주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.02.19 – 2024.02.23

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. 부트로더 개발 2. VGA 드라이버 개발

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 부트로더 개발

더미 커널 구현:

- C 커널

기본 부트로더 기능을 실행하고 나면 부트로더에서 호출할 수 있는 작은 더미 커널 함수만 C로 만들면 됩니다. 16 비트 실제 모드에서 벗어나면 더 이상 BIOS를 마음대로 사용할 수 없고 자체 I/O 드라이버를 작성해야 하지만, 이제 C와 같은 고차 언어로 코드를 작성할 수 있습니다! 즉, 더 이상 어셈블리 언어에 의존할 필요가 없습니다.

현재 커널의 임무는 화면 왼쪽 상단에 문자 X를 출력하는 것입니다. 이를 위해서는 비디오 메모리를 직접 수정해야 합니다. VGA 텍스트 모드가 활성화된 컬러 디스플레이의 경우 메모리는 0xb8000 에서 시작됩니다.

각 문자는 2 바이트로 구성됩니다: 첫 번째 바이트는 ASCII로 인코딩된 문자를 나타내고, 두 번째 바이트는 색상 정보를 포함합니다. 아래는 화면 왼쪽 상단에 X를 인쇄하는 kernel.c 내부의 간단한 주요 함수입니다.

```
1 void main() {
2     char* video_memory = (char*) 0xb8000;
3     *video_memory = 'X';
4 }
```

- 커널 진입

mbr.asm을 다시 살펴보면 여전히 C로 작성된 메인 함수를 호출해야 한다는 것을 알 수 있습니다. 이를 위해 부팅 이미지를 만들 때 컴파일된 C 커널 앞의 KERNEL_OFFSET 위치에 배치할 작은 어셈블리 프로그램을 만들 것입니다.

kernel-entry.asm의 내용을 살펴봅시다:

```
1 [bits 32]
2 [extern main]
3 call main
4 jmp $
```

예상대로 여기에는 할 일이 많지 않습니다. 메인 함수만 호출하면 됩니다. 어셈블리 프로세스에서 오류를 방지하려면 메인 함수를 어셈블리 파일 내에 정의되지 않은 외부 프로시저로 선언해야 합니다. 메인 함수를 성공적으로 호출할 수 있도록 메인 함수의 메모리 주소를 확인하는 것은 링커의 임무입니다.

kernel-entry.asm은 mbr.asm에 포함되지 않지만 다음 섹션에서 커널 바이너리의 맨 앞에 배치된다는 점을 기억하는 것이 중요합니다. 이제 우리가 만든 모든 조각을 어떻게 결합할 수 있는지 살펴봅시다.

모든 것을 하나로 모으기

운영 체제 이미지를 구축하려면 약간의 도구가 필요합니다. 어셈블리 파일을 처리하려면 nasm이 필요합니다. C 코드를 컴파일하려면 gcc가 필요합니다. 컴파일된 커널 객체 파일과 컴파일된 커널 항목을 바이너리 파일로

연결하려면 ld가 필요합니다. 그리고 cat을 사용하여 마스터 부트 레코드와 커널 바이너리를 부팅 가능한 단일 바이너리 이미지로 결합할 것입니다.

하지만 이 모든 깔끔한 도구를 어떻게 함께 연결할까요? 다행히도 이를 위한 또 다른 도구인 make가 있습니다. 여기 makefile이 있습니다:

```
# $@ = target file
# $< = first dependency
# $^ = all dependencies

# First rule is the one executed when no parameters are fed to the Makefile
all: run

kernel.bin: kernel-entry.o kernel.o
    ld -m elf_i386 -o $@ -Ttext 0x1000 $^ --oformat binary

kernel-entry.o: kernel-entry.asm
    nasm $< -f elf -o $@

kernel.o: kernel.c
    gcc -m32 -ffreestanding -c $< -o $@

mbr.bin: mbr.asm
    nasm $< -f bin -o $@

os-image.bin: mbr.bin kernel.bin
    cat $^ > $@

run: os-image.bin
    qemu-system-i386 -fda $<

clean:
    $(RM) *.bin *.o *.dis
```

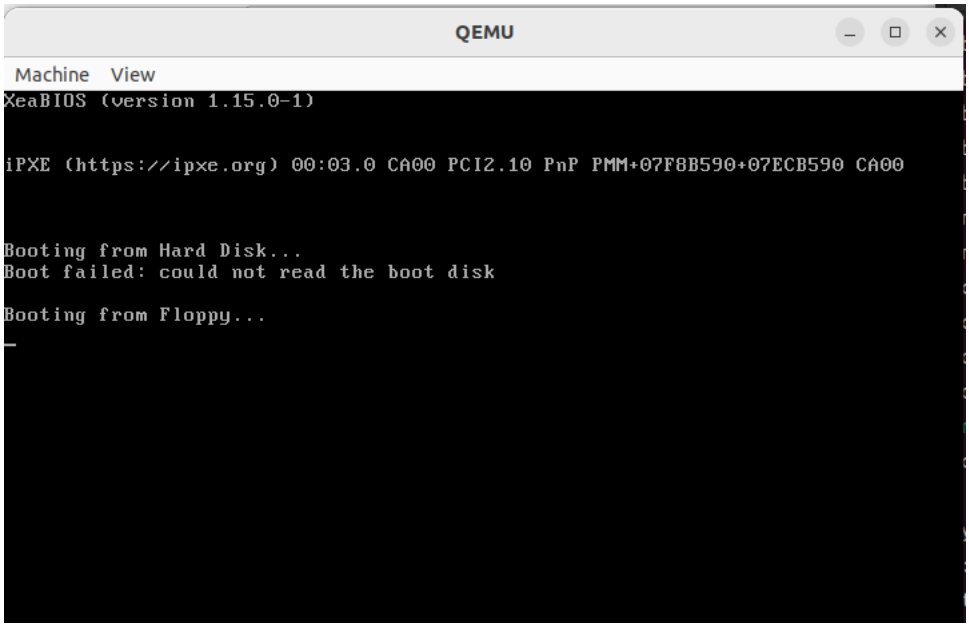
독립형 x86 머신 코드로 컴파일하고 링크하려면 ld와 gcc를 교차 컴파일해야 할 수도 있다는 점에 유의해야합니다. 저는 적어도 Mac에서 이 작업을 수행해야 했습니다.

이제 컴파일, 어셈블리, 링크, 이미지 로딩을 마치고 화면 왼쪽 상단에 있는 X를 살펴봅시다.

M1 mac에서 실행하려고 하였지만 x86 아키텍처를 기반으로 하는 부트로더라서 실행을 할 수 없었습니다.
그래서 Ubuntu 환경에서 재실행 하려고 합니다.

Ubuntu 환경에서 실행하기 위해 Makefile의 gcc 바로 뒤에 -fno-pie를 붙여주고 make 명령어(make, make run)를 통하여 컴파일 하였습니다.

```
# $@ = target file
# $< = first dependency
# $^ = all dependencies

# First rule is the one executed when no parameters are fed to the Makefile
all: run

kernel.bin: kernel-entry.o kernel.o
	ld -m elf_i386 -o $@ -Ttext 0x1000 $^ --oformat binary

kernel-entry.o: kernel-entry.asm
	nasm $< -f elf -o $@

kernel.o: kernel.c
	gcc -fno-pie -m32 -ffreestanding -c $< -o $@

mbr.bin: mbr.asm
	nasm $< -f bin -o $@

os-image.bin: mbr.bin kernel.bin
	cat $^ > $@

run: os-image.bin
	qemu-system-i386 -fda $<

clean:
	$(RM) *.bin *.o *.dis
```



2. VGA 드라이버 개발

왜 VGA 드라이버인가?

운영 체제는 사용자와 상호 작용할 수 있는 방법이 필요합니다. 이를 위해서는 어떤 형태의 입출력이 필요합니다. 먼저 시각적 출력에 초점을 맞추고자 합니다. 다루기 매우 편리하고 기본 터미널 기능에 충분히 유연하기 때문에 VGA 80x25 텍스트 비디오 모드를 사용할 것입니다. 이 모드는 커널을 부팅하는 동안 BIOS에서 이미 사용했던 모드와 동일합니다.

VGA를 사용하면 비디오 메모리라고 하는 전용 메모리 영역을 직접 수정하여 화면 출력을 생성할 수 있습니다. 그 외에도 포트 I/O CPU 명령어 입출력을 사용하여 장치 포트와 상호 작용하는 데 사용할 수 있는 특정 포트 주소가 있습니다. 이는 모든 I/O 포트(VGA 포트 포함)가 특정 메모리 위치에 매핑되어 있기 때문에 가능합니다.

VGA 드라이버의 임무는 이러한 하위 레벨 메모리 조작을 상위 레벨 함수 내에 캡슐화하는 것입니다. 개별 CPU 명령어를 실행하고 메모리 주소를 수정하는 대신 화면에 문자열을 인쇄하거나 모든 출력을 지우는 함수를 호출할 수 있기를 원합니다. 이러한 최소한의 VGA 드라이버를 작성해 보겠습니다.

방법을 설명한 후, 이 지식을 활용하여 텍스트 커서 위치를 검색하고 설정하는 함수를 구현합니다. 그런 다음 비디오 메모리에 글을 써서 화면에 개별 문자를 인쇄하는 코드를 작성하겠습니다. 커서 조작과 문자 인쇄를 결합하여 화면에 문자열을 인쇄하는 기능을 제공할 것입니다. 그 이후의 섹션에서는 줄 바꿈 문자 처리, 스크롤, 화면 지우기 등 몇 가지 확장 기능에 중점을 둡니다. 마지막 섹션에서는 새로 작성한 드라이버를 사용할 수 있도록 기본 커널 기능을 조정합니다.

C에서 I/O 포트와 인터페이스하기

I/O 드라이버에서 중요한 부분 중 하나는 포트를 통해 I/O 장치와 인터페이스 하는 기능입니다. VGA 드라이버에서는 텍스트 모드에서 커서 위치를 읽고 설정하기 위해 현재 0x3d4 및 0x3d5 포트에만 액세스하면 됩니다.

앞서 언급했듯이 인풋과 아웃 인스트럭션을 활용해 각각 포트 데이터를 읽고 쓸 수 있습니다. 하지만 C 내에서 이러한 명령어를 어떻게 활용할 수 있을까요?

다행히도 C 컴파일러는 __asm__ 함수를 호출하여 어셈블러 코드를 작성하고, C 변수를 입력으로 전달하고, 결과를 다시 C 변수에 쓰는 인라인 어셈블러 코드를 지원합니다. 어셈블러 명령어, 출력 매개변수, __asm__ 함수의 입력 매개변수는 :로 구분됩니다. 명령어 피연산자의 순서가 뒤바뀌는 등 NASM과 구문이 약간 다릅니다.

다음 두 함수를 통해 지정된 포트에서 데이터를 읽고 쓰는 방법을 살펴보겠습니다.

```
unsigned char port_byte_in(unsigned short port) {
    unsigned char result;
    __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
    return result;
}

void port_byte_out(unsigned short port, unsigned char data) {
    __asm__("out %%al, %%dx" : : "a" (data), "d" (port));
}
```

port_byte_in 함수의 경우 C 변수 포트를 dx 레지스터에 매핑하고, al, dx에서 실행한 다음, al 레지스터의 값을 C 변수 결과에 저장합니다. port_byte_out 함수도 비슷합니다. 포트는 dx, al로, 데이터는 al로 매핑하여 실행합니다. 데이터만 쓰기 때문에 출력 파라미터가 없고 함수에는 반환값이 없습니다.

커서 위치 가져오기 및 설정하기

새로 작성한 포트 I/O 함수를 통해 VGA 텍스트 모드 커서와 상호작용할 준비가 되었습니다. 커서 위치를 읽거나 변경하려면 VGA 제어 레지스터 0x3d4 를 수정하고 해당 데이터 레지스터 0x3d5 에서 읽거나 써야 합니다.

16 비트 커서 위치는 2 개의 개별 바이트, 즉 하이 바이트와 로우 바이트로 인코딩됩니다. 제어 레지스터가 0x0f로 설정된 경우 데이터 레지스터는 로우 바이트를, 0x0e 값을 사용하는 경우 하이 바이트를 보유하게 됩니다. 먼저 레지스터 주소와 오프셋 코드를 C 상수로 정의하겠습니다.

```
#define VGA_CTRL_REGISTER 0x3d4
#define VGA_DATA_REGISTER 0x3d5
#define VGA_OFFSET_LOW 0x0f
#define VGA_OFFSET_HIGH 0x0e
```

커서 오프셋을 비디오 메모리 오프셋으로 표현하겠습니다. 텍스트 그리드의 각 위치는 문자 및 색상 정보에 각각 2 바이트로 표시되므로 메모리 오프셋은 커서 오프셋의 두 배입니다.

16 비트 커서 오프셋의 두 배 크기인 메모리 오프셋을 16 비트에 맞출 수 없으므로 32 비트 정수를 사용합니다. 이제 내부 커서 오프셋을 가져오는 set_cursor와 get_cursor 함수를 작성할 수 있습니다.

```
void set_cursor(int offset) {
    offset /= 2;
    port_byte_out(VGA_CTRL_REGISTER, VGA_OFFSET_HIGH);
    port_byte_out(VGA_DATA_REGISTER, (unsigned char) (offset >> 8));
    port_byte_out(VGA_CTRL_REGISTER, VGA_OFFSET_LOW);
    port_byte_out(VGA_DATA_REGISTER, (unsigned char) (offset & 0xff));
}

int get_cursor() {
    port_byte_out(VGA_CTRL_REGISTER, VGA_OFFSET_HIGH);
    int offset = port_byte_in(VGA_DATA_REGISTER) << 8;
    port_byte_out(VGA_CTRL_REGISTER, VGA_OFFSET_LOW);
    offset += port_byte_in(VGA_DATA_REGISTER);
    return offset * 2;
}
```

메모리 오프셋은 커서 오프셋의 두 배이므로 두 오프셋을 곱하거나 2 로 나누어 매핑해야 합니다. 또한 정수에서 높은 바이트와 낮은 바이트를 가져오기 위해 약간의 비트 시프트/마스킹을 수행해야 합니다.

화면에 문자 인쇄하기

커서 조작이 완료되었으므로 화면의 지정된 위치에 문자를 인쇄할 수 있어야 합니다. 이전 포스트의 더미 커널에서 이미 이 작업을 수행했습니다. 이제 그 코드를 가져와서 좀 더 일반화해 보겠습니다. 먼저 비디오 메모리의 시작 주소, 텍스트

그리드 크기, 캐릭터에 사용할 기본 색 구성표를 포함하는 몇 가지 유용한 상수를 정의하겠습니다.

```
#define VIDEO_ADDRESS 0xb8000
#define MAX_ROWS 25
#define MAX_COLS 80
#define WHITE_ON_BLACK 0x0f
```

다음으로, 주어진 메모리 오프셋에서 비디오 메모리에 문자를 기록하여 화면에 문자를 인쇄하는 함수를 작성해 보겠습니다. 지금은 다른 색상을 지원하지 않지만 나중에 필요한 경우 조정할 수 있습니다.

```
void set_char_at_video_memory(char character, int offset) {
    unsigned char *vidmem = (unsigned char *) VIDEO_ADDRESS;
    vidmem[offset] = character;
    vidmem[offset + 1] = WHITE_ON_BLACK;
}
```

이제 화면에 문자를 인쇄하고 커서를 수정할 수 있으므로 문자열을 인쇄하고 그에 따라 커서를 이동하는 함수를 구현할 수 있습니다.

텍스트 인쇄 및 커서 이동

C에서 문자열은 ASCII로 인코딩된 바이트의 0 바이트 종료 시퀀스입니다. 화면에 문자열을 인쇄하려면 다음을 수행해야 합니다:

1. 현재 커서 오프셋을 가져옵니다.
2. 문자열의 바이트를 반복하여 비디오 메모리에 쓰면서 오프셋을 증가시킵니다.
3. 커서 위치를 업데이트합니다.

다음은 코드입니다:

```
void print_string(char *string) {
    int offset = get_cursor();
    int i = 0;
    while (string[i] != 0) {
        set_char_at_video_memory(string[i], offset);
        i++;
        offset += 2;
    }
    set_cursor(offset);
}
```

이 코드는 현재 줄 바꿈 문자나 범위를 벗어난 오프셋을 처리하지 않습니다. 오프셋이 범위를 벗어나는 경우 스크롤 기능을 구현하고 줄 바꿈 문자를 감지하면 커서를 다음 줄로 이동하면 이 문제를 해결할 수 있습니다.

다음에는 줄 바꿈 문자를 처리하는 방법을 살펴보겠습니다.

개행 문자 처리하기

개행 문자는 실제로 인쇄할 수 없는 문자입니다. 그리드에서 공백을 차지하지 않고 커서를 다음 줄로 이동시킵니다. 이를 위해 주어진 커서 오프셋을 취하고 다음 행의 첫 번째 열인 새 오프셋을 계산하는 함수를 작성하겠습니다.

이를 구현하기 전에 두 개의 작은 도우미 함수를 작성하겠습니다. get_row_from_offset은 메모리 오프셋을 가져와 해당 셀의 행 번호를 반환하고, get_offset은 주어진 셀의 메모리 오프셋을 반환합니다.

```
int get_row_from_offset(int offset) {
    return offset / (2 * MAX_COLS);
}

int get_offset(int col, int row) {
    return 2 * (row * MAX_COLS + col);
}
```

이 두 함수를 결합하면 오프셋을 다음 줄로 이동하는 함수를 쉽게 작성할 수 있습니다.

```
int move_offset_to_new_line(int offset) {
    return get_offset(0, get_row_from_offset(offset) + 1);
}
```

이 함수를 마음대로 사용하면 ₩n을 처리하도록 print_string 함수를 수정할 수 있습니다.

```
void print_string(char *string) {
    int offset = get_cursor();
    int i = 0;
    while (string[i] != 0) {
        if (string[i] == '\n') {
            offset = move_offset_to_new_line(offset);
        } else {
            set_char_at_video_memory(string[i], offset);
            offset += 2;
        }
        i++;
    }
    set_cursor(offset);
}
```

다음으로 스크롤을 구현하는 방법을 살펴보겠습니다.

스크롤

커서 오프셋이 최대 값인 25x80x2 = 4000을 초과하면 터미널 출력은 아래로 스크롤됩니다. 스크롤 버퍼가 없으면 맨 위 줄이 손실되지만 지금은 괜찮습니다. 다음 단계를 실행하여 스크롤을 구현할 수 있습니다:

1. 첫 번째 행을 제외한 모든 행을 한 줄씩 위로 이동합니다. 맨 위 행은 어차피 범위를 벗어날 것이므로 이동할 필요가 없습니다.
2. 마지막 행을 공백으로 채웁니다.
3. 다시 그리드 경계 안에 들어오도록 오프셋을 수정합니다.

비디오 메모리의 청크를 복사하여 행 이동을 구현할 수 있습니다. 먼저 메모리에서 주어진 바이트 수만큼 *source에서 *dest로 복사하는 함수를 작성합니다.

```
void memory_copy(char *source, char *dest, int nbytes) {
    int i;
    for (i = 0; i < nbytes; i++) {
        *(dest + i) = *(source + i);
    }
}
```

memory_copy 함수를 사용하면 주어진 오프셋을 가져와 원하는 메모리 영역을 복사하고 마지막 행을 지운 다음 다시 그리드 경계 안에 오도록 오프셋을 조정하는 스크롤 도우미 함수를 구현할 수 있습니다. 주어진 셀의 오프셋을 편리하게 결정하기 위해 get_offset 헬퍼 메서드를 사용하겠습니다.

```
int scroll_ln(int offset) {
    memory_copy(
            (char *) (get_offset(0, 1) + VIDEO_ADDRESS),
            (char *) (get_offset(0, 0) + VIDEO_ADDRESS),
            MAX_COLS * (MAX_ROWS - 1) * 2
    );

    for (int col = 0; col < MAX_COLS; col++) {
        set_char_at_video_memory(' ', get_offset(col, MAX_ROWS - 1));
    }

    return offset - 2 * MAX_COLS;
}
```

이제 각 루프 반복마다 현재 오프셋이 최대 값을 초과하는지 확인하고 필요한 경우 스크롤하도록 print_string 함수를 수정하기만 하면 됩니다. 이것이 함수의 최종 버전입니다:

```
void print_string(char *string) {
    int offset = get_cursor();
    int i = 0;
    while (string[i] != 0) {
        if (offset >= MAX_ROWS * MAX_COLS * 2) {
            offset = scroll_ln(offset);
        }
        if (string[i] == '\n') {
            offset = move_offset_to_new_line(offset);
        } else {
            set_char_at_video_memory(string[i], offset);
            offset += 2;
        }
        i++;
    }
    set_cursor(offset);
}
```

화면 지우기

커널이 시작된 후 비디오 메모리는 더 이상 관련이 없는 BIOS의 일부 정보로 채워집니다. 따라서 화면을 지울 방법이 필요합니다. 다행히도 이 기능은 기존 도우미 기능을 사용하면 쉽게 구현할 수 있습니다.

```
void clear_screen() {
    for (int i = 0; i < MAX_COLS * MAX_ROWS; ++i) {
        set_char_at_video_memory(' ', i * 2);
    }
    set_cursor(get_offset(0, 0));
}
```

Hello World와 스크롤 기능

이제 문자열을 인쇄하도록 메인 함수를 조정할 수 있습니다! 컴파일러가 드라이버 함수가 존재한다는 것을 알 수 있도록 디스플레이 헤더 파일만 포함하면 됩니다.

```
#include "../drivers/display.h"

void main() {
    clear_screen();
    print_string("Hello World!\n");
}
```

스크롤을 시각화하기 위해 증가하는 문자를 인쇄하는 확장된 메인 함수를 작성하고, 디버그 모드에서 QEMU를 실행하고, GNU 디버거(gdb)를 연결한 다음 인쇄 함수에 중단점을 넣고 다음 디버그 명령을 실행하여 스크롤을 느리게 하여 표시되도록 했습니다.

```
while (1)
shell sleep 0.2
continue
end
```

다음은 위의 내용을 단계별로 수행하는 방법입니다:

1. 먼저, 메인 함수를 수정하여 증가하는 문자를 출력하도록 합니다. 이렇게 하면 스크롤링이 발생하는지 확인할 수 있습니다.

2. 그런 다음 QEMU를 디버그 모드로 실행합니다. `Makefile`에서 제공한 `debug` 타겟을 사용할 수 있습니다. 예를 들어, 다음과 같이 입력할 수 있습니다:

```
make debug
```

이 명령을 실행하면 QEMU가 디버그 모드로 실행되고, gdb가 자동으로 연결됩니다.

3. gdb에서 `print` 함수에 중단점을 설정합니다. 예를 들어, `print` 함수가 'print_string`이라면, 다음과 같이 입력할 수 있습니다:

```
break print_string
```

이 명령을 실행하면 `print_ string` 함수에서 실행이 중단되고, gdb가 제어권을 다시 얻습니다.

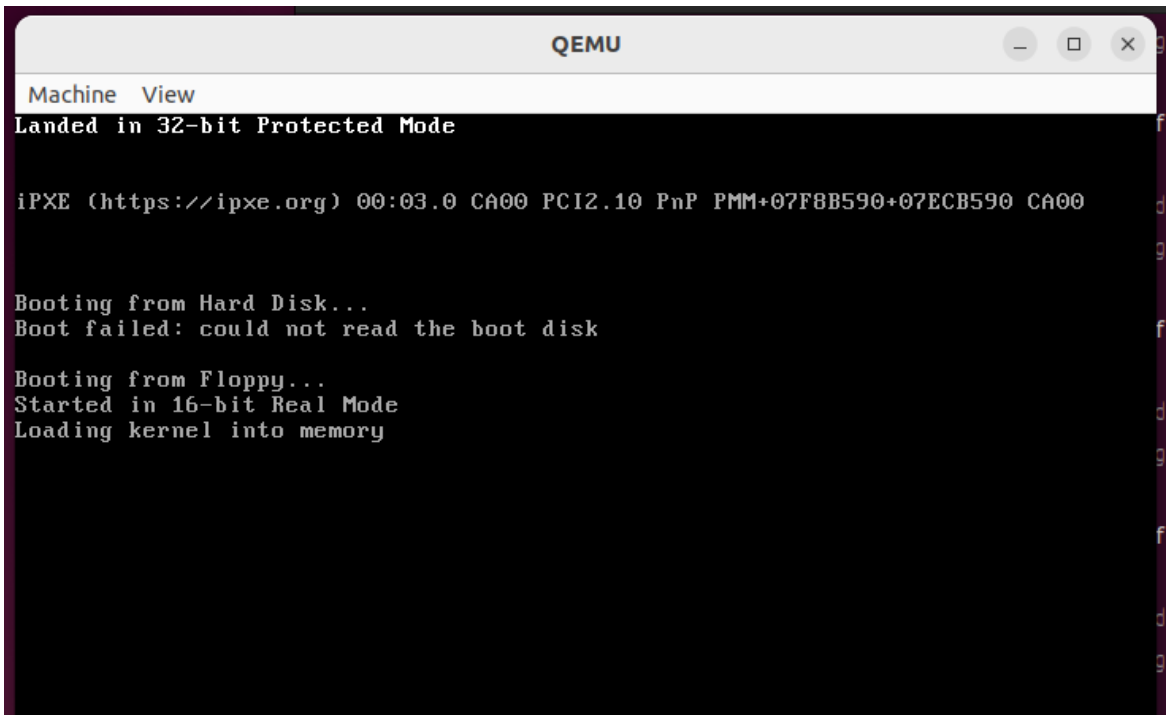
4. 마지막으로, gdb에서 다음의 디버그 명령을 실행합니다:

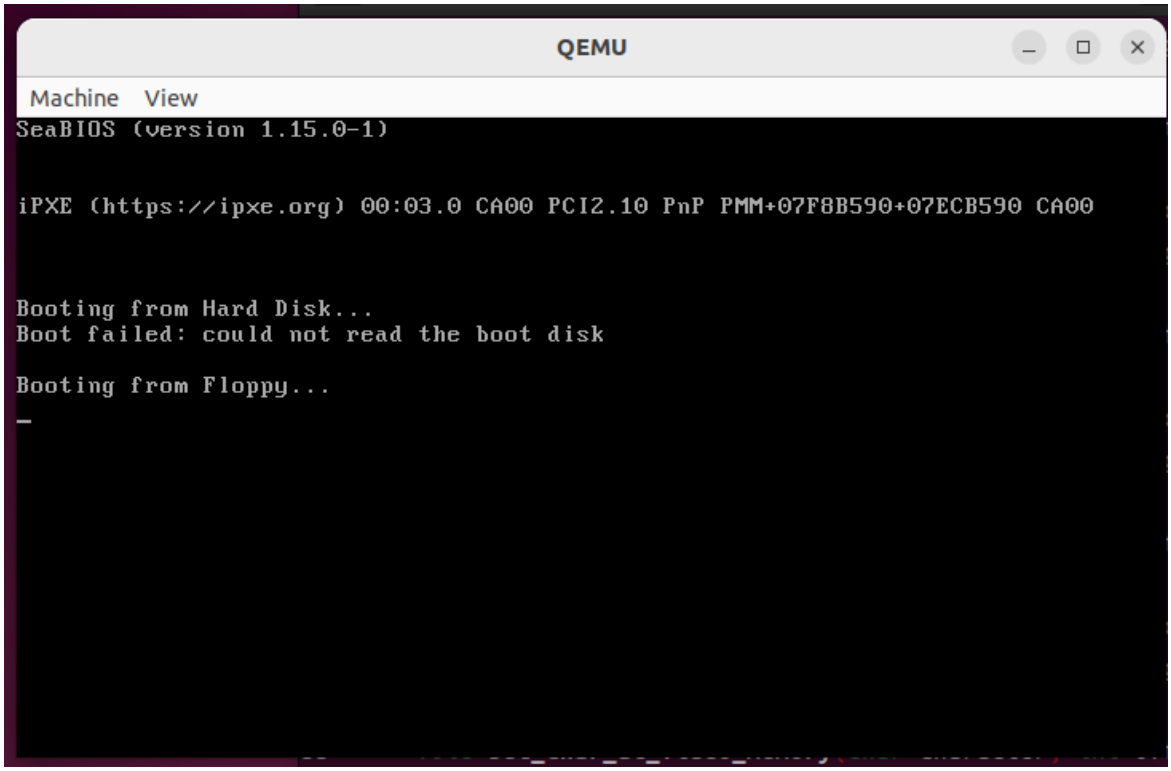
```
while (1)
shell sleep 0.2
continue
end
```

이 명령은 무한 루프를 실행하면서 매번 0.2 초 동안 일시 정지하고, 그 후에 실행을 계속합니다. 이렇게 하면 스크롤링이 천천히 진행되어 시각적으로 확인할 수 있게 됩니다.

위의 단계를 수행하면, QEMU에서 실행되는 운영체제 커널의 스크롤링을 시각적으로 확인할 수 있습니다.

결과는 다음과 같습니다. 아래 2 개 화면이 교차로 출력됩니다.





make를 하려는데 계속 오류가 발생해 처음에 힘들었었습니다.
그 이유는 i686-elf-tools (https://github.com/lordmilko/i686-elf-tools)라는 운영체제 구축 도구를 설치를 안하였기 때문이었습니다. 설치를 하려는데 디스크 용량 부족 문제와 가상 머신의 고정 ip를 해놓아서 네트워크 문제, 버전 문제 등이 발생하여 우분투를 삭제한 뒤 디스크 용량을 확보하여 설치 후 컴파일 하였습니다.

위의 코드는 전체 코드를 나타내는 것은 아닙니다. 하지만 핵심 코드들을 알 수 있으며, 전체 코드는 이 글을 작성한 개발자의 Github를 참고하여 공부하였습니다. 어려운 개념이라 시간 내(4 일)에 완성을 하지 못할 것 같아서 위와 같이 핵심 코드들만 살펴보았습니다.

3. 키보드 드라이버 개발

개요

위에서는 화면에 텍스트를 인쇄할 수 있도록 비디오 드라이버를 구현했습니다. 그러나 운영 체제가 사용자에게 유용하려면 사용자가 명령을 입력할 수 있어야 합니다. 텍스트 입력과 출력은 향후 셸 기능의 기반이 될 것입니다.

그렇다면 키보드와 운영 체제 간의 통신은 어떻게 이루어질까요? 키보드는 물리적 포트(예: 직렬, PS/2, USB)를 통해 컴퓨터에 연결됩니다. PS/2 의 경우 마더보드에 있는 마이크로컨트롤러가 데이터를 수신합니다. 키를 누르면 마이크로컨트롤러는 I/O 포트 0x60 에 관련 정보를 저장하고 인터럽트 요청 IRQ 1 을 프로그래밍 가능 인터럽트 컨트롤러(PIC)로 보냅니다.

그러면 PIC는 외부 IRQ에 따라 미리 정의된 인터럽트 번호로 CPU를 인터럽트합니다. 인터럽트를 수신하면 CPU는 인터럽트 설명자 테이블(IDT)을 참조하여 호출해야 할 각 인터럽트 처리기를 찾습니다. 핸들러가 작업을 완료하면 CPU는 인터럽트 이전부터 일반 실행을 재개합니다.

완전한 체인이 작동하려면 커널 초기화 중에 몇 가지 준비를 해야 합니다. 먼저, IRQ가 실제 인터럽트로 올바르게 변환되도록 PIC 내부에 올바른 매핑을 설정해야 합니다. 그런 다음 키보드 핸들러에 대한 참조가 포함된 유효한 IDT를 생성하고 로드해야 합니다. 그런 다음 핸들러는 각 I/O 포트에서 모든 관련 데이터를 읽고 사용자에게 표시할 수 있는 텍스트(예: LCTRL 또는 A)로 변환합니다.

아래는 다음과 같이 구성되어 있습니다. 다음 섹션에서는 IDT를 정의하고 로드하는 데 중점을 둡니다. 그 후에는 키보드 인터럽트 핸들러를 구현하고 등록하겠습니다. 마지막으로 새로 작성한 코드를 올바른 순서로 실행하기 위해 커널 기능을 확장합니다.

아래의 코드 예제에서는 원래 C 타입보다 조금 더 구조화된 #include <stdint.h>의 타입 별칭을 사용합니다. 예를 들어 uint16_t는 부호 없는 2 바이트(16 비트) 값에 해당합니다.

IDT 설정

- IDT 구조

IDT는 게이트라고 하는 256 개의 설명자 항목으로 구성됩니다. 각 게이트의 길이는 8 바이트이며 테이블 내 위치에서 결정되는 정확히 하나의 인터럽트 번호에 해당합니다. 게이트에는 작업 게이트, 인터럽트 게이트, 트랩 게이트의 세 가지 유형이 있습니다. 인터럽트 게이트와 트랩 게이트는 사용자 정의 핸들러 함수를 호출할 수 있으며, 인터럽트 게이트는 핸들러 호출 중에 하드웨어 인터럽트 처리를 일시적으로 비활성화하여 하드웨어 인터럽트를 처리하는 데 유용합니다. 태스크 게이트를 사용하면 하드웨어 태스크 전환 메커니즘을 사용하여 프로세서의 제어권을 다른 프로그램으로 넘길 수 있습니다.

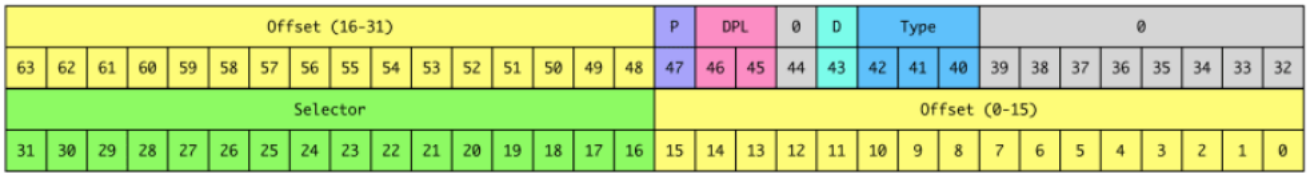
지금은 인터럽트 게이트만 정의하면 됩니다. 인터럽트 게이트에는 다음 정보가 포함됩니다:

- 오프셋. 32 비트 오프셋은 각 코드 세그먼트 내의 인터럽트 핸들러의 메모리 주소를 나타냅니다.
- 셀렉터. 핸들러를 호출할 때 점프할 코드 세그먼트의 16 비트 선택기입니다. 커널 코드 세그먼트가 됩니다.

- 유형. 게이트 유형을 나타내는 3 비트입니다. 인터럽트 게이트를 정의하고 있으므로 110 으로 설정됩니다.
- D. 코드 세그먼트가 32 비트인지 여부를 나타내는 1 비트입니다. 1 로 설정됩니다.
- DPL. 2 비트 설명자 권한 수준은 핸들러를 호출하는 데 필요한 권한을 나타냅니다. 00 으로 설정됩니다.
- P. 게이트가 활성 상태인지 여부를 나타내는 1 비트입니다. 1 로 설정됩니다.
- 0. 인터럽트 게이트의 경우 항상 0 으로 설정해야 하는 일부 비트입니다.

아래 다이어그램은 IDT 게이트의 레이아웃을 보여줍니다.

IDT Gate Layout



C에서 IDT 게이트를 생성하려면 먼저 idt_gate_t 구조체 유형을 정의합니다. __attribute__((packed))는 구조체 내부의 데이터를 정의된 만큼 꽉 채우도록 gcc에 지시합니다. 그렇지 않으면 컴파일러는 예를 들어 CPU 캐시 크기와 관련하여 구조체 레이아웃을 최적화하기 위해 패딩을 포함할 수 있습니다.

```
typedef struct {
    uint16_t low_offset;
    uint16_t selector;
    uint8_t always0;
    uint8_t flags;
    uint16_t high_offset;
} __attribute__((packed)) idt_gate_t;
```

이제 IDT를 256 개의 게이트 배열로 정의하고 인터럽트 n에 대한 핸들러를 등록하는 세터 함수 set_idt_gate를 구현할 수 있습니다. 핸들러의 32 비트 메모리 주소를 분할하기 위해 두 개의 작은 헬퍼 함수를 사용할 것입니다.

```
#define low_16(address) (uint16_t)((address) & 0xFFFF)
#define high_16(address) (uint16_t)(((address) >> 16) & 0xFFFF)

idt_gate_t idt[256];

void set_idt_gate(int n, uint32_t handler) {
    idt[n].low_offset = low_16(handler);
    idt[n].selector = 0x08; // see GDT
    idt[n].always0 = 0;
    // 0x8E = 1  00 0 1  110
    //        P DPL 0 D Type
    idt[n].flags = 0x8E;
    idt[n].high_offset = high_16(handler);
}
```

- 내부 ISR 설정하기

인터럽트 핸들러는 인터럽트 서비스 루틴(ISR)이라고도 합니다. 처음 32 개의 ISR은 예외 및 오류와 같은 CPU 특정 인터럽트를 위해 예약되어 있습니다. 이러한 설정은 나중에 PIC를 다시 매핑하고 IRQ를 정의할 때 문제가 있는지 알 수 있는 유일한 방법이기 때문에 매우 중요합니다. 전체 목록은 소스 코드나 위키피디아에서 찾을 수 있습니다.

먼저, 인터럽트와 관련된 모든 필요한 정보를 추출하고 그에 따라 작동할 수 있는 일반 ISR 핸들러 함수를 C로 정의합니다. 지금은 각 인터럽트 번호에 대한 문자열 표현을 포함하는 간단한 조회 배열을 만들겠습니다.

```
char *exception_messages[] = {
    "Division by zero",
    "Debug",
    \\ ...
    "Reserved"
};

void isr_handler(registers_t *r) {
    print_string(exception_messages[r->int_no]);
    print_nl();
}
```

모든 정보를 사용할 수 있도록 하기 위해 다음과 같이 정의된 함수에 레지스터_t 타입의 구조체를 전달합니다:

```
typedef struct {
    // data segment selector
    uint32_t ds;
    // general purpose registers pushed by pusha
    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    // pushed by isr procedure
    uint32_t int_no, err_code;
    // pushed by CPU automatically
    uint32_t eip, cs, eflags, useresp, ss;
} registers_t;
```

이 구조가 매우 복잡한 이유는 어셈블리 내에서 핸들러 함수(C로 작성됨)를 호출하기 때문입니다. 함수가 호출되기 전에 C는 인수가 스택에 존재할 것으로 예상합니다. 스택에는 이미 일부 정보가 포함되어 있으며 추가 정보로 스택을 확장하고 있습니다.

아래는 처음 32 개의 ISR을 정의하는 어셈블리 코드의 발췌본입니다. 안타깝게도 핸들러를 호출하는 데 사용된 게이트를 알 수 있는 방법이 없으므로 각 게이트마다 하나의 핸들러가 필요합니다. 나중에 C 코드에서 참조할 수

있도록 레이블을 전역으로 정의해야 합니다.

```
global isr0
global isr1
; ...
global isr31

; 0: Divide By Zero Exception
isr0:
    push byte 0
    push byte 0
    jmp isr_common_stub

; 1: Debug Exception
isr1:
    push byte 0
    push byte 1
    jmp isr_common_stub

; ...

; 12: Stack Fault Exception
isr12:
    ; error info pushed by CPU
    push byte 12
    jmp isr_common_stub

; ...

; 31: Reserved
isr31:
    push byte 0
    push byte 31
    jmp isr_common_stub
```

각 프로시저는 잠시 후에 살펴볼 공통 ISR 프로시저로 넘기기 전에 int_no와 err_code가 스택에 있는지 확인합니다. 첫 번째 푸시(err_code)가 있는 경우 스택 오류와 같은 특정 예외와 관련된 오류 정보를 나타냅니다. 이러한 예외가 발생하면 CPU는 이 오류 정보를 스택에 푸시합니다. 모든 ISR에 대해 일관된 스택을 유지하기 위해 오류 정보를 사용할 수 없는 경우에는 0 바이트를 푸시합니다. 두 번째 푸시는 인터럽트 번호에 해당합니다.

이제 일반적인 ISR 절차를 살펴봅시다. 이 절차는 스택을 레지스터_t에 필요한 모든 정보로 채우고, 세그먼트 포인터를 준비하여 커널 ISR 핸들러인 isr_handler를 호출하고, 스택 포인터(실제로는 레지스터_t에 대한 포인터)를 스택으로 푸시하고, isr_handler를 호출하고, CPU가 중단된 곳에서 다시 시작할 수 있도록 나중에 정리하는 순서입니다. isr_handler는 C에 정의되므로 외부로 표시해야 합니다.

```
[extern isr_handler]

isr_common_stub:
    ; push general purpose registers
    pusha

    ; push data segment selector
    mov ax, ds
    push eax

    ; use kernel data segment
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    ; hand over stack to C function
    push esp
    ; and call it
    call isr_handler
    ; pop stack pointer again
    pop eax

    ; restore original segment pointers segment
    pop eax
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    ; restore registers
    popa

    ; remove int_no and err_code from stack
    add esp, 8

    ; pops cs, eip, eflags, ss, and esp
    ; https://www.felixcloutier.com/x86/iret:iretd
    iret
```

마지막으로, 앞서 설명한 set_idt_gate 함수를 사용하여 IDT에 처음 32 개의 ISR을 등록할 수 있습니다. 모든 호출을 isr_install 안에 래핑합니다.

```
void isr_install() {
    set_idt_gate(0, (uint32_t) isr0);
    set_idt_gate(1, (uint32_t) isr1);
    // ...
    set_idt_gate(31, (uint32_t) isr31);
}
```

이제 CPU 내부 인터럽트 핸들러가 준비되었으므로 이제 PIC를 다시 매핑하고 IRQ 핸들러를 설정할 수 있습니다.

- PIC 리매핑
x86 시스템에서 8259 PIC는 하드웨어 인터럽트 관리를 담당합니다. 최신 컴퓨터에는 업데이트된 표준인 APIC(고급 프로그래밍 가능 인터럽트 컨트롤러)가 존재하지만 이 글의 범위를 벗어난다는 점에 유의하세요. 여기서는 두 개의 PIC로 구성된 캐스케이드를 활용하며, 각 PIC는 8 개의 서로 다른 IRQ를 처리할 수 있습니다. 보조 칩은 IRQ를 통해 기본 칩에 연결되므로 효과적으로 15 개의 서로 다른 IRQ를 처리할 수 있습니다.

BIOS는 16 비트 리얼 모드에서 처음 8 개의 IRQ가 IDT의 처음 8 개의 게이트에 매핑되는 합리적인 기본값으로 PIC를 프로그래밍합니다. 그러나 보호 모드에서는 이러한 값이 CPU 내부 인터럽트용으로 예약된 처음 32 개의 게이트와 충돌합니다. 따라서 충돌을 피하려면 PIC를 다시 프로그래밍(리매핑)해야 합니다.

PIC 프로그래밍은 각 I/O 포트에 액세스하여 수행할 수 있습니다. 기본 PIC는 0x20(명령) 및 0x21(데이터) 포트를 사용합니다. 보조 PIC는 0xA0(명령) 및 0xA1(데이터) 포트를 사용합니다. 프로그래밍은 4 개의 초기화 명령어(ICW)를 전송하는 방식으로 이루어집니다. 다음 단락이 혼란스럽다면 이 포괄적인 설명서를 읽어보시기 바랍니다.

먼저 초기화 명령어 ICW1(0x11)을 두 PIC에 모두 전송해야 합니다. 그런 다음 데이터 포트에서 다음 세 가지 입력을 기다립니다:

- ICW2(IDT 오프셋). 1 차 PIC의 경우 0x20(32), 2 차 PIC의 경우 0x28(40)로 설정됩니다.
- ICW3(PIC 간 배선). IRQ 2(0x04, 0b00000100)에서 보조 PIC의 IRQ를 수신하도록 1 차 PIC에 지시합니다. 0x02 = 0b00000010 으로 설정하여 보조 PIC를 보조로 표시합니다.
- ICW4(모드). 8086 모드를 활성화하기 위해 0x01 = 0b00000001 로 설정합니다.

마지막으로 첫 번째 연산 명령어(OCW1) 0x00 = 0b00000000 을 전송하여 모든 IRQ를 활성화합니다(마스킹 없음). 이전 포스트의 port_byte_out 함수를 사용하면 다음과 같이 isr_install을 확장하여 PIC 리매핑을 수행할 수 있습니다.

```
void isr_install() {
    // internal ISRs
    // ...

    // ICW1
    port_byte_out(0x20, 0x11);
    port_byte_out(0xA0, 0x11);

    // ICW2
    port_byte_out(0x21, 0x20);
    port_byte_out(0xA1, 0x28);

    // ICW3
    port_byte_out(0x21, 0x04);
    port_byte_out(0xA1, 0x02);

    // ICW4
    port_byte_out(0x21, 0x01);
    port_byte_out(0xA1, 0x01);

    // OCW1
    port_byte_out(0x21, 0x0);
    port_byte_out(0xA1, 0x0);
}
```

이제 인터럽트 게이트 32-47 로 IRQ를 전송하도록 PIC를 성공적으로 리매핑했으므로 각 ISR을 등록할 수 있습니다.

IRQ 핸들러 설정

IRQ를 처리하기 위해 ISR을 추가하는 것은 우리가 처음 만든 32 개의 CPU 내부 ISR과 매우 유사합니다. 먼저 IRQ 0-15 에 대한 게이트를 추가하여 IDT를 확장합니다.

```
void isr_install() {
    // internal ISRs
    // ...

    // PIC remapping
    // ...

    // IRQ ISRs (primary PIC)
    set_idt_gate(32, (uint32_t)irq0);
    // ...
    set_idt_gate(39, (uint32_t)irq7);

    // IRQ ISRs (secondary PIC)
    set_idt_gate(40, (uint32_t)irq8);
    // ...
    set_idt_gate(47, (uint32_t)irq15);
}
```

그런 다음 어셈블러 코드에 IRQ 프로시저 레이블을 추가합니다. IRQ 번호와 인터럽트 번호를 스택에 푸시한 다음 irq_common_stub을 호출합니다.

```
global irq0
; ...
global irq15

irq0:
    push byte 0
    push byte 32
    jmp irq_common_stub

; ...

irq15:
    push byte 15
    push byte 47
    jmp irq_common_stub
```

irq_common_stub은 isr_common_stub과 유사하게 정의되며, C에서 irq_handler 함수를 호출합니다. 하지만 키보드 핸들러와 같이 커널을 로드할 때 개별 핸들러를 동적으로 추가할 수 있기를 원하기 때문에 IRQ 핸들러는 좀 더 모듈식으로 정의됩니다. 이를 위해 이전에 정의된 레지스터_t를 취하는 함수인 인터럽트 핸들러 isr_t 배열을 초기화합니다.

```
typedef void (*isr_t)(registers_t *);

isr_t interrupt_handlers[256];
```

이를 기반으로 범용 irq_handler를 작성할 수 있습니다. 이 함수는 인터럽트 번호에 따라 배열에서 해당 핸들러를 검색하고 주어진 레지스터_t로 호출합니다. PIC 프로토콜로 인해 관련된 PIC에 인터럽트 종료(EOI) 명령을 보내야 합니다(IRQ 0~7 의 경우 기본값만, IRQ 8~15 의 경우 둘 다). 이는 PIC가 인터럽트가 처리되었음을 알고 추가 인터럽트를 보낼 수 있도록 하기 위해 필요합니다. 다음은 코드입니다:

```
void irq_handler(registers_t *r) {
    if (interrupt_handlers[r->int_no] != 0) {
        isr_t handler = interrupt_handlers[r->int_no];
        handler(r);
    }

    port_byte_out(0x20, 0x20); // primary EOI
    if (r->int_no < 40) {
        port_byte_out(0xA0, 0x20); // secondary EOI
    }
}
```

이제 거의 다 끝났습니다. IDT가 정의되었으므로 CPU에 로드하도록 지시하기만 하면 됩니다.

- IDT 로드하기

lidt 명령어를 사용하여 IDT를 로드할 수 있습니다. 정확히 말하면 lidt는 IDT를 로드하는 것이 아니라 IDT 설명자를 로드합니다. IDT 설명자에는 IDT의 크기(바이트 단위 제한)와 기본 주소가 포함됩니다. 디스크립터를 다음과 같이 구조체로 모델링할 수 있습니다:

```
typedef struct {
    uint16_t limit;
    uint32_t base;
} __attribute__((packed)) idt_register_t;
```

그런 다음 load_idt라는 새 함수 내에서 lidt를 호출할 수 있습니다. 이 함수는 IDT 게이트 배열에 대한 포인터를 가져와서 베이스를 설정하고 IDT 게이트 수(256)에 각 게이트의 크기를 곱하여 메모리 제한을 계산합니다. 평소와 같이 한계는 크기 - 1 입니다.

```
idt_register_t idt_reg;

void load_idt() {
    idt_reg.base = (uint32_t) &idt;
    idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) - 1;
    asm volatile("lidt (%0)" : : "r" (&idt_reg));
}
```

이제 모든 ISR을 설치한 후 IDT를 로드하는 isr_install 함수를 마지막으로 수정합니다.

```
void isr_install() {
    // internal ISRs
    // ...

    // PIC remapping
    // ...

    // IRQ ISRs
    // ...

    load_idt();
}
```

이것으로 IDT 섹션을 마치고 키보드 관련 코드로 넘어가겠습니다.

키보드 핸들러

키를 눌렀을 때 어떤 키였는지 식별할 수 있는 방법이 필요합니다. 이는 각 키의 스캔 코드를 읽으면 됩니다. 스캔 코드는 키를 눌렀는지(아래쪽) 또는 놓았는지(위쪽)를 구분합니다. 키 해제에 대한 스캔 코드는 해당 키 다운 코드에 0x80 을 더하여 계산할 수 있습니다.

스위치 문에는 현재 처리하려는 모든 키 다운 스캔 코드가 포함되어 있습니다. 스캔 코드가 이러한 경우 중 하나라도 일치하지 않는 경우 세 가지 이유가 있을 수 있습니다. 알 수 없는 키 다운이거나 릴리스된 키일 수 있습니다. 릴리스된 키가 예상 범위 내에 있으면 코드에서 0x80 을 빼면 됩니다. 이 로직을 print_letter 함수에 넣을 수 있습니다:

```
void print_letter(uint8_t scancode) {
    switch (scancode) {
        case 0x0:
            print_string("ERROR");
            break;
        case 0x1:
            print_string("ESC");
            break;
        case 0x2:
            print_string("1");
            break;
        case 0x3:
            print_string("2");
            break;
        // ...
        case 0x39:
            print_string("Space");
            break;
        default:
            if (scancode <= 0x7f) {
                print_string("Unknown key down");
            } else if (scancode <= 0x39 + 0x80) {
                print_string("key up ");
                print_letter(scancode - 0x80);
            } else {
                print_string("Unknown key up");
            }
            break;
    }
}
```

스캔 코드는 키보드에 따라 다릅니다. 예를 들어 위의 스캔 코드는 IBM PC 호환 PS/2 키보드에 유효합니다. USB 키보드는 다른 스캔 코드를 사용합니다. 다음으로 키 누름에 대한 인터럽트 핸들러 기능을 구현하고 등록해야 합니다. PIC는 IRQ 1 이 전송된 후 포트 0x60 에 스캔 코드를 저장합니다. 따라서 키보드_콜백을 구현하고 인터럽트 번호 33 에 매핑된 IRQ 1 에 등록해 봅시다.

```
static void keyboard_callback(registers_t *regs) {
    uint8_t scancode = port_byte_in(0x60);
    print_letter(scancode);
    print_nl();
}
```

```
#define IRQ1 33

void init_keyboard() {
    register_interrupt_handler(IRQ1, keyboard_callback);
}
```

거의 다 끝났습니다. 남은 것은 메인 커널 기능을 수정하는 것뿐입니다.

새로운 커널

새로운 커널 기능은 모든 조각을 하나로 모아야 합니다. ISR을 설치하여 IDT를 효과적으로 로드해야 합니다. 그런 다음 sti를 사용하여 인터럽트 플래그를 설정하여 외부 인터럽트를 활성화합니다. 마지막으로 키보드 인터럽트 핸들러를 등록하는 init_keyboard 함수를 호출하면 됩니다.

```
void main() {
    clear_screen();
    print_string("Installing interrupt service routines (ISRs).\n");
    isr_install();

    print_string("Enabling external interrupts.\n");
    asm volatile("sti");

    print_string("Initializing keyboard (IRQ 1).\n");
    init_keyboard();
}
```

이제 부팅하고 입력해 보겠습니다.

위와 같이 키보드를 입력하면 입력한 값이 나와야 하는데 아래와 같이 디스크를 인식하지 못하는 오류가 발생하여 원인을 찾아봐야 할 것 같습니다.

```
minsung@minsung-virtual-machine:~/dev/OS/KeyboardDriver$ make
qemu-system-i386 -fda os-image.bin
WARNING: Image format was not specified for 'os-image.bin' and probing guessed r
aw.
         Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```