 구조는 다음과 같습니다:

1. null 세그먼트 설명자(0 바이트 8 개). 이는 코드가 메모리 세그먼트 선택을 잊어버려서 잘못된 세그먼트가 기본 세그먼트로 생성되는 오류를 잡아내기 위한 안전 장치로 필요합니다.
2. 4GB 코드 세그먼트 디스크립터.
3. 4GB 데이터 세그먼트 디스크립터.

세그먼트 설명자는 다음 정보를 포함하는 데이터 구조입니다:

- 기본 주소: 세그먼트의 32 비트 시작 메모리 주소. 두 세그먼트 모두 0x0 이 됩니다.
- 세그먼트 제한: 세그먼트의 20 비트 길이. 두 세그먼트 모두 0xfffff입니다.
- G(세분성): 설정하면 세그먼트 제한은 4096 바이트 페이지로 계산됩니다. 이 값은 두 세그먼트 모두에 대해 1 이 되며, 0xfffff 페이지 제한은 0xfffff000 바이트 = 4GB로 변환됩니다.
- D(기본 피연산자 크기) / B(큰): 설정하면 32 비트 세그먼트, 그렇지 않으면 16 비트입니다. 두 세그먼트 모두 1 입니다.
- L(긴): 설정된 경우 64 비트 세그먼트입니다(D는 0 이어야 함). 이 경우 32 비트 커널을 작성하고 있으므로 0 입니다.
- AVL(사용 가능): 원하는 용도로 사용할 수 있지만(예: 디버깅) 여기서는 0 으로 설정하겠습니다.
- P (현재): 여기서 0 은 기본적으로 세그먼트를 비활성화하여 다른 사람이 참조할 수 없도록 합니다. 당연히 두 세그먼트 모두 1 이 될 것입니다.
- DPL(설명자 권한 수준): 이 설명자에 액세스하는 데 필요한 보호 링의 권한 수준입니다. 커널이 액세스하기 때문에 두 세그먼트 모두 0 이 될 것입니다.
- 유형: 1 이면 코드 세그먼트 디스크립터입니다. 0 으로 설정하면 데이터 세그먼트입니다. 이 플래그는 코드 세그먼트와 데이터 세그먼트 디스크립터 간에 다른 유일한 플래그입니다. 데이터 세그먼트의 경우 D는 B로 대체되고, C는 E로 대체되며, R은 W로 대체됩니다.
- C(준수): 이 세그먼트의 코드는 권한이 낮은 수준에서 호출될 수 있습니다. 커널 메모리를 보호하기 위해 이 값을 0 으로 설정합니다.
- E(아래로 확장): 데이터 세그먼트가 한계에서 베이스까지 확장되는지 여부입니다. 스택 세그먼트에만 관련되며, 저희의 경우 0 으로 설정했습니다.

- R(읽기 가능): 코드 세그먼트에서 읽을 수 있는지 설정합니다. 그렇지 않으면 실행만 가능합니다. 이 경우 1 로 설정합니다.
- W(쓰기 가능): 데이터 세그먼트에 쓸 수 있는지 설정합니다. 그렇지 않으면 읽기만 가능합니다. 이 경우 1 로 설정합니다.
- A(액세스 가능): 이 플래그는 세그먼트가 액세스될 때 하드웨어에 의해 설정되며 디버깅에 유용할 수 있습니다.

안타깝게도 세그먼트 설명자는 이러한 값을 선형적인 방식으로 포함하지 않고 데이터 구조 전체에 흩어져 있습니다. 따라서 어셈블리에서 GDT를 정의하는 것이 다소 어렵습니다. 데이터 구조의 시각적 표현은 아래 다이어그램을 참조하세요.

31	—	24	23	22	21	20	19	—	16	15	14	13	12	11	10	9	8	7	—	0
Base Address[31:24]			G	D/B	L	AVL	Segment Limit[19:16]			P	DPL		1	Type		C/E	R/W	A	Base Address[23:16]		
Base Address[15:0]										Segment Limit[15:0]										

GDT 자체 외에도 GDT 디스크립터도 설정해야 합니다. 디스크립터에는 GDT 위치(메모리 주소)와 크기가 모두 포함됩니다.

이론은 그만하고 코드를 살펴봅니다. 아래에서 GDT 디스크립터의 정의와 두 개의 세그먼트 디스크립터, 그리고 코드 세그먼트와 데이터 세그먼트의 위치를 알 수 있는 두 개의 어셈블리 상수가 포함된 gdt.asm을 확인할 수 있습니다.

```
;;; gdt_start and gdt_end labels are used to compute size

; null segment descriptor
gdt_start:
    dq 0x0

; code segment descriptor
gdt_code:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10011010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31

; data segment descriptor
gdt_data:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10010010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31

gdt_end:

; GDT descriptor
gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; size (16 bit)
    dd gdt_start ; address (32 bit)

CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

GDT와 GDT 디스크립터가 준비되면 마침내 32 비트 보호 모드로 전환하는 코드를 작성할 수 있습니다.

- 보호 모드로 전환:
32 비트 커널에 제어권을 넘길 수 있도록 32 비트 보호 모드로 전환하려면 다음 단계를 수행해야 합니다:

1. cli 명령어를 사용하여 인터럽트를 비활성화합니다.
2. lgdt 명령어를 사용하여 GDT 기술자를 GDT 레지스터에 로드합니다.
3. 제어 레지스터 cr0 에서 보호 모드를 활성화합니다.
4. jmp를 사용하여 코드 세그먼트로 파 점프합니다. CPU 파이프라인을 플러시하여 미리 가져온 16 비트 명령어를 제거할 수 있도록 멀리 점프해야 합니다.
5. 모든 세그먼트 레지스터(ds, ss, es, fs, gs)가 단일 4GB 데이터 세그먼트를 가리키도록 설정합니다.
6. 32 비트 하단 포인터(ebp)와 스택 포인터(esp)를 설정하여 새 스택을 설정합니다.
7. mbr.asm으로 돌아가서 32 비트 커널 진입 절차를 호출하여 커널에 제어권을 부여합니다.

이제 이를 어셈블리로 변환하여 switch-to-32bit.asm을 작성해 보겠습니다:

```
1 [bits 16]
2 switch_to_32bit:
3     cli                     ; 1. disable interrupts
4     lgdt [gdt_descriptor]   ; 2. load GDT descriptor
5     mov eax, cr0
6     or eax, 0x1             ; 3. enable protected mode
7     mov cr0, eax
8     jmp CODE_SEG:init_32bit ; 4. far jump
9
10 [bits 32]
11 init_32bit:
12     mov ax, DATA_SEG        ; 5. update segment registers
13     mov ds, ax
14     mov ss, ax
15     mov es, ax
16     mov fs, ax
17     mov gs, ax
18
19     mov ebp, 0x90000        ; 6. setup stack
20     mov esp, ebp
21
22     call BEGIN_32BIT        ; 7. move back to mbr.asm
```

모드를 전환하고 나면 커널에 제어권을 넘길 준비가 된 것입니다. 이제 더미 커널을 구현해 보겠습니다.